Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies

FIIT-XXXXXX-82385

Tomáš Belluš

Bait network based monitoring of malicious actors

Master's thesis

Supervisor: Ing. Tibor Csóka, PhD.

2021, January

Slovak University of Technology in Bratislava

Faculty of Informatics and Information Technologies

FIIT-XXXXXX-82385

Tomáš Belluš

Bait network based monitoring of malicious actors

Master's thesis

Study program: Information Security

Field of study: 9.2.4 Computer Engineering

Training workplace: Institute of Computer Engineering and Applied Informatics

Supervisor: Ing. Tibor Csóka, PhD.

Departmental Advisor: Ing. Katarína Jelemenská, PhD.

2021, January

Annotation

Slovak University of Technology Bratislava
FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES
Degree Course: Information Security

Author: Tomáš Belluš
Master's Thesis: Bait network based monitoring of malicious actors
Supervisor: Ing. Tibor Csóka, PhD.
Departmental advisor: Ing. Katarína Jelemenská, PhD.
2021, January

Internal network, demilitarized zone (DMZ) or data pipelines have been compromised by a threat actor and the information gathered from this incident is minimal. Knowing the threat actor's agenda (entering, potentially leaving and fulfilling a goal) is more valuable, because it may lead to enforcing the system perimeter or endpoint security. By deliberately baiting access to highly monitored isolated networks, all further activities may be learned and enlighten a security engineer. This thesis, by utilizing state of the art container orchestration mechanism - Kubernetes, designs and implements a monitored isolated environment. Understanding and logging all file system changes, process executions helps to create a timeline of events constructing a possible incident. The resulting implementation is a robust proof of concept virtualized and completely automated immutable infrastructure monitored on the host machine level. Container observation mechanisms are capable, but not accurate enough to yet determine the order of file system events. Further design and implementation is required to identify the *point of enter* and *exit*.

Anotácia

Slovenská technická univerzita v Bratislave
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ
Študijný program: Informačná bezpečnosť

Autor: Tomáš Belluš
Diplomová práca: Sledovanie zlovoľných činiteľov nástražným systémom
Vedúci diplomovej práce: Ing. Tibor Csóka, PhD.
Pedagogický vedúci: Ing. Katarína Jelemenská, PhD.
Január 2021

Interná sieť, demilitarizovaná zóna DMZ alebo zreťazené spracovanie údajov sú kompromitované útočníkom, a získané informácie o tomto incidente sú minimálne. Vedieť útočníkovú agendu (od vstup cez vykonanie agendy až po prípadný výstup) je velmi cenné, lebo to môže viesť k vylepšeniu informačnej bezpečnosti sieťového okruhu alebo cieľových staníc. Úmyselným lákaním útočníkov na prístup k vysoko sledovaným a izolovaným sieťam, poskytuje možnosť pre bezpečnostného experta sa poučiť zo zlovoľných aktivít. Diplomová práca, využitím vyspelého orchestračného mechanizmu kontajnerov - `Kubernetes`, sa venuje návrhu a implementácii sledovania izolovaného prostredia. Porozumenie a zaznamenávanie všetkým zmenám súboroého systému a vykonávania procesov napomáha vytváraniu časovej osi udalostí spájaných do prípadného incidentu. Výsledná implementácia je rozsiahlým dôkazom predstavy ako kompletne virtualizovaná a automatizovaná nemenná infraštruktúra monitorovaná na úrovni hostiteľa. Mechanizmy sledovania kontajnerov sú funkčné, ale nie sú dostatočne presné v udávaní poradia výskytu udalostí o zmene v súborovom systéme. Preto je nevyhnutný daľší návrh a implementácia aj pre konkrétne idetifikovanie *bodu vstupu* a *výstupu* zo systému.

Tu vložiť zadanie diplomovej práce

Potom, vložiť finálny návrh zadania diplomovej práce

Contents

Chapter 1

Introduction

this thesis focuses on malware analysis in the Linux environment - the most used OS hosting application and systems in a company network.

`https://github.com/NodyHub/k8s-ctf-rocks` is proof that the thesis topic is valid and up-to-date in addition, the numerous Kubernetes vulnerabilities show that k8s clusters may become a victim of an attack

Chapter 2

Analysis

This chapter introduces malware analysis (see section 2.1) and differentiates the techniques (see subsection 2.1.1), outlines and describes mechanisms and environments utilized by cyber security professionals and companies (see subsection 2.1.2), describes the virtualization technology KVM (see subsection 2.2.1) and the orchestration mechanism Kubernetes (see section 2.3).

2.1 Malware analysis

Monitoring a malicious actor, sample or an entity in any environment is interchangeable from the analyst point of view. Therefore, knowing the malware analysis techniques is crucial for secure monitoring of malicious actors. Malware analysis may be static and dynamic with varying tools and mechanisms [17] [12].

2.1.1 Dynamic and static

Dynamic analysis is a process of actively monitoring, ideally in an isolated environment, the execution of a malware sample. Static analysis, as the name implies, inspects the function calls, readable strings, control and data flow of a malware sample (i.e. binary, program code).

Dynamic analysis may be unsafe and devastating, unless environment or system isolation is applied, but it's more precise and bypasses the reversing of self-modified code. Even though, dynamic analysis does not explore all execution paths, there are techniques resolving this drawback (i.e. virtual machine snapshots [12]). Additionally, dynamic analysis in an isolated or virtualization environment is exposed to the risk of isolation detection [10], meaning the executed malicious program stops or even self-destructs.

Static analysis is safe [and inspects all execution paths of the binary], but may be difficult to interpret when malicious actors utilize the obfuscation, compression or encryption techniques. Advanced malware sample poses a challenge for a malware analyst due the usage of control-flow flattening [26] and other methods. It suggests that dynamic malware analysis bypasses this troublesome procedure and discloses the sample outcome or agenda.

Manual Egele, et al. in their article [17] highlight the problems of static malware analysis approaches. The authors introduce multiple state-of-the-art techniques and tools dedicated to dynamic malware analysis - Function Call Monitoring, Function Parameter Analysis focusing of the final values, Information Flow Tracking, Instruction Trace and Auto start Extensibility Points (monitor startup programs, cron jobs, etc.).

2.1.2 Bait and execution environments

These mechanisms must provide solution to dynamic malware analysis limitations in order to effectively capture the malicious actor's agenda. It should be a robust system implying the impression of a production environment.

Honeypot and honeynet

Honeypot is a bait service, system or a even whole network (honeynet) usually hosted on public server. Its main purpose is to be scanned, attacked or compromised by the malicious actor. Every honeypot provides the desired functionality of the target resource, to mimic the production environment, leaving the malicious actor unaware of the honeypot [23].

Based on the ENISA honeypot study [23], honeypots are classified from the level of interaction view and based on the attacked resource type. The Low Interaction Honeypot (LIH) provides very low availability of the host OS. Most services and application are mocked and simulated in a static environment. Everything accessible is controlled by a decoy application with absolutely minimal in-depth features (e.g. shell, configuration files, other programs etc.). LIHs are more secure for the host, but far less capable or useful for malware/attack detection and inspection.

On the contrary, High Interaction Honeypot (HIH) is a fully responsive system with live applications and services with minimal to none emulated functionalities. It provides the attacker a wide attack surface ranking the HIH far less secure with the whole OS at the malicious actor's disposal. The idea is to make HIHs believable as possible and isolating it from production environment including virtualization [46].

Honeypots are differentiated by the type of attacked resource. The server-side honeypot is the well-known honeypot with running service(s) and monitoring the activity of the server-side connections. The attacked resources are the services listening on the dedicated ports. It's main purpose is to detect and identify botnets and forced authentication/authorization attempts.

The client-side honeypot is deployed as a user application, which utilizes the server's services. The monitored subject is the application (e.g web-browser, document editor). It's main purpose is to detect client-side attacks originating from the application (i.e. web-browser attacks via web pages and plugins).

Sandbox

In the security branch, sandbox is used for controlled both static and dynamic malware analysis as mentioned in subsection 2.1.1. It is dedicated for execution or testing of unverified functionality, behavior of programs or other activity [39]. Moreover, the environments hosting or serving as a sandbox must be isolated to secure any spreading. A possible attack of this kind is referred to as the escape attack, which is a vulnerability allowing a malicious actor to move from virtualized/containerized environment to the host system.

"When a threat is contained within the sandbox environment, it is quarantined and available for study by the in-house IT team or external cybersecurity experts" [39]. Slightly changing the notion makes a sandbox technology ideal for active and live analysis of malicious actors. Other benefits are flexibility of easily changing the environment to adapt to the current scenario.

2.2 Virtualization technology

The section discusses thesis related virtualization technologies kernel virtual machines (KVM), Docker containerization and the Kubernetes orchestration mechanism. Together these technologies may create a sandboxed honeynet holding theoretically multiple environments, thanks to the Kubernetes capabilities.

2.2.1 Kernel-based Virtual Machine

KVM is a type-1 (bare-metal) hypervisor enabling technology used for virtualization of full operating systems. The core functionality is based on the kernel capabilities to manage the hardware resource, hence the name **Kernel-based** VM [45]. KVM is native to the Linux as a kernel module, which requires the host machine processor virtualization support and by utilizing them, multiple VMs may coexist on one machine [1].

KVM, QEMU and Libvirt

Although, KVM is not a complete solution without the *libvirt* virtualization API daemon and the *QEMU* hardware emulation [2]. When referring to KVM, it is often substituted for both KVM and QEMU. The combination provides a performance considered nearly native-like [1]. While the KVM kernel module translates code in the kernel-space, QEMU provides the virtualization and emulation of hardware resources. In addition, Libvirt is a virtualization management tool and API. It is fully configurable by automation mechanism to automate the whole process of VM creation.

