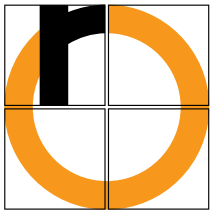
Hochschule **Rosenheim**
University of Applied Sciences



Fakultät für Informatik

Studiengang Informatik B.Sc.

# Attack surfaces and security measures in enterprise-level Platform-as-a-Service solutions

Bachelor Thesis

von

Lukas Grams

Datum der Abgabe: TT.MM.JJJJ TODO

Erstprüfer: Prof. Dr. Reiner Hüttl

Zweitprüfer: Prof. Dr. Gerd Beneken

## Erklärung

Ich versichere, dass ich diese Arbeit selbständig angefertigt, nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel verwendet sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

## Declaration

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references. This paper was not previously presented to another examination board and has not been published.

Rosenheim, den XX.XX.2019 TODO

Lukas Grams

# Abstract (german)

Die wachsende Zahl von Platform-as-a-Service (PaaS) Lösungen, Cloud-Umgebungen und Microservice-Architekturen bieten neue Angriffsszenarien. Dies erhöht den Bedarf an neuen Verteidigungs-Strategien in der IT-Sicherheit von Entwicklungs- und Produktiv-Umgebungen. Gerade Lösungen auf Basis von (Kubernetes-konformer) Container-Orchestrierung sind stark gefragt und haben einen sehr unterschiedlichen Aufbau im Kontrast zu konventionelleren und etablierteren Lösungen. Dieser Umstand legt eine nähere Untersuchung dieses Teilbereichs nahe. Ziel dieser Arbeit ist es, folgende Fragestellungen zu beantworten:

- Was für Sicherheits-Risiken existieren für den Anbieter und/oder Nutzer einer mandantenfähigen PaaS-Lösung, wenn jeder Mandant über eigene Entwicklungs-, Verteilungs- und Laufzeit-Umgebungen für seine Anwendungen verfügt?
- Wie kann ein Anbieter von PaaS-Lösungen an interne und/oder externe Nutzer diese Risiken eindämmen?
- Was spricht in diesem Kontext aus Sicht eines PaaS-Anbieters jeweils für und gegen Vor-Ort- bzw. Cloud-Lösungen?

Ein weiteres Ziel ist es, Maßnahmen für die unterschiedlichen Implementierungsmöglichkeiten zu empfehlen. Der Vergleich bestehender Risiken und daraus abgeleitet der priorisierte Handlungsbedarf sollen hierbei als zentraler Betrachtungswinkel dienen. Unter den etablierten PaaS-Lösungen in unterschiedlichen Umgebungen finden sich OpenShift Container Platform als Vor-Ort-Lösung und Azure Kubernetes Service für Cloud-Umgebungen. Um die genannten Ziele zu erreichen, grenzt diese Arbeit den Problembereich zuerst ein, indem sie sich auf die standardmäßig eingerichteten und zum Betrieb notwendigen Komponenten dieser zwei gängigen und Kubernetes-zertifizierten PaaS-Lösungen konzentriert. Das Hauptaugenmerk liegt hierbei auf den Kubernetes-konformen Teilkomponenten, da hier die Risiken und Maßnahmen den weitreichensten Gültigkeitsbereich haben, unabhängig von der betrachteten Lösung. Aus den Zielen werden drei gängige Angriffsszenarien hergeleitet:

- Angriff von böswilligen Dritten auf die Infrastruktur von innerhalb des LAN und/oder dem Internet

- Angriff von böswilligen Dritten aus einem Container heraus, über den die Kontrolle übernommen wurde. Ein Beispiel wäre das Asführen von Code oder Befehlen per Zugriff von außen.
- Fahrlässiger, übernommener oder böswilliger Nutzer bzw. Nutzer-Identität, was ein Kompromittierungsrisiko für diesen und/oder weitere Mandanten und deren Anwendungen darstellt.

Sie dienen als Grundlage zur Identifikation von Angriffsvektoren, deren jeweiliges Schadenspotential evaluiert und das bestehende Risiko eingeschätzt werden. Es werden potentielle Maßnahmen zur Risiko-Eindämmung gesucht, evaluiert und exemplarisch in der Praxis umgesetzt. Im Anschluss findet eine Neubewertung des Risikos vorgenommen, um so beispielhaft eine Methode zum Risikomanagement zu demonstrieren. Mithilfe der Ergebnisse werden verschiedene Implementierungs-Empfehlungen verglichen, anhand der verschiedenen Anwendungsfälle differenziert und jeweils Maßnahmen empfohlen.

Schlagworte:
Platform-as-a-Service, PaaS, Cloud, Security, Container, Kubernetes, Docker, Risikomanagement, OpenShift Container Platform, OCP, Azure Kubernetes Service, AKS

# Abstract (english)

The increasing amount of Platform-as-a-Service (PaaS) solutions, cloud-hosted environments and microservice architectures introduce new attack scenarios. This creates the need for new defense strategies in both Development and Operations. Especially solutions providing (Kubernetes compliant) container orchestration are identifiably different and in high demand compared to long established solutions. This calls for a more detailed, focused examination. The thesis aims to answer the following questions:

- What generic security risks emerge when providing or using a multi-tenant PaaS solution, with each tenant developing, deploying and running their own applications?
- How can a PaaS provider (serving internal and/or external users) mitigate those risks?
- In this scope and from a PaaS provider viewpoint, how does an on-premise solution compare to a public cloud solution?

Another goal is to recommend security measures for different implementation use cases. A comparison of existing risks and the need for action derived from it should serve as central point of view. Examples for widely used PaaS solutions in different environments include OpenShift Container Platform as an on- premise solution and Azure Kubernetes Service as a public cloud solution. To achieve the aforementioned goals, the thesis will first limit the view on the problem to a manageable scope by concentrating on the components enabled by default and those required for operations of these established and Kubernetes-compliant solutions. Components providing Kubernetes compliance will be the main focus, as these bear the most significance across all Kubernetes Certified solutions. Three common attack scenarios will be derived from the goals:

- Malicious third party attacking the underlying infrastructure from within the LAN and/or the internet
- Malicious third party attacking from inside a hijacked container, i.e. remotely executing code or commands

- Bad User, i.e. a negligent, hijacked or malicious developer (account) risking compromise of his own and/or other applications

These serve as a foundation to identify attack vectors, evaluate their respective potential impact and estimate their risks. Possible measures to mitigate those risks will also be explored, evaluated and exemplary put to use in practical examples. Subsequently the risk will be re-evaluated in order to illustrate a viable risk management method. With the results gathered, the thesis will compare different best practice implementations for different use cases and recommend measures for each.

Keywords:
Platform-as-a-Service, PaaS, Cloud, Security, Container, Kubernetes, Docker, risk management, OpenShift Container Platform, OCP, Azure Kubernetes Service, AKS

# Contents

# List of Figures



1 Wat17.
2 Pro19b.
3 Mei19.
4 Mei19.
5 Mei19.

# List of Tables

# Listings



1 Pro19a.

# List of abbreviations

| | |
|---|---|
| AKS | Azure Kubernetes Service |
| CIS | Center for Internet Security |
| CLI | Command Line Interface |
| CNCF | Cloud Native Computing Foundation |
| CRI | Container Runtime Interface |
| CSVS | Container Security Verification Standard |
| IaaS | Infrastructure-as-a-Service |
| k8s | Kubernetes |
| LAN | Local Area Network |
| LXC | LinuX Containers |
| OCI | Open Container Initiative |
| OCP | OpenShift Container Platform |
| OKD | Origin Community Distribution of Kubernetes |
| OS | operating system |
| OWASP | Open Web Application Security Project |
| PaaS | Platform-as-a-Service |
| RHEL | Red Hat Enterprise Linux |
| VM | virtual machine |

# 1 Introduction

With this chapter, the reader should be able to comprehend why this thesis was written, what it tries to accomplish and which topics are considered within the scope of this work.