Libvirt is the solution to the variations of virtual machine images, since each

technology has a different format. For example the storage backend provides the same API of so called volumes for any image format [32]. Apart from all of the libvirt objects available, networking is the most relevant. According to the API specification [31], the configuration options are for both VM network and host interface with DHCP, DNS and NAT. Additionally, there is also support for *openvswitch*, which is particularly useful for more complex network architectures with VLANs and other protocols and standards.

Nevertheless, libvirt and thereby KVM/QEMU is compatible with the Linux bridge [3], which is a simple virtual network component used for connecting networks. Creation and maintenance is straightforward and the functionality is effective enough to connect the VMs with the host machine. An advantage is it can be managed by various control tools (e.g. *ip command*, *ifupdown*, *bridge-utils* and other), which provides the separation of the development environment setup and the main host environment.

Vagrant

According to official documentation[4], Vagrant is a VM provisioning technology and one level up in terms of KVM virtualization. Meaning, it adds another configuration layer for a whole multi-VM environment. It composes of manifest file called Vagrantfile. This file is a ruby program using specific calls to defined and configure multiple VMs. Vagrant is ideal for completely bypassing the direct usage of Libvirt and instead utilize the Vagrant libvirt provider[1]. In a Kubernetes blog [29], the author describes multi-node Kubernetes cluster deployment using Vagrant with Ansible. It is practical approach to deploy a custom base environment with additional configuration using third-party tools e.g. Ansible.

[1] Open-source solution to the vagrant libvirt provider - `https://github.com/vagrant-libvirt/vagrant-libvirt`

```
ENV["LC_ALL"] = "en_US.UTF-8"
BOX_DISTRO = "generic/ubuntu1804"
Vagrant.require_version ">= 2.0.0"

Vagrant.configure("2") do |config|

    config.vm.define "test" do |node|
        node.vm.box = BOX_DISTRO

        # node-specific libvirt settings
        node.vm.provider :libvirt do |domain|
            domain.cpus = 2
        end

        # configure a network interface connected to
        # a network with specific address
        node.vm.network :private_network,
            :type => "dhcp",
            :libvirt__network_address => '10.20.30.0'
    end

    # global libvirt settings
    config.vm.provider :libvirt do |libvirt|
        libvirt.driver = "kvm"
        libvirt.storage_pool_name = "default"
        libvirt.memory = 2048
    end
end
```

Listing 2.1: Working Vagrantfile example with the 'generic/ubuntu1804' using the libvirt provider and configuring 2GB or RAM, 2 virtual CPUs, DHCP network interface, libvirt storage pool to 'default'.

The fact that Vagrantfile is a ruby program provides the freedom of not being bound to a set of configuration variables and other related limitations. The definition is enclosed in `Vagrant.configure()` call and further divided in to `config.vm`, `config.ssh` and `config.vagrant` sections. `config.vm` is the con-

text for configuring the VM directly with Vagrant e.g. vagrant box, provider, networking, disk. `config.ssh` is for configuring SSH access for the vagrant utility and `config.vagrant` is used for modifying Vagrant itself like plugins.

The API is unified across all Vagrant providers and is enriched by provider-specific variables as the `libvirt__network_address` on line 19 in Listing 2.1. With the capabilities of *vagrant-libvirt* provider, the VM can be defined is this one file - Vagrantfile.

Another term is the Vagrant box. It is a specifically formatted unit packaging the VM image, which is ready for deployment. A box can found in public repositories e.g. Vagrant Cloud[2] similarly to the Docker Hub. Benefit is the customization and direct workflow when creating a box, because then it can be reused for other VMs are share the same configuration and setup.

2.2.2 Docker

Docker is very popular containerization technology used by developers and in production environments. Therefore a detailed explanation is not in scope of this thesis. On other hand this section covers deeper aspects of Docker and its mostly containers.

Similarly to Vagrant (see section 2.2.1), a Docker image is defined in a manifest file called Dockerfile. The image is used to create a Docker container as a single unit. Multiple containers can share an image with different configuration to adapt to the current environment e.g. database configuration, port publishing, storage mounting.

But, each container is uniquely identified by an ID and other resources and objects

[2] `https://app.vagrantup.com/boxes/search`

belonging to the single container e.g. mapped network interface, storage volumes. When not specifically configured, both networking and volumes/storage would be local to the container and be removed along with the container. In either case the data, networking and yet unmentioned processes are accessible from the host system. Main reference for the base information is the `docker inspect` command, which returns every settings, mounted directory, networking configuration and execution metadata.

Julia Evans constructively discusses various container spying techniques dedicated to Docker in a video demonstration [20]. Apart from Docker-specific spying with the `inspect` command, there is `nsenter` and the inspection of `overlayfs`. Nsenter is nearly equivalent to the `docker exec` command, but it executes a command with a specified namespace [19]. Despite knowing the container namespace, it has no requirements and is Linux native. The advantage over `docker exec` is that for nsenter the executed command/program (see Listing 2.2) does not needs to be installed in the target container.

```
root@node2:~# nsenter -t 6234 -n ip a
```

Listing 2.2: Executes the `ip address` command within the network (option `-n`) namespace of a taget process with ID 6234

"OverlayFS is a modern union filesystem that is similar to AUFS, but faster and with a simpler implementation" [43]. As the Docker image is build, each compact change creates a new layer of the container root file system linked to the previous layer and including the difference. These layers can be inspected in the host system as bare directories. Mounted volumes are no exception and both can be identified from the `docker inspect` command output. */var/lib/docker/overlay2* stores all these layers (see Listing 2.3).

```
1 root@node2:~# find -L /var/lib/docker/overlay2/l/ -name shadow
```

```
2  /var/lib/docker/overlay2/l/7XOWKH6XWUWBNEQV7OJ2ZB5YRG/etc/shadow
3  [.. OUTPUT TRIMMED ..]
4
5  root@node2:~# cat /var/lib/docker/overlay2/l/7
       XOWKH6XWUWBNEQV7OJ2ZB5YRG/etc/shadow
6  root:*:::::::
7  daemon:*:::::::
8  bin:*:::::::
9  sys:*:::::::
10 sync:*:::::::
11 mail:*:::::::
12 www-data:*:::::::
13 operator:*:::::::
14 nobody:*:::::::
```

Listing 2.3: Container files, independent of the layer, are accessible from the host system. For example output the contents of an arbitrary file

Useful information in the `docker inspect` output:

- Main process ID.
- Path to the top layer of the overlay file system.
- List of mounted volumes and directories.
- Network interface.
- Environment variables.

2.3 Kubernetes

This section covers the Kubernetes concepts focusing on the aspects essential for properly organizing an efficient setup. That includes Kubernetes nodes, Pods, Deployments, Services and persistent storage resources. *Disclaimer: the following concept definitions are derived from the official Kubernetes documentation page* [13].

2.3.1 Components and nodes

Kubernetes is a platform for orchestrating multiple containers among its cluster nodes. A functional cluster comprises of control-plane components i.e. API server, *etcd*, Pod scheduler, controller manager and cloud-specific controller manager. The *kubelet*[3], *kube-proxy*[4] and container runtime components run on every node, wheres the control-plane components aside on a single node - control-plane node[5]. All other nodes are called workers.

When creating or updating a cluster, nodes are registered in the API server by kubelet or manually. A node must be considered healthy to run a Pod, which si important because the environment/cluster may be highly dependent on the node's computational resources. The declarative nature of Kubernetes simplifies the creation of development environments. The configuration defines a desired state of an application ecosystem, microservices or other custom environment and the control-plane schedules and assign the Pods to suitable nodes [9].

2.3.2 Resources

Kubernetes environments are build upon API resources or objects.

Workloads

Mainly the smallest running unit - Pod encapsulates one or more containers, storage resource and a unique network identifier. Best is to run single application

[3]Communicates with the API server on the master node. Ensures that the Pods are up and running

[4]The node networking endpoint for Pod communications.

[5]Also referred to as the master node. The current Kubernetes documentation does not mention this node role anymore.

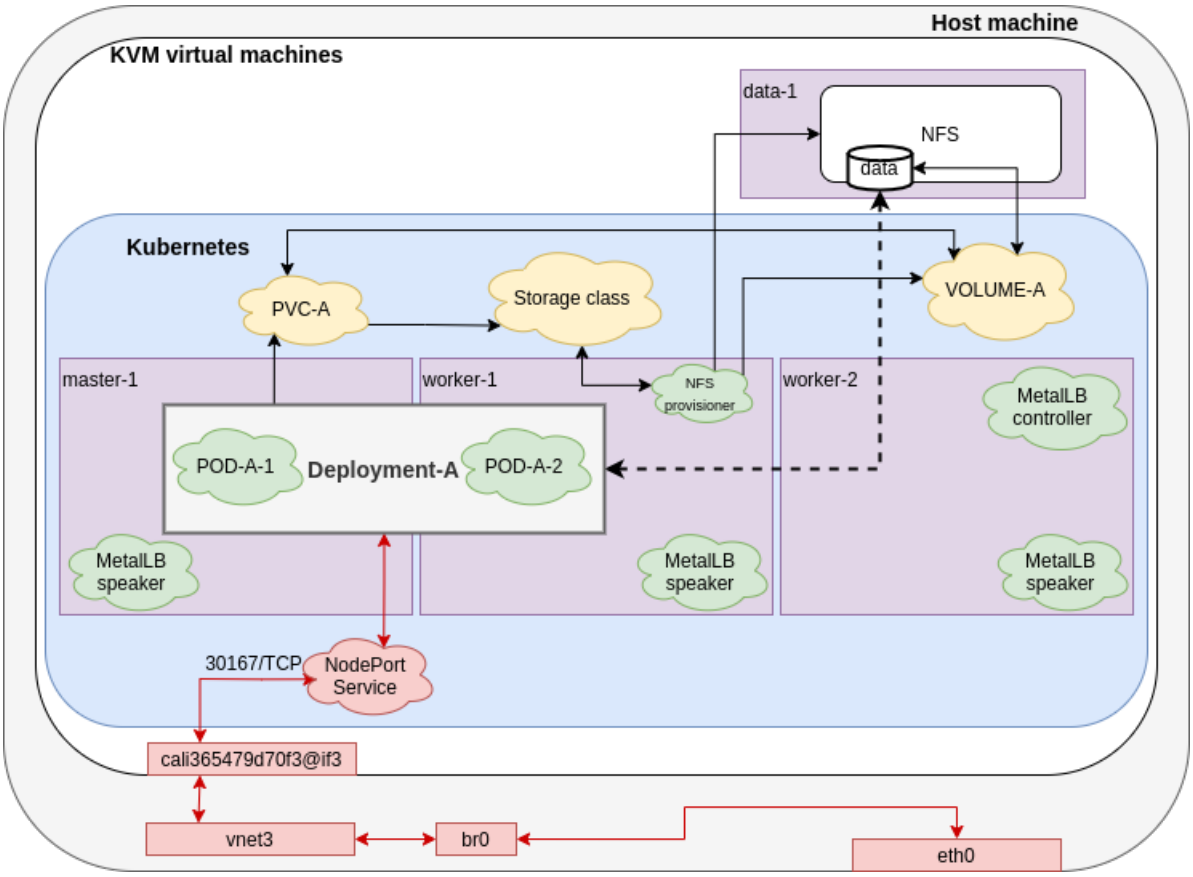

Figure 2.1: Kubernetes sample architecture with Pods, Deployments, Services and Volumes. A 'cloud' is a Kubernetes component (green = Pod, yellow = volume-related objects, red = Service and network-related objects). Lines represent communication links and the dotted lines are abstract/logical connections.

instance per Pod [37]. Above is a ReplicaSet[6] and Deployment, which at first glance fulfills the same, except that Deployment (see the Deployment object in Figure 2.1) is a abstraction above Pods and ReplicaSets combined. Kubernetes suggests that "you may never need to manipulate ReplicaSet objects: use a Deployment instead" [38]. Deployments allow rolling updates of both ReplicaSets and Pods. In comparison to *docker-compose*, the definition of the container(s) in the Deployment resource is very syntactically similar.

[6]Definition of horizotally scaled Pods with a specific number of replicas.

Volumes

Kubernetes, dy default, does not provide data persistence, as it is in docker by design. There are various options of mounting volumes to Pods via PersistentVolumes (PV) e.g. cloud-specific volume plugins or *hostPath*. The PV is linked to a special resource PersistentVolumeClaim (PVC), which is manually created alongside a Pod and binds to a suitable PV object [36]. In other words, PVC is a request for assigning a PV to Pod/Deployment. Although the PV must be created and meant all requirements of the PVC in case of successful binding. For that reason a PV template called StorageClass exists, which dynamically creates PVs for subsequent binding (refer to Figure 2.1 for graphical representation of the relations).

Since for this thesis the Kubernetes cluster is local and bare-metal (no cloud and in VMs) and hostPath expects the Pod(s) to remain on the same node[7] [44]. The Network File System (NFS) is supported on any Linux system (as a NFS server) and has Kubernetes provisioner integration [30].

Service

By default every Pod running a network service is unable to communicate with the outside network or other applications within the cluster, except for the shared containers in the same Pod. The Service resource cooperates with the kube-proxy component, which assigns the Service a virtual IP address (also called ClusterIP) visible only inside the cluster [40].

The level of access via the Service depends on its type - ClusterIP, NodePort, LoadBalancer and ExternalName. To open a Service to the outside network (see

[7] This can be resolved by using Taints and schedule the Pod only on the specified node. Although this beats the purpose of Kubernetes

the red section in Figure 2.1) any type but ClusterIP serves this purpose. The ExternalName requires a functional kube-dns component and NodePort does not map the service port to a reserved port, but instead uses a port from immutable range (30000-32767) [40].

A LoadBalancer is a cloud provider-specific Service type combining NodePort and Cluster in a way to expose the Service over a public IP address. An advantage is an existing bare-metal implementation[8] and the ability to expose reserved ports (e.g. 22, 80, 443).

2.4 Environment monitoring

Knowing what and how to extract and monitor is vital to understanding the threat actor's agenda. There are various mechanisms and possibilities to effectively observe container activity events i.e. file system, networking and process execution. Full related solutions are documented in the next chapter (see chapter 3). Regarding the expected setup of virtual machines hosting the Kubernetes cluster, the possible monitoring strategies are:

- From within the container as an agent reporting events over an associated Service.
- Abuse the container-host mapped resources (e.g. network interfaces, container mounted directories) and stealthily spying.

Container monitoring from the container itself may appear more straightforward, but it's similar to monitoring of a honeypot, where agents could be discovered or in other way uncovered by the threat actor. Since containers are wrapped

[8]MetalLB - Metal Load Balancer is a bare-metal LB implementation utilizing either a RM-OSI layer 2 routing or BGP [11].

environments hosted on the host machine, the file system, networking processes are readable. The apparent advantage is a transparent view of any monitoring for the threat actor, since the container has an enclosed environment by namespaces [21]. For example docker with the *overlay* file system driver has a predefined location where each container root FS exists (unless the volume binding overrides this).

2.4.1 Existing tools

This section outlines a briefly discusses various applicable and suitable container activity monitoring tools.

BPF Compiler Collection (BCC)

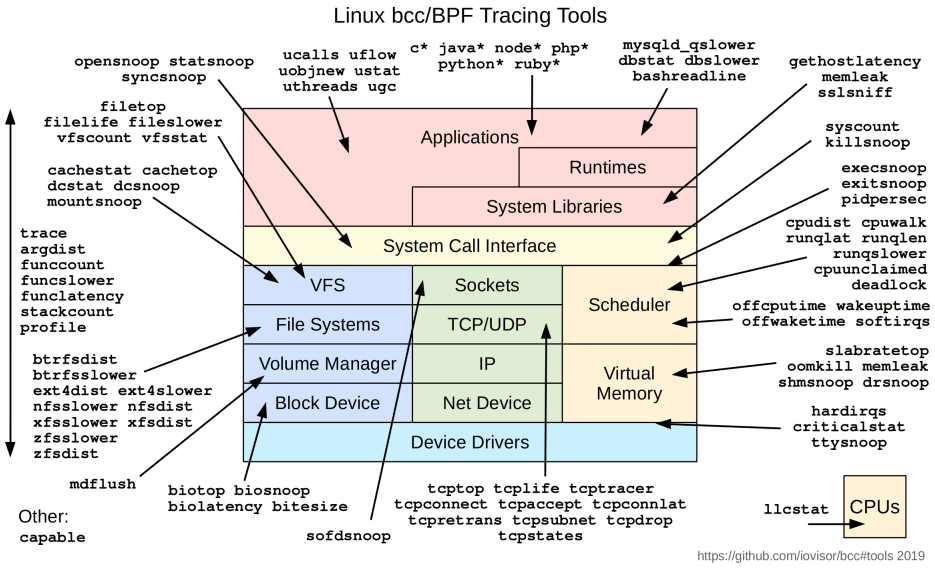

Figure 2.2:

It's a state-of-the-art monitoring tools bundle used by large companies (e.g. Netflix, Facebook) [22]. BCC serves as front-end (CLI) program to various eBPF-

based tools. Concretely, *execsnoop*, *opensnoop*, *tcpaccept*, *tcpconnect* and *tcplife* are programs tracing `exec` syscalls, `open` syscalls, tcp `accept` and `connect` syscalls and tcp connection sessions respectively.

Some BCC tools have a mount namespace option to filter the monitored events. This can potentially select a specific container. The mount namespace is passed via a BPF map object created by the *bpftool* tool [41].

monks

Monks[9] works as a kernel module hijacking system calls on the target system. "Monks is like strace, but tracing all and every single process from any user, at any level" [35].

execmon

Execmon[10] is a similar utility intercepting the `execve` system calls in kernel-space and notifies the user. It's a kernel module combined with a user-space utility receiving events.

fswatch

For file system monitoring there is a CLI program *fswatch* [11] acting as the API to the *libfswatch*, which is ultimately an API to *inotify*. It provides real time FS monitoring with a second accuracy. Fswatch distinguishes events based on the action (e.g. created, renamed, removed, owner changed).

[9] `https://github.com/alexandernst/monks`
[10] `https://github.com/kfiros/execmon`
[11] `https://github.com/emcrisostomo/fswatch`

Chapter 3

Related Work

Honeypot, sandbox and deception technology make up the leading techniques in dynamic malware analysis. They differ in the scope of knowledge before the analysis at hand. Some know the filename, malware type, other artifacts and possibly the expected outcome (in which case the tool observes only the behavior). Other analyze the behavior of all activity - searching for anomalies and malicious indicators of compromise. The following sections briefly introduce the malware analysis, deception technology, honeypot and other existing solutions and studies relevant to the thesis topic.

3.1 Malware file analysis solutions

Malware file analysis is a dynamic procedure when the filename and possibly malware type is known. In comparison to the scope of this thesis, all use similar techniques of malicious activity observation/monitoring, but differ in use case scenarios.

3.1.1 Cuckoo sandbox

A most common sandbox environment for malware analysis by executing a given file in an isolated environment with reporting of the outcome [5]. All files affecting mainstream operating systems i.e. Windows, MacOS, Linux and Android are supported. In addition to known artifacts, cuckoo has no interfering processes, so all traces must be followed and provide insight to the behavior. Based on the official cuckoo documentation, the system produces various results (the following artifacts are copied from the documentation site [5]):

- Traces of calls performed by all processes spawned by the malware.
- Files being created, deleted and downloaded by the malware during its execution.
- Memory dumps of the malware processes.
- Network traffic trace in PCAP format.
- Screenshots taken during the execution of the malware.
- Full memory dumps of the machines.

Despite all differences with this thesis, cuckoo's architecture consists of the management software (host machine) and a number of virtual/physical machines for analysis. It's a tool for different use case, so a comparison is insignificant.