## 1.1 Motivation

The increasing amount of Platform-as-a-Service (PaaS) solutions, cloud-hosted environments and microservice architectures introduce new attack scenarios. This creates the need for new defense strategies in both Development and Operations. Especially solutions providing k8s compliant container orchestration are identifiably different and in high demand compared to long established solutions. This calls for a more detailed, focused examination.

## 1.2 Objective

This thesis aims to answer the following questions:

- What generic security risks emerge when providing or using a multi-tenant PaaS solution, with each tenant developing, deploying and running their own applications?
- How can a PaaS provider (serving internal and/or external users) mitigate those risks?
- In this scope and from a PaaS provider viewpoint, how does an on-premise solution compare to a public cloud solution?

Another goal is to recommend security measures for different implementation use cases. A comparison of existing risks and the need for action derived from it should serve as central point of view. Possible measures to mitigate those risks shall be explored, evaluated and exemplary put to use in practical examples. Subsequently the risk will be re-evaluated in order to illustrate a viable risk management method. With the results gathered, the thesis will compare different best practice implementations for different use cases and recommend measures for each.

## 1.3 Scope limitation

The thesis will first limit the view on the problem to a manageable scope by concentrating on the components enabled by default and those required for operations of two established and k8s compliant solutions. These solutions will be OpenShift Container Platform (OCP) as an on-premise solution and Azure Kubernetes Service (AKS) as a public cloud solution. The latest stable version of OCP during the work on this thesis was version 3.11[1], which is based on v1.11 of k8s. Although k8s v1.13 was already available through AKS at the same point in time, the available v1.11 was chosen to improve comparability. Components providing k8s compliance will be the main focus, as these bear the most significance across all Kubernetes Certified solutions. A look at popular tools and frameworks used in such clusters will be avoided in order to keep the scope manageable, though some might be recommended as a mitigation. In order to be applicable to a higher number of use cases, attacks and measures seen in environments with exceptionally high security requirements might be mentioned , but not covered in their entirety. This might entail state-sponsored actors deploying zero-day exploits, which are not applicable to a majority of solutions deployed. This thesis aims to provide insight to the risks of providing a PaaS solution and mitigations thereof. As such, it will look at the capabilities a potential provider has to (mis-)configure such solutions - inherent risks of the technologies themselves are only explored when measures to mitigate them are accessible from a provider standpoint. In short, the goal is to improve the security of your k8s cluster, not k8s itself. To follow the Open Web Application Security Project (OWASP) Risk Rating Methodology[2] down the line, we need to define our threat agents and group them when applicable[3]. Examining a list of threat actors published through SANS[4], we will only exclude state-sponsored actors. This leaves us with cyber criminals, hacktivists, systems administrators, end users, executives and partners. Grouping the remaining actors by the factors used to estimate risks in section 4.2, we can identify find three scenarios that encompass all of the actors:

- Malicious third party attacking the solution from within the Local Area Network (LAN) and/or the internet
- Malicious third party attacking from inside a hijacked container, i.e. remotely executing code or commands
- Bad user, i.e. a negligent, hijacked or malicious account risking compromise of their own and/or other applications

1 Red18a.
2 Fou19a.
3 Fou19b.
4 Irw14.

## 1.4 Research basis and its limits

During the research for this thesis, a considerable amount of sources has been examined. Surprisingly, very little has been found regarding a risk-based point of view on k8s solutions. A lot of sources start with measures and end with attacks, including one of the few published books[1]. They recommend specific security measures an explain the sort or attacks they defend against. The only commonly referenced source that specifically introduces major risks for container technologies, without rating them, is the NIST Application Container Security Guide.[2] In addition to that, many of them are still works in progress, either unfinished or outdated.