3.1.2 Droidbox

Another open source tool droidbox [6], sadly discontinued several years ago, analyzes android applications using Android Virtual Devices (AVD) and the android emulator, which supports the Android activity monitoring. Analyzed applications are sandboxed in the AVDs and afterwards reports the following results (the following artifacts are copied from the documentation site [6]):

- Hashes for the analyzed package.
- Incoming/outgoing network data.
- File read and write operations.
- Started services and loaded classes through DexClassLoader.
- Information leaks via the network, file and SMS.
- Circumvented permissions.
- Cryptographic operations performed using Android API.
- Listing broadcast receivers.
- Sent SMS and phone calls.

Overall, droidbox introduces a simple way of analyzing android applications via an existing API of the emulator.

3.1.3 Virustotal

Similarly to cuckoo, virustotal utilizes both static and dynamic malware analysis. "VirusTotal's aggregated data is the output of many different antivirus engines, website scanners, file and URL analysis tools, and user contributions" [25].

3.1.4 Falcon sandbox

A direct equivalent to virustoal is a less famous tool called Hybrid Analysis [1] powered by the Falcon sandbox. Again, it's similar to cuckoo, except the anti-evasion feature [16], which allows, even sandbox-aware malware, to be analyzed despite their evasion techniques.

[1] `hybrid-analysis.com`

3.1.5 Evaluation

Most of these are the same in concept and usually in realization too. The sandbox analysis techniques are concentrated on malware files rather than an actor's activity in an environment sneaking around a system. Nevertheless, the collected data for reports and some architectural aspects are very valuable for this research.

3.2 Active analysis

This section explores existing honeypot/honeynet technologies and a recently emerged concept - deception technologies. These technologies may be divided into two categories - dynamic [34] and static, where the environment adapt to the scenarios or remains unchanged respectively.

3.2.1 Honeystat

Honeystat[15] is a honeypot solution observing the behavior of the Blaster worm and may be used to detect zero-day worm threats. Authors assume the infection may be described in a systematic way, so by knowing the worm agenda and steps they model the monitoring procedure. The observation is event-based with memory, disk and network events. Since there are no regular users in the system, the memory events are all interesting violations e.g. buffer overflows. Disk events are file system modifications and network events should always be infection over related outgoing traffic. Worms require a multi-host network to have spreading possibility, so honeystat is deployed in a multi-homed VMWare environment (64 VMs * 32 IP addresses = 2^{11} IPs) with minimal honeypots. The procedure when events are encountered is:

1. The honeystat is capturing memory and disk events.
2. If a network event occurs, the honeypot is reset to stop further spread of the worm to other machines/honeypots.
3. Any previous memory/disk event is updated with additional information from the network event.
4. Resets ought to be faster in virtual environment. Host VM is not rebooted, only the virtual disk (VD) is kept in a suspended state before it's replaced with a fresh copy of a VD. The reset always completes before a TCP timeout.
5. Other steps include an analysis node, which is out of scope of this thesis.

This solution does not introduce any isolation techniques beside utilizing virtualization. The emulation mechanisms are exposing the virtualized environment via e.g. BIOS strings or MAC address. All features and considerations for honeystat are purely for worm infection detection, other infection types could require more observables.

3.2.2 Honeypoint

Service emulation is what Honeypoint [2] utilizes to lure malicious actors and detect their agenda. Production services lie in the same environment as the robust architecture of Honeypoint, which can mimic a complex network environment for deceiving an attacker. The Microsolved CEO Brent Huston claims [27] that having a honeypot is a great deception technology with almost no false positives, since it is expected that no legitimate user interacts with it. It means that any recorded activity should be considered suspicious, if the honeypot targets malicious actors scanning the Internet regardless of possible domain - randomly trying IP addresses and looking for a services ought to have malicious intent. Consists of various com-

[2] `https://www.microsolved.com/honeypoint`

ponents [28] that could be replicated in the Kubernetes architecture design.

3.2.3 Cybertrap

A (commercial) solution Cybertrap [18] operates as a deception technology luring attackers away from production systems. Looking apart from that services in Honeystat are emulated, Cybertrap's deployed services cannot be distinguished by the attacker. Once the malicious actor gets inside such network, all movements are tracked. In addition, the Cybertrap's network is inaccessible by regular users, so any activity within the simulated environment is consider malicious - minimal to no false positives. Cybertrap is close to the idea of the goal of this thesis - sandboxed honeynet.

3.2.4 A distributed platform of high interaction honeypots and experimental results

A case study [42] serving as a proof of concept in live Internet traffic observes malicious actors' trends and agenda. As a monitoring technique they patched the kernel's `tty` and `exec` modules to intercept the keystrokes and system calls respectively. The architecture is 4 machines anywhere in the world working as relays to the authors' local setup of virtual honeypots. The traffic incoming to the public interface of the relay is routed to a GRE tunnel connected to the local VM.

In a SSH scenario they created a new syscall and modified the SSH server to use it in order to intercept the login credentials. Logged data is periodically copied from the VM disk to the host disk (such extractions should be undetected by the malicious actors). All login data is stored to the database of this structure:

- Data from each ssh login attempt.
- Data from each successful ssh connection - tty buffer content and tty name.
- Data of programs executed with parameters and the terminal in which it ran.
- Session data grouping ssh connections.

Experiment

In the period of 30 days, they monitored what are the most common login-password pairs when no accounts exist on the target system. It resulted in low number of high frequency pairs. Subsequently, for half a year they monitored the time it took a threat actor from a successful login to running command under the same account. The results showed some attackers managed to escalate privileges to the root account and in some cases even changing the passwords of the other accounts on the system. In addition to executed commands, they observed the attackers were attempting to download programs from websites, which were hosted in the same country as the attacker's IP address. Furthermore, below are listed general trends of successfully logged in attackers:

- Checked who else is active on the system.
- System reconnaissance - OS name and version, processor characteristics, etc.
- Changed the password of the current user.
- Install an IP scan program and scans the IP range to proceed for potential lateral movement.
- Internet Relay Chat client setup for receiving instructions.
- Privilege escalation attempt.

In comparison, general trends of attacker behaviors with root privileges:

- Change the root password.
- Setup backdoor - open a network port future logins.
- Checkout information about legitimate users of the computer via custom installed software.
- One attacker replaced the ssh client binary.

3.2.5 SIPHON

A case study [24] on Scalable High-Interaction Physical Honeypots (SIPHON), similar to the study before, serving as a proof of concept in live Internet traffic observing the IOT related malicious intents. Leveraging Shodan to appear visible and legitimate in the eyes of malicious actors, the honeypots where based on real devices. The architecture is divided into physical IOT devices, wormholes exposed to the Internet forwarding to the IOT devices via the proxy forwarder. Technically are devices separated using VLANs 802.1Q and the wormhole to forwarder connection are via reverse ssh tunnels. As compromise countermeasures the *suricata* IPS and IDS features are enabled in the local network and periodic resets of IOT devices.

They observed the influence of device listing in Shodan. The number of scans/-connection attempts on the device has tripled between 'one week before listing' and 'one week after listing'. It proves that being visible by Shodan increases the possibility of attack reconnaissance on device at hand. Although, after two week after listing in Shodan, the connection attempts has decreased, which good piece of knowledge before implementation.

3.2.6 Evaluation

Most of the solutions proposed introduce practical and efficient techniques such as in subsection 3.2.4 they modified the ssh server to use a custom system call, or monitoring in event based fashion (see subsection 3.2.1). All were similar in hiding their true intent and efficiently retrieve relevant facts towards identification of the attacks agenda.

3.3 Kuberentes specific solutions

3.3.1 alcide

TODO?

Chapter 4

Design

This chapter focuses on the overall design of the solution, depicting the crucial parts. Firstly, the base system of the monitored environment. Secondly, container remotely controlled event-based monitoring of processes, file system changes and networking. Thirdly, deployment and emplacement of the isolated environment and additionally connecting it over network in a target facility or system.

4.1 Specification

This short section summarizes all specifications and assumptions considering the design. The target operating system is always Linux distribution Ubuntu 18.04.5+ with hardware enablement (HWE) or 20.04.x. The kernel specification is important for the machine hosting VMs to satisfy eBPF tools. In addition, the KVM (with QEMU) is used with the libvirt management library, which fully satisfies the choice of virtualization technology.

Regarding the VMs, there are no kernel specification, but they have Ubuntu

18.04.5+ or 20.04.x. Additionally, each VM has the minimum memory size of 4 GB and 2 virtual processors (vCPU). Although, this can vary depending of the host system resources.

4.2 Environment architecture

The environment has a solid underlying architecture considering a proper isolation layer and other prevention mechanisms to secure the hosting system. These operations and precautions are in compliance with the ISO 27002 standard. The bait environments within, are deliberately vulnerable in some way, therefore they are not designed to abide much of any of the ISO 27000 standards. The main goal is a mimicking production environment build atop of Kubernetes cluster. Sequentially, this section covers all in a bottom-up fashion through Kubernetes cluster design to system environments.

The considered specific control categories of the "Information technology – Security techniques – Code of practice for information security controls" - ISO 27002 are[1]:

12. Operations security
    1. Operational procedures and responsibilities
    2. Protection from malware
    3. Backup
    4. Logging and monitoring
    5. Control of operational software

[1] All of the outlined categories are derived from the official standard [7] with proper numbering preserved

13. Communications security
    1. Network security management
14. System acquisition, development and maintenance
    2. Security in development and support processes

Before implementation, the whole solution including program code, configurations and operation process must be documented with proper backups in terms of these control categories. According to ISO-27002 category 12.1 part "12.1.1 Documented operating procedures", the solution must have a technical documentation of the operations, which means including installation, setup and configuration procedures whether they are automated or not. This category links to ISO-27002 category 14.2, which discusses the security of version control systems and remote repositories used for sharing and archiving. All developed programs have a dedicated repository with no sensitive configurations, which could compromise the system. Last but not least, ISO-27002 category 12.3 refers to creating and maintaining proper backup procedures of valuable assets. Partly it correlates with the previous categories and only together function as secure, documented, archived and reproducible in case of any failure.