Some of the sources commonly referenced include (in alphabetical order):

- The Center for Internet Security (CIS) Docker Benchmark[3]
- The CIS Kubernetes Benchmark[4]
- The book "Kubernetes Security - How to Build and Operate Applications Securely in Kubernetes"[5]
- The NIST Application Container Security Guide[6]
- The OWASP Container Security Verification Standard (CSVS)[7]
- The OWASP Docker Top 10[8]

Other sources include solution specific advisories, i.e. for OCP[9] and AKS[10], as well as recorded and openly available talks by different people in the industry. Especially the talks at KubeCon, the annual k8s conference, are accessible through the Cloud Native Computing Foundation (CNCF) youtube presence[11]. Many speakers include recognized people in the industry, contributors to the k8s project and employees of reputable companies like Google, Red Hat and Microsoft. A considerable number of other online sources were referenced in the in the sources above or found with online search engines. These common recommendations exist, but neither claim to be exhaustive nor applicable to all future versions. In case of the CIS benchmarks, they are not even intended to be followed in full, but just as

1 Liz18.
2 Mur11.
3 Int19a.
4 Int19b.
5 Liz18.
6 Mur11.
7 Red19a.
8 Fou19c.
9 Red19b.
10 Mic18.
11 Channel Link: https://www.youtube.com/channel/UCvqbFHwN-nwalWPjPUKpvTA/videos

basis for security considerations.[1] In accordance to this, we can only claim to provide these findings on a best-effort basis and not without reservation towards possible changes through future software versions or new information. As with all systems, new attacks and vulnerabilities may emerge at any point in time. With sufficient research, the chance to have identified the most common attack vectors is probable but we want to empasize the limitations of this research.

1 McC18.

# 2 Theory

With this chapter, a reader with foundational knowledge of topics regarding Computer Science and/or Informatics will be able to grasp the specialized technologies discussed within the thesis and familiarize themselves with the definitions and terminology used throughout it.

## 2.1 Infrastructure-as-a-Service

In Infrastructure-as-a-Service (IaaS) environments, a consumer trusts his IaaS provider with the management and control of the infrastructure needed to deploy his applications. The provided service ends at provisioning of processing, storage, networks and other computing resources.[2] Therefore consumers do not need to manage their own data centers or topics like system uplink availability. As shown in Figure 2.1, consumers are responsible for the operating system (OS) layer and everything above it.