4.2.1 Base system

The lowest layer is an Ubuntu host machine with KVM virtual machines (VM). Deriving from a Kubernetes cluster design in production environment, which often isn't a single-node architecture, the base system must be a set of coexisting VMs. Choosing the right number of VMs with respect to the overall resource capacity is not in scope of this thesis. Nevertheless, there are three base VMs serving as the base of the deceiving system.

The main specification defines that all VMs are identical and configured for remote operations, have functional inter-VM communication and meet all requirements (e.g. kernel version, basic security settings, hardening). Creating multiple identical VMs can be achieved by preceding mechanisms (e.g. *cloud-init*[2]), sharing the same image or by provisioning mechanisms (e.g. *Vagrant*, *Ansible*).

It depends on the implementation, but since these VMs have a static and simple setup, preceding is not necessary. This technique is fully configurable and must be automated or manually ran for each VM with deviating variables e.g. host name, IP address. This is not a complicated process, but a more suitable approach is using a shared pre-configured image combined with a dedicated tool - Vagrant for seamless provisioning and installation. KVM with libvirt and Vagrant create a "VM as code" concept efficient in provisioning, preceding and whole VM management.

Setup

Base system has several dependencies and requires a custom setup of networking and host machine altogether. As mention in section 4.1, the host depends on QEMU, KVM and libvirt to run VMs. Additionally, the VMs are connected to a libvirt-managed management network (MGMT network) and a standard inter-VM network (NODE network) also simulating public IP address pool for the Kubernetes cluster. Both of these networks are represented as Linux network bridges and are segregated to comply with ISO-27002 category 13.1.

Given that, the deceiving environment is isolated twice (Kubernetes-managed containers and VMs), the host machine is not expected to experience any malicious activity. Even though, a set of precautions is advised in case of any suspicious

[2]https://github.com/canonical/cloud-init

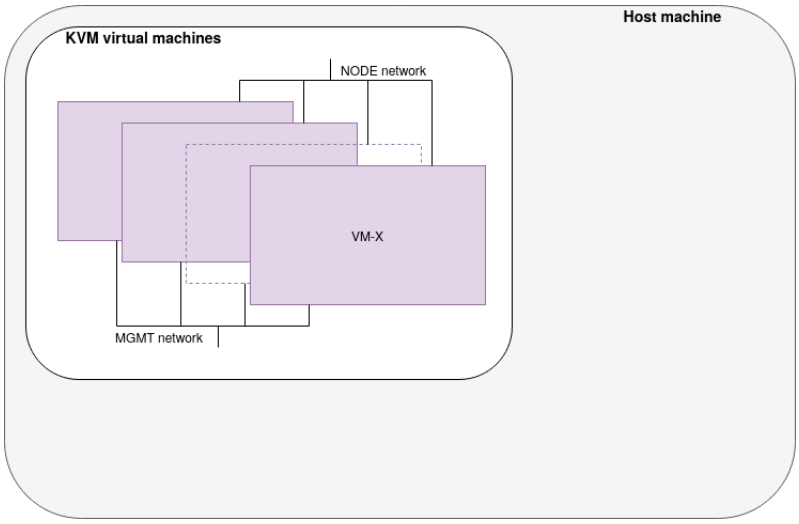

Figure 4.1: Base system visualization. !!!!REVIEW ME!!!!

activity occurs otherwise. But only passive techniques are suitable, because all of activity from the isolated environment is routed through the host system, which must be allowed. Network security monitoring (NSM) solution designed for detection rather than prevention should effectively provide the sufficient visibility over the host system. Although, this is not the main thesis objective, so there are no specific requirements and it depends on the implementation.

Vagrant

Sharing a custom base image (Vagrant box), saves time on configuration and is less prone to error. Utilizing a pre-configured Vagrant box does not necessarily mean the configuration is immutable, all can be changed via Vagrant post-deploy commands and provisioning in the *Vagrantfile*. Security-wise, creating a custom Vagrant box from the official Ubuntu image is recommended over publicly available Vagrant boxes from unverified owners. Not knowing the whole agenda of those boxes a full audit would be appropriate to approve their usage. Therefore, the all

VMs have been created with a specific Vagrant box.

Each Vagrant box is a minimal Ubuntu (of satisfying version defined in section 4.1) installation with the following configuration and setup:

- user setup including password protected *root*
- kernel parameters
  - enabled IP forwarding - `net.ipv4.ip_forward`
- VM routing table entry to satisfy reverse path check - new route through the NODE network to the administration network or machine
- SSH daemon configuration
- custom dependencies based on the implementation

Additionally everything else is done within the Vagrantfile, which is configurable with environment variables or other type of arguments. The input variables are the NODE network bridge name, IP prefix for the NODE network, number of nodes and vagrant box identifier. Altogether, the Vagrantfile creates all requested nodes as functional and remotely accessible VMs ready for Kubernetes installation and the deception environment configuration.

4.2.2 Kubernetes cluster

Kubernetes is complex and highly configurable, therefore a simple configuration is sufficient. There are many Kubernetes installation techniques and various derivatives meant for minimal setup and development (e.g. *microk8s*[3], *minikube*[4], *k3s*[5]). For this thesis a full Kubernetes ecosystem is preferred to mimic a production environment as much as possible.

[3] `https://microk8s.io/`
[4] `https://minikube.sigs.k8s.io/docs/`
[5] `https://k3s.io/`

Setup

Considering Kubernetes as a cloud-only platform, there are few differences to take into account when deploying microservices and applications on a bare-metal variant. Most importantly, Kubernetes is installed on those three base VMs in an arrangement of one master node with two worker nodes. Which briefly means, that the master node controls the cluster and does not provide any computational resources like the worker nodes do.

Things like storage and load balancing network traffic, are cloud provider services, which need to be substituted. Regarding persistent storage for demanding applications, the cluster is scaled up with additional dedicated node (data node) for storage. Data node is not part of the Kubernetes cluster, it serves as an external Network File System (NFS) server providing persistent storage. This node is setup in the same way as the other cluster nodes, except for the Kubernetes installation. Instead, setup with simple NFS server.

The Kubernetes ecosystem is installed via *kubespray*[6], which is a full Ansible skeleton for a complete cluster setup. Kubespray installs all dependencies, configures networking and assigns roles (master vs. worker) to all nodes. The cluster is ready for creating new objects, applications and environments.

Only after the cluster is fully functional, the before mentioned load balancing can be setup. For a bare-metal installation, the *MetalLB*[7] load balancer effective and provides full Kubernetes loadbalancer capabilities. MetalLB is setup in layer 2 mode, which refers to the data link layer of the RM OSI model. This mode utilizes the address discovery protocol (i.e. ARP) to route between nodes and turns one node into a point of enter. It is not a true load balancing technique, but it serves

[6] `https://kubespray.io/`
[7] `https://metallb.org/`

the key purpose of publishing common ports to the outside network.

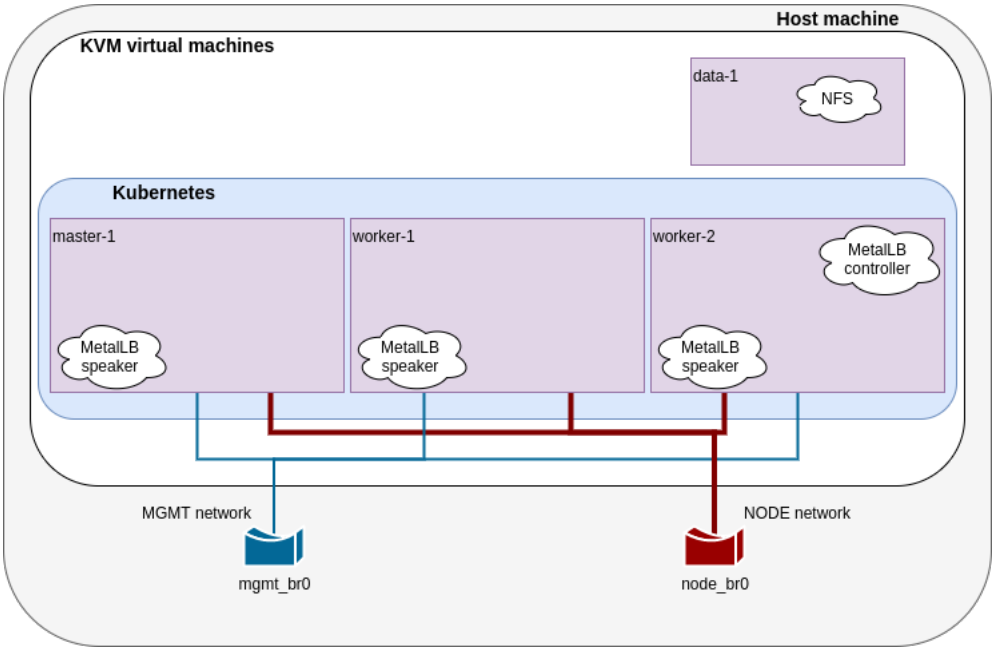

Figure 4.2: Desired system setup with an environment ready Kubernetes.

Environments

Each environment covers a whole system with multiple applications and subsystems. In terms of Kubernetes, the environment is a collection of compatible objects/resources. Some with simple functions and dedicated applications (e.g. FTP server, private Docker registry, Gitea/GitLab, mail server or Nextcloud) others with connected active processes or dependent applications (e.g. data pipelines or microservice ecosystem).

Technically, an environment in terms of this thesis is a collection of YAML files stored in a remote repository easily deployed to the cluster. More specifically, Kubernetes *pods*, *services*, configuration (via e.g. *configmaps*, *secrets*), mounted volumes and/or permission-related objects. Practically, pods are replaced with

deployments and published services are using service type *LoadBalancer* (utilizes MetalLB).

4.3 Container monitoring

Monitoring a container requires to observe container file system, networking and process execution in the most efficient way. Inspired by some related solutions, this section describes the monitoring mechanisms, tools and techniques utilized in this thesis.

- Points of enter, such as honeypots that lure the threat actors to the environment. Could be local to the environment or remote anywhere in the Internet.
- 3 Ubuntu server nodes are the base to Kubernetes cluster holding and orchestrating the whole environment.
- There are to be multiple environments.
- Any environment is automated and deployed to the cluster via Ansible playbooks.
- The core monitoring tools and programs are deployed on the hosting nodes

4.3.1 Activity events

The traced events, which are used to identify the malicious actor's agenda are file system, network and process related. Each type of event has a different repetition period and importance. For example, network and process events are also self-occurring (with no user initialization activity), therefore not all are held as valuable as file system changes. Files are changed less frequently and selectively choosing

valuable directories, the monitoring agents gain visibility over the desired parts of the file system. Nevertheless, the down side to all types of these events are those interpreted as false positive.

According to ISO-27002 category 12.4 part "12.4.4 Clock synchronisation" "the clocks of all relevant information processing systems within an organization or security domain should be synchronised to a single reference time source" [7]. All events are tagged with a timestamp to correctly create a timeline of all events together. Since the designed system is wrapped by VMs, which share the host machine, the time is identical each VM.

File system events

Any modifications or removal of existing and creation of new files in the specified locations of the file system are detected and logged. Additional events possible to monitor, are file permissions, owners, access lists and renaming. Since, "Everything is a file" mostly applies, there are expected to be many uninteresting events.

Process execution events

It depends on the implementation, but process events are any new instances that are readable in the output of "`ps waxu`" command. Including process and parent process IDs (PID, PPID resp.), user, command line including arguments and optionally memory and CPU usage creates an image of observed activities.

Network events

Malicious actors use network to exfiltrate data, send commands or connect to the target service/system. Malicious programs propagate and mutate through

network. For all these scenarios a different protocol or technique may be used. Although, in Kubernetes pod/container network services must be published over Kubernetes service objects, therefore setting up a backdoor listing on arbitrary port requires access to *kubectl* or a vulnerability in Kubernetes itself. Meaning only existing services are of interest in addition to general exfiltration network protocols e.g. FTP, SMTP, HTTP/S, DNS, SMB according to the MITRE technique - "Exfiltration Over Alternative Protocol" [8].

4.3.2 Tools and techniques

Whether it is monitoring from the host machine or directly from Kubernetes, a list of target containers is vital. In case of monitoring from within the containers, an agent/daemon is predefined as part of the pod in its object definition. On the other hand, for a host-based monitoring some kind of reconnaissance must occur to identify all targets in an automated way. The designed tools are described in depth in the following sections.

Host-based monitoring

All events are detectable, to some extend, from the host. Since all activity originates from a Docker container, by design the file system, processes and network traffic is accessible.

Firstly, dynamically getting all the active containers is crucial for inspecting any of the discussed events. Using the control tool *kubectl* get the cluster nodes, which host the pods. Combining *kubectl* and *docker-inspect*, it is simple to retrieve all containers and additional metadata. Altogether the designed tool maps all application containers to nodes for further processing.

Docker container metadata provide all the necessary information to locate the container root directory and mounted file systems. Using a standardized technique, these directories or selective subdirectories may be actively monitored for changes. More specifically, '`.[0].GraphDriver.Data.MergedDir`'[8] gives the top level file system of the *overlayfs* and '`.[0].Mounts[].Source`'[8] returns the mounted directories located on the host. This tool extension executes file system monitoring for each container utilizing tools like *fswatch*, *inotify* or *opensnoop*.

Detecting executed processes is a more sofisticated issue, but using the right technique is important. eBPF enables the tool *execsnoop* to tap into the kernel and record system calls. Fortunately, execsnoop allows various filtering options. Specifically the mount namespace efficiently isolates the observable system calls within a single container. For execsnoop the mount namespace identifier is shared over the BPF map objects using *bpftool*, which are available for a latest kernel version requirement mentioned in section 4.1.

Last of the event types - network traffic monitoring is much more intuitive and native to the architecture and technology used. Since the applications within the cluster are virtualized, the pod network interfaces can be traced to host accessible interfaces. Consequently, a tool taps on to the interface and effectively scans the whole traffic including a pod-to-pod and pod-to-external communications. This leaves open possibilities to tools like *DPDK*, python *scapy*, *libpcap* library or netflow collectors like *suricata*, Although the design expects many pods with many interfaces, so the network traffic collector should have a minimal overhead.

[8] jq filter

Control tool

The main control (*API server*) tool is for managing and controlling all of the above mentioned tools and functionality. It is a centralized API server, which has access to the execsnoop, fswatch and nettool (**TODO change**) for basic management. Additionally, API server provides additional threat hunting functionality such as:

- running process information
- file information **MUST IMPLEMENT**
- **TODO other**

Kubernetes monitoring

The minimal monitoring of the whole cluster is an ideal job for the *Prometheus* toolkit combined with *Grafana* dashboards for visualization. Prometheus provides visibility of the golden metrics[9].

Apart from the network and pod monitoring, other events are not being tracked from within the cluster.

4.4 Deployment and emplacement

This section focuses on the design of deployment methods and ideal emplacement of the system. The focus is mainly on *Ansible*, *Vagrant*, *kubespray* and *kubectl*. The main goal is an Ansible skeleton with playbooks and roles and applicable division by hosts in custom inventory(s).

[9] "Golden metrics (or "golden signals") are the top-line metrics that you need to have an idea of whether your application is up and running as expected."[33]

4.4.1 Deployment

The prerequisite are cluster nodes, which are deployed and pre-configured using Vagrant as discussed in section 4.2.1. While the Vagrantfile defines cluster nodes, Ansible executes scripts, additional configuration and provisioning of Kubernetes and installation of tools' dependencies.

Following the Ansible design patterns, any compact deployment is packed in an Ansible playbook or Ansible role. section 4.2.2 already addresses kubespray as the Kubernetes deployment mechanism, which is not integrated to the main Ansible skeleton.

Apart from the Kubernetes deployment all required setups (both Kubernetes-related and not) and tools are provisioned by playbooks or roles. Each is fully configurable by Ansible group and host variables, which define a specific environment setup and can be cloned to configure an environment with different properties.

4.4.2 Emplacement

An abstract visualization in Figure 4.3 shows the bait system behind a firewall next to a organization's production environment. This one of possible emplacements of the system in a network.

Apart from this emplacement, the designed system can be located anywhere in an internet (e.g. the Internet). Alternatively, a set of low interaction honeypots can used to proxy (or using a reverse proxy) the network to a local instance. This benefits from the flexibility of quickly moving the access points to a different zone, country or network.

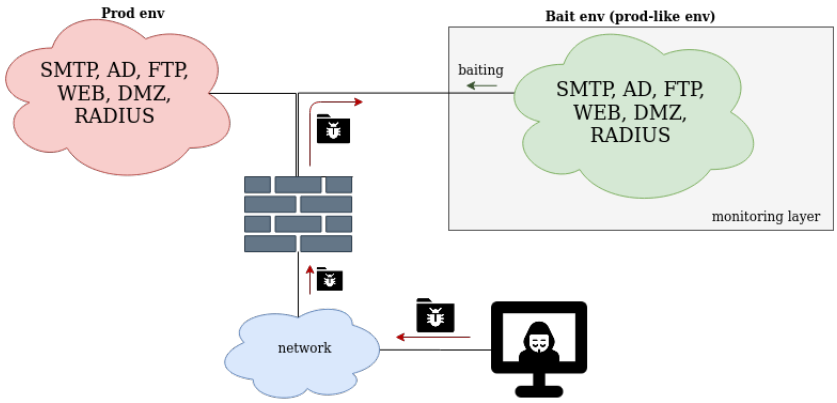

Figure 4.3: Abstract idea of this thesis system emplacement.

Chapter 5

Implementation

In design chapter 4 all required techniques and tools were addressed and in this chapter the solution is documented. The design system is a deception technology monitoring the activity of present users or processes. Each section describes used technology, sub-problem solutions and possible changes in comparison to the designed solution.

5.1 Kubernetes

The cluster is hosted on VMs sharing a common host machine, which meets certain specifications outlined in section 4.1. The base system consists of four VMs (see Figure 4.2), each serving its purpose in the Kubernetes cluster. Precisely, there are three cluster nodes with one additional VM serving a data node. To begin with, almost all configuration and provisioning operations are covered by Ansible roles and playbooks (see Appendix Figure A.1).

5.1.1 Deployment

The Vagrantfile is partially configurable via environment variables specified in extra *environment.rb* (see Listing 5.1) file located in the same directory. Even though, they are environment variables, they inclusion in a separate ruby file is necessary.

```
1  # mandatory
2  K8S_NETWORK_BRIDGE_NAME = "mybr0"
3  K8S_NODES_NET_PREFIX = "172.30.0."  # string prefix representation
       of 172.30.0.0/24
4  K8S_DOMAIN = "k8s-host.local"
5  K8S_SSH_PUBLIC_KEY_PATH = "~/.ssh/id_vagrant_k8s.pub"
6
7  # optional with default value
8  K8S_BOX_DISTRO = "generic/ubuntu1804"
9  K8S_MASTER_NODE_COUNT = 1
10 K8S_WORKER_NODE_COUNT = 2
11 K8S_DATA_NODE_COUNT = 1
```

Listing 5.1: Contents of the *environment.rb*. These all available variables to configure a new deployment in different environment. Applying these variables, Vagrant create four (one master, two worker and one data node) based on *generic/ubuntu1804* box. Furthermore connects each node to *mybr0* network bridge and sequentially assigns IP addresses and hostnames based on the `172.30.0.0/24` network and `k8s-host.local` higher level domain respectively.