2 ST11, pages 2 to 3.

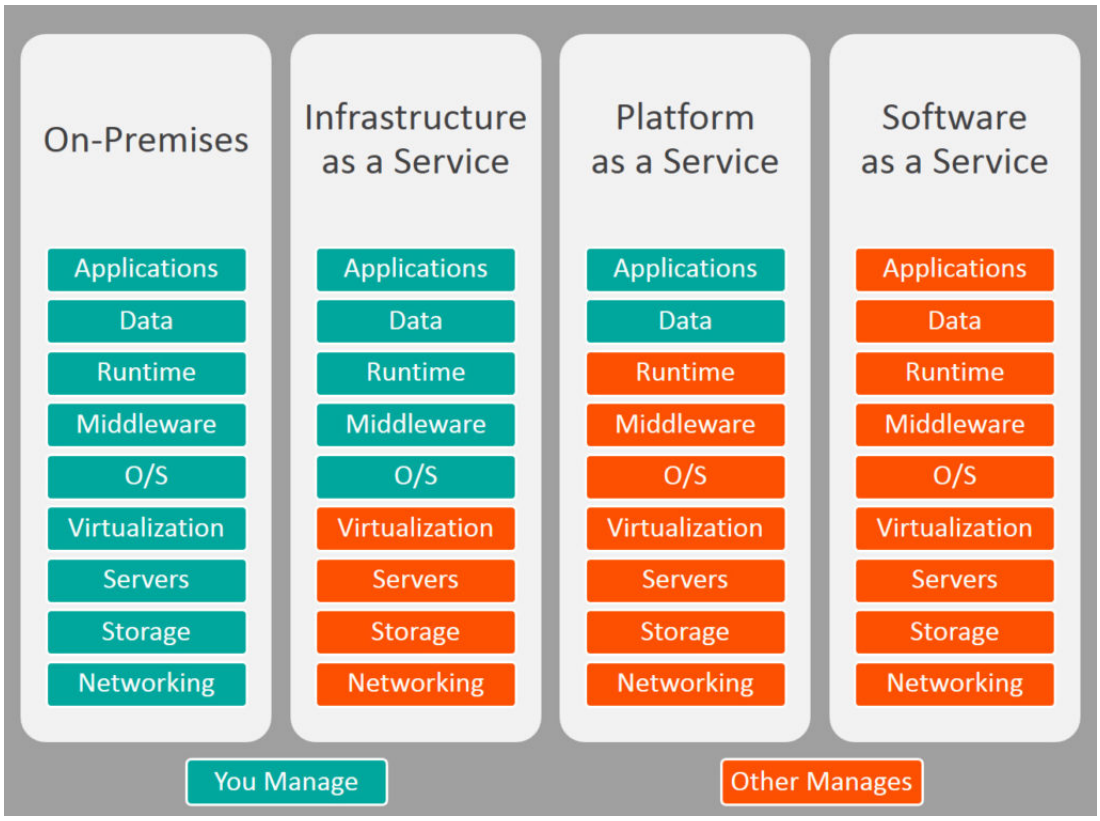



**Figure 2.1** Comparison of responsibilities in different service models[a]

*a* Wat17.

## 2.2 Platform-as-a-Service

In Platform-as-a-Service (PaaS) environments, a consumer trusts his PaaS provider with the management and control of even more resources needed to deploy his applications beyond those covered by IaaS.[1] In an ideal scenario, this leads to a consumer not having to concern himself with the underlying network, hardware, servers, operating systems, storage or even common middleware like log data collection and analysis[2] and allows him to focus on other tasks, i.e. application development. As a downside to this, a consumer might only have limited control on, among other factors, the software installed on the provisioned machines. Although this shifts some of the responsibility burden towards the provider, the situation isn't as clear-cut as one might think. Figure 2.1 shows middleware and runtime as provided, but there is no clear standard on what capabilities or tools are included. A consumer might require capabilities which aren't provided or wishes to avoid provider lock-in through proprietary tools, resulting in some middleware responsibilities falling back to the consumer. A consumer might also

1 ST11.
2 Cor19a.

have to extensively configure or modify the application-hosting environment for compliance or security purposes. Even some low-level tasks aren't completely managed, i.e. virtual machine (VM) reboots to apply security updates.[1]

## 2.3 Containers

Unsurprisingly, a basic building block of running containerized applications are containers. In order to run and manage containers, several components and systems are needed. The most important ones will be introduced here.

### 2.3.1 What are containers?

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another.[2] From a technical viewpoint, a container is an isolated process running in the userspace of a host OS. The host system shares the kernel with all containers on it and might share other resources, but containers are run from container images which include any code and dependencies needed in order to make them independent of the infrastructure they are running on (except the kernel).[3]

Linux containers were widely popularized through Docker, a container system initially based on LinuX Containers (LXC).[4] Containers provide isolation of multiple appllications running on the same machine and are often deployed in environments where this isolation would formerly have been achieved by using multiple VMs. Thus, they are often compared to them despite the fundamental technical differences. The Kubernetes documentation illustrates the differences as seen in Figure 2.2.

1 Cor19b.
2 Inc19a.
3 Raa18.
4 Osn18.

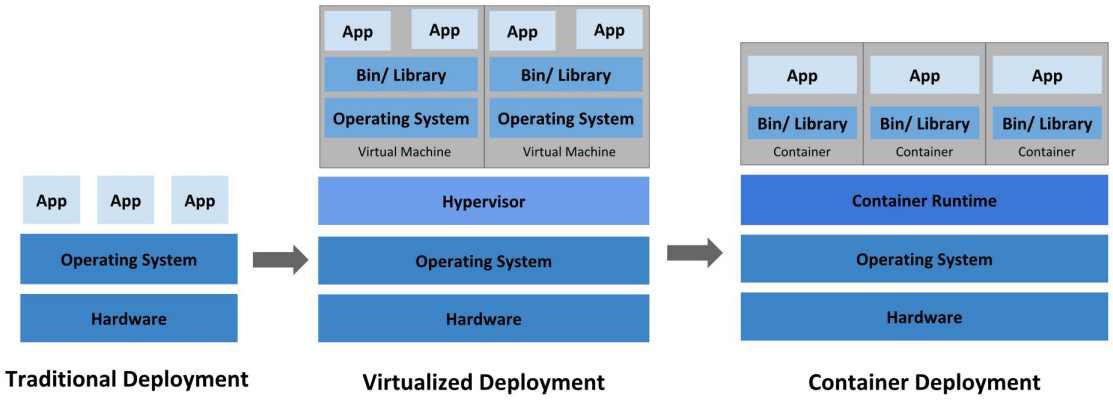



**Figure 2.2** Comparison of different application deployments on the same hardware[a]

*a* Pro19b.