The dependencies are the existence of the SSH public key, the network bridge and its IP address corresponding the prefix. The Ansible playbook *vagrant_k8s_up.yml* prepares the dependencies, configures the Vagrantfile environment variables and runs `vagrant up`. At any point the base system can be halted with an opposing playbook *vagrant_k8s_down.yml* At this point the base system is ready for Kubernetes cluster configuration utilizing *kubespray* with specifically changed variables

outlined in Listing 5.2. To reproduce the procedure of installation follow the official guidance[1]. With the defined configuration, kubespray setups Kubernetes with the additional *helm*[2] package manager for more convenient resource deployment.

```
1 helm_enabled: true
```

Listing 5.2: Changed variables in the created inventory. In addition to the the hosts defined in *hosts.yaml* file, kubespray defaults with two master nodes for the setup of three nodes in total - this is changed to one master node.

5.1.2 Environments

A Kubernetes environment in terms of this thesis is a collection of resources within the same namespace (virtual cluster). Before deploying creating any resources there are a couple of playbooks setting up the data node, load balancer, cluster monitoring and the `kubectl` tool.

- `nfs.yml` – Installs the NFS server on the data node and setups the NFS share directory.
- `metallb.yml` – Deploys the MetalLB load balancer related resources to Kubernetes.
- `prometheus.yml` – Setups Prometheus monitoring using the Helm package manager. A couple of arguments configure it and Grafana to retrieve an "public" IP address from the MetalLB address pool, isolate them to a separate namespace and other.
- `kubectl.yml` – Completely configures the `kubectl` tool with bash autocomplete.

[1] `https://github.com/kubespray/kubespray/blob/master/README.md`
[2] `https://helm.sh/`

The environments designed and used are held is a separate git repository[3]. The chosen applications and services are not that important and would require a whole research process, because it's dependent on the user. Even though, concentrating on vulnerable software is a convenient way of baiting an attacker.

Although, as of now, there is only one environment grouping multiple applications and services - Elasticsearch with Kibana, mail server[4], git service Gitea and a custom vulnerable SSH service (vulnerable-ssh). Although, at the time of writing, not all applications are functional. The environment is meant to be versatile to mimic a production system.

For the sake of a proof-of-concept (PoC) vulnerable-ssh is the most important Pod configured with OpenSSH *6.6p1-2ubuntu2.13* and most importantly Bash *4.3.0(1)-release*. As such, the whole environment is considered vulnerable to the shellshock attack exploiting a Bash vulnerability (CVE-2014-6271 [14]) and leading the attacker to run arbitrary code.

5.2 Container monitoring

5.2.1 Reconnaissance

While gathering information of the system dynamically, a set od Bash scripts creates a necessary map of all the container in the environment. A collection tool *sneakpeek*[5] queries the Kubernetes and Docker API, process information and setups the some of the monitoring features.

[3] `https://github.com/tomas321/k8s-environment-scenarios`

[4] `https://github.com/technicalguru/docker-mailserver/tree/master/examples/kubernetes`

[5] `https://github.com/tomas321/sneakpeek`

- `sneakpeek.sh` – Main script calling sequentially all other scripts. It operates in two main options. *fswatch* for setting up the file system monitoring as systemd services based on the "merged directory" of the docker container. *procmon* for setting up the process execution monitoring with container segregation via BPF maps holding the mount namespaces.
- `get_all_containers.sh` – Maps the node IP address to the Docker container and returns it as comma-separated lines.
- `get_container_ns.sh` – It is executed on each node and essentially retrieves the mount namespace of each container. (see Listing 5.3).
- `bpftool_map_container_ns.sh` – It manages the creation of BPF maps holding the mount namespaces and starts the process monitoring *systemd* services.

```
1 pid=$(sudo docker inspect $CONTAINER | jq -r '.[0].State.Pid');
2 echo $(sudo stat -Lc '%i' /proc/$pid/ns/mnt)
```

Listing 5.3: The main piece of script code for retrieving the muont namespace inode number of the given container. This is executed for each container on a corresponding node.

In the first iteration of development the prime script `get_all_containers.sh` was slowing down the whole execution due to iterating through all cluster nodes and executing programs over SSH protocol. Listing 5.2.1 shows that the improved version was 40 times faster.

```
sneakpeek$ time ./get_all_containers.sh 1>/dev/null


real    0m0,404s
user    0m0,163s
sys     0m0,035s
sneakpeek$ time ./get_all_containers.sh.old 1>/dev/null
```

```
real    0m16,896s
user    0m1,695s
sys     0m0,402s
```

5.2.2 Threat hunting

Visibility and control is vital to successfully threat hunt a occurring attack. Additional tools are deployed to monitor the process execution and FS changes. Namely the `execsnoop` syscall spying tool and `fswatch` respectively. Fswatch is a forked[6] Github project and additionally modified to meet the requirements. Execsnoop is a ready to use tool form the BCC dependent on the latest kernel version.

Fswatch

The existing project is functional as is. Although changing the timestamp precision and file event exclusion feature was necessary. Since it's working with *sys/inotify* library, the shortest unit of time is a second. So the precision change is only in the fswatch application layer. It required a thorough inspection of the whole source code to cover each timestamp usage.

Fswatch knows and differentiates FS events. Although, a feature to exclude some events from all the whole selection was missing. The necessity was due to better readability of various usage scenarios when the requested events differ and this speeds up the process.

```
1 fswatch --ex-event=IsSymLink --ex-event=IsDir --ex-event=IsFile --
    ex-event=PlatformSpecific -l 0.5 -r -x -t -f "%m-%d-%Y  %T"
    $DIR
```

[6] https://github.com/tomas321/fswatch

```
2
3 # example output triggered after:
4 # ‘echo "Almost there" > $DIR/tmp/flag‘
5 05-12-2021  16:31:07.816426 $DIR/tmp/flag Created
6 05-12-2021  16:31:07.816494 $DIR/tmp/flag Updated
```

Listing 5.2.2 shows the command arguments used for each instance of FS monitoring process. As mentioned before, here four of 14 known events are ignored using the *–ex-event* option, because they produce too many uninteresting events triggered by common programs e.g. `ls`. To not drain the CPU a latency of maximum 0.5 seconds is chosen and followed by recursive (`-r`) *$DIR* directory monitoring. Output-related flags are for including the event identifier (`-x`), timestamp of occurrence (`-t`) with a specific format (`-f "%m-%d-%Y %T"`).

Execsnoop

Each execsnoop instance is operating in a specified mount namespace shared over a BPF map. This is ensured by the `bpftool` backed by the latest kernel. Listing 5.4 is a `bpftool_map_container_ns.sh` script snippet for creating the BPF map (mapped to a specific file typically in */sys/fs/bpf* directory) if not present already and updates it with a the mount namespace. Additionally, it starts the templated execsnoop systemd services.

```
1 FILE=/sys/fs/bpf/mnt_ns_container1
2 NAME="${FILE##*/}"
3 NS_ID_HEX="$(printf '%016x' $INODE | sed 's/.\{2\}/&\n/g' | tac |
     tr '\n' ' ')"
4
5 # check if eBPF map already exists
6 bpfmap=$(sudo bpftool map show pinned $FILE 2>/dev/null)
7 if [ $? -eq 0 ]; then
8   id=$(echo $bpfmap | cut -d: -f1)
9   sudo bpftool map dump id $id | grep -q -i "$NS_ID_HEX"
```

```
10    if [ $? -eq 0 ]; then
11      # eBPF map already exists
12      sudo systemctl start execsnoop@$NAME; exit 0
13    fi
14  else
15    # creating eBPF map '$FILE'
16    sudo bpftool map create $FILE type hash key 8 value 4 entries
          128 name $NAME flags 0
17  fi
18
19  # updating eBPF map with inode ID '$NS_ID_HEX'
20  sudo bpftool map update pinned $FILE key hex $NS_ID_HEX value hex
      00 00 00 00 any
```

Listing 5.4: Setup of BPF map for sharing the mount namespave with execsnoop instance. INODE variable is retrieved using the command sequence specified in Listing 5.3

Deployment

Both fswatch and execsnoop are deployed and configured using the combination of `sneakpeek.sh` script and Ansible playbooks.

- `fswatch.yml` - Fully configures all nodes with fswatch by runnig the `sneakpeek.sh` script and ultimately starting systemd services for each container.
- `execsnoop.yml` - Similarly, completely configures the nodes with process monitoring running execsnoop as systemd services.

All are logging their according the ISO-27002 category 12.4. The attackers agenda can be identified by inspecting the logged information in the designated location - */var/log/sneakpeek*.

Literature

[1] URL: https://www.stratoscale.com/blog/compute/difference-kvm-and-qemu-virtualization/.

[2] URL: https://wiki.qemu.org/Main_Page.

[3] URL: https://wiki.archlinux.org/title/Network_bridge.

[4] URL: https://www.vagrantup.com/docs.

[5] URL: https://cuckoo.sh/docs/introduction/what.html.

[6] URL: https://github.com/pjlantz/droidbox/blob/master/README.md.

[7] ISO/IEC 27002:2013. *Information technology – Security techniques – Code of practice for information security controls*. Standard. Geneva, CH: International Organization for Standardization, Oct. 2013. URL: https://www.iso.org/standard/62711.html.

[8] Alfredo Abarca, ed. *Exfiltration Over Alternative Protocol*. Version 1.2. The MITRE Corporation. Mar. 28, 2020. URL: https://attack.mitre.org/techniques/T1048/ (visited on 05/29/2021).

[9] Saad Ali. *Kubernetes Design Principles: Understand the Why - Saad Ali, Google*. Dec. 16, 2018. URL: https://www.youtube.com/watch?v=ZuIQurh_kDk (visited on 05/20/2020).

[10] Aditya Anand. "Malware Analysis 101 - Sandboxing. Cons of using a VM". Sept. 29, 2019. URL: https://medium.com/bugbountywriteup/malware-analysis-101-sandboxing-746a06432334 (visited on 03/27/2020).

[11] Dave Anderson. *Concepts*. 2017. URL: https://metallb.universe.tf/concepts/ (visited on 11/10/2020).

[12] Ujaliben Kalpesh Bavishi et al. "Malware Analysis". In: *ijarcsse* (Dec. 2017). URL: https://ijarcsse.com/index.php/ijarcsse/article/view/507 (visited on 03/26/2020).

[13] *Concepts*. Kubernetes. 2020. URL: https://kubernetes.io/docs/concepts/ (visited on 08/24/2020).

[14] *CVE-2014-6271 Detail*. Feb. 1, 2021. URL: https://nvd.nist.gov/vuln/detail/CVE-2014-6271 (visited on 11/15/2020).

[15] *Lecture Notes in Computer Science*. Vol. 3224: *HoneyStat: Local Worm Detection Using Honeypots*. Recent Advances in Intrusion Detection. Springer, Berlin, Heidelberg, 2004, pp. 39–58. ISBN: 978-3-540-30143-1. DOI: https://doi.org/10.1007/978-3-540-30143-1_3. URL: https://people.engr.tamu.edu/guofei/paper/honeystat.pdf (visited on 10/09/2020).

[16] "Defense in Depth: Detonation Technologies". Mar. 12, 2018. URL: https://inquest.net/blog/2018/03/12/defense-in-depth-detonation-technologies (visited on 10/11/2020).

[17] Manuel Egele et al. "A Survey on Automated Dynamic Malware-Analysis Techniques and Tools". In: *ACM Computing Surveys. Article 6* 44.2 (Feb. 8, 2012). Article 6, pp. 5–21. DOI: https://dl.acm.org/doi/10.1145/2089125.2089126. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.640.6356&rep=rep1&type=pdf (visited on 03/26/2020).

[18] "ENDPOINT DECEPTION". URL: https://cybertrap.com/en/solutions/#endpointdeception (visited on 10/05/2020).

[19] Karel Zak Eric Biederman. *nsenter(1) — Linux manual page*. June 1, 2013. URL: https://man7.org/linux/man-pages/man1/nsenter.1.html (visited on 01/08/2020).

[20] Julia Evans. *Ways to spy on a Docker container*. Dec. 29, 2019. URL: https://www.youtube.com/watch?v=YCVSdnYzH34 (visited on 01/08/2020).

[21] Julia Evans. *What even is a container: namespaces and cgroups*. Oct. 10, 2016. URL: https://jvns.ca/blog/2016/10/10/what-even-is-a-container/ (visited on 05/07/2021).

[22] Brendan Gregg. *Linux Extended BPF (eBPF) Tracing Tools*. Mar. 24, 2021. URL: http://www.brendangregg.com/ebpf.html (visited on 05/07/2021).

[23] Tomasz Grudziecki et al. "Proactive Detection of Security Incidents - Honeypots". enisa study. Nov. 2012. URL: https://www.enisa.europa.eu/publications/proactive-detection-of-security-incidents-II-honeypots.

[24] Juan Guarnizo et al. "SIPHON: Towards Scalable High-Interaction Physical Honeypots". In: (Jan. 12, 2017). URL: https://arxiv.org/pdf/1701.02446.pdf (visited on 09/25/2020).

[25] "How it works. Many contributors". Mar. 23, 2020. URL: https://support.virustotal.com/hc/en-us/articles/115002126889-How-it-works (visited on 10/11/2020).

[26] Vladislav Hrčka. "Stantinko's new cryptominer features unique obfuscation techniques". Mar. 19, 2020. URL: https://www.welivesecurity.com/2020/03/19/stantinko-new-cryptominer-unique-obfuscation-techniques/ (visited on 03/23/2020).

[27] Brent Huston. *State Of Security Episode 15.* Apr. 5, 2019. URL: https://www.podbean.com/media/share/pb-cgwhv-ad161e?utm_campaign=w_share_ep%5C&utm_medium=dlink%5C&utm_source=w_share (visited on 10/05/2020).

[28] Brent Huston. "What is this HoneyPoint Thing Anyway?" Dec. 6, 2012. URL: https://stateofsecurity.com/what-is-this-honeypoint-thing-anyway/ (visited on 10/05/2020).

[29] Naresh L J. *Kubernetes Setup Using Ansible and Vagrant.* Mar. 15, 2019. URL: https://kubernetes.io/blog/2019/03/15/kubernetes-setup-using-ansible-and-vagrant/ (visited on 11/02/2019).

[30] "Kubernetes NFS Subdir External Provisioner". Feb. 4, 2021. URL: https://github.com/kubernetes-sigs/nfs-subdir-external-provisioner/blob/master/README.md (visited on 02/06/2021).

[31] Libvirt. *Network XML format.* URL: https://libvirt.org/formatnetwork.html (visited on 05/04/2020).

[32] Libvirt. *Storage pool and volume XML format. Storage pool XML.* URL: https://libvirt.org/formatstorage.html (visited on 05/03/2020).

[33] Risha Mars. *Kubernetes observability with a service mesh. What are golden metrics and why are they important?* Jan. 11, 2021. URL: https://buoyant.io/2021/01/11/kubernetes-monitoring-with-a-service-mesh/ (visited on 04/30/2021).

[34] Yifat Mor. "Using Dynamic Honeypot Cyber Security: What Do I Need to Know?" Oct. 10, 2018. URL: https://www.guardicore.com/2018/10/dynamic-honeypot-cyber-security/ (visited on 10/11/2020).

[35] Alexander Nestorov. "What is Monks". Feb. 24, 2015. URL: https://github.com/alexandernst/monks (visited on 12/04/2020).

[36] *Persistent Volumes*. Kubernetes. 2020. URL: https://kubernetes.io/docs/concepts/storage/persistent-volumes/ (visited on 05/06/2021).

[37] *Pods. Using Pods*. Kubernetes. 2020. URL: https://kubernetes.io/docs/concepts/workloads/pods/ (visited on 08/24/2020).

[38] *ReplicaSet. When to use a ReplicaSet*. Kubernetes. 2020. URL: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/ (visited on 05/06/2021).

[39] *Sandboxing*. Fortinet. URL: https://www.fortinet.com/resources/cyberglossary/what-is-sandboxing (visited on 05/02/2021).

[40] *Service*. Kubernetes. 2020. URL: https://kubernetes.io/docs/concepts/services-networking/service/ (visited on 05/06/2021).

[41] *Special Filtering. Filtering by mount by namespace*. Mar. 22, 2020. URL: https://github.com/iovisor/bcc/blob/master/docs/special_filtering.md (visited on 01/03/2021).

[42] Ivan Studnia et al. "A distributed plateform of high interaction honeypots and experimental results (extended version)". In: (June 10, 2012). DOI: https://hal.archives-ouvertes.fr/hal-00706333. URL: https://www.researchgate.net/publication/262277761_A_distributed_platform_of_high_interaction_honeypots_and_experimental_results (visited on 09/26/2020).

[43] *Use the OverlayFS storage driver*. Docker. May 3, 2021. URL: https://docs.docker.com/storage/storagedriver/overlayfs-driver/ (visited on 05/04/2021).

[44] *Volumes*. Kubernetes. 2020. URL: https://kubernetes.io/docs/concepts/storage/volumes/ (visited on 05/06/2021).

[45] *What is KVM?* Red Hat. Mar. 21, 2018. URL: https://www.redhat.com/en/topics/virtualization/what-is-KVM (visited on 03/29/2021).

[46] Lenny Zeltser. “5 Steps to Building a Malware Analysis Toolkit Using Free Tools”. SANS Institute instructor. Mar. 2015. URL: https://zeltser.com/build-malware-analysis-toolkit/#allocate-virtual-systems.

Appendix A

Ansible

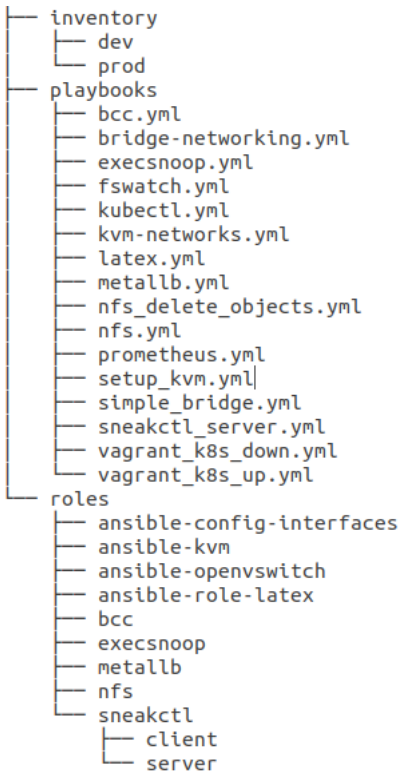

Figure A.1: REQUIRES LATEST!!. Ansible skeleton structure. *ansible-config-interfaces*, *ansible-kvm* and *ansible-openvswitch* are forked or cloned third-party roles.

Appendix B

Vagrantfile

```
def create_node(config, prefix, i, ip)
    short_prefix = prefix[0]
    config.vm.define "#{prefix}-#{i}" do |node|
        node.vm.hostname = "#{short_prefix}#{i}.#{$k8s_domain}"
        node.vm.network :public_network,
            :dev => $network_bridge,
            :mode => "bridge",
            :type => "bridge",
            :ip => ip
        node.vm.post_up_message = "#{prefix} #{i} started"
        node.vm.provision "file", source: $public_key_path,
           destination: "~/.ssh/id_k8s_host.pub"
        node.vm.provision "shell", inline: "cat /home/vagrant/.ssh
           /id_k8s_host.pub >> /home/vagrant/.ssh/authorized_keys"
        # node.vm.provider :libvirt do |libvirt|
        #     libvirt.memory = 2048
        # end
    end
end
```

Listing B.1: A ruby function for node provisioning. Apart for simple network interface configuration, it ensures a given public SSH key is present to bypass the vagrant-specific SSH.

Appendix C

Contents of Included archive

Archive included to the thesis contains following files:

- `/file1` — First file
- `/file2` — Second file