### 2.3.2 Differentiating docker

Talking about docker can be quite difficult, since the term is overloaded with different meanings - a company (Docker Inc.), their commercial products (Docker CE and Docker EE) and the former name of their open source project (formerly known as Docker, now called Moby).[1] Additionally, there is a Command Line Interface (CLI) called docker engine, which serves as an interface to the containers running on a host. It includes a high-level container runtime (which will be talked about in the following section).[2] Some sources also talk about docker-formatted containers when they implement the same interfaces as the docker container runtime.[3]

### 2.3.3 Images, image building and Dockerfiles

A container image is a binary including all the data needed to run a container. It might also contain metadata on its needs and capabilities, i.e. version information through tags.[4] Container images are sometimes referred to as docker images or docker-formatted images, but they can be run by other container runtimes and vice-versa. In order to create a container image, you have to build it. This is often done through a build process executed by a container runtime. The instructions for container builds are commonly defined and documented in a Dockerfile[5] (which may also be done by non-docker programs, adding onto the vocabulary confusion). Container images can be distributed through container image

1 Cha17.
2 Inc19b.
3 Red19c.
4 Red18b.
5 Inc18.

registries, where images can be uploaded to and downloaded from. A commonly known example is docker hub, a public registry run by Docker Inc.

### 2.3.4 Container standards and interfaces

Without going into the nuances and historical developments, there are a multitude of programs mostly implementing three interfaces for container management. The two basic interfaces are the runtime and image specifications under the Open Container Initiative (OCI) which standardize how containers and container images should be formatted and executed.[1] The OCI also maintains a commonly used reference implementation called runc, alternatives include rkt and lmctfy.[2] Runc and similar programs implementing these specifications are commonly called low level runtimes, in contrast to the high level runtimes that control them. These high level runtimes like containerd or CRI-O commonly manage more abstract features like downloading and verifying container images.[3] Many high level runtimes today adhere to the Container Runtime Interface (CRI) so they can be used interchangeably by container orchestrators.[4]

## 2.4 Container orchestration

Once you want to use multiple containers on different machines talking to each other and offering stable services that continue even when a container or machine fails, the need for automated systems to manage these containers arises. Orchestrators can also provide other advantages like load balancing and automated scaling. Kubernetes systems are popular orchestrators currently in use. The sum of hosts running the orchestrator and containers are called a cluster.

### 2.4.1 Kubernetes

At its core, k8s is a control system for containers. It constantly compares the current state with the set target state and tries to correct towards the target when needed, i.e. when a container crashes. The intended way for a user to deploy or change their application is to manipulate the target state and let the k8s system take care of the rest.[5] There are many k8s distributions, many of which are certified kubernetes offerings, meaning they all comply to the same standards and interfaces. These are set by

1 Fou19d.
2 Lew19a.
3 Lew19b.
4 Pro16.
5 Pro19c.

the Linux Foundation through the CNCF which oversees the project.[1] The kubernetes code base is open source and maintained on GitHub, but any implementation fulfilling the (publicised) requirements can become k8s certified, regardless of how much code they changed.[2] You could look at k8s as a standardized interface for container orchestration with a public reference implementation.

### Kubernetes components

In order to deliver a functioning k8s cluster, multiple binary components are needed. Master components provide the control plane, while node components are run on each underlying machine in order to maintain and provide the environment to execute the containers you want to run eventually. Master components are often exclusively run on machines dedicated to them, which are called master nodes - in contrast to worker nodes, which run the containers your applications consist of.[3]

The most relevant master components from the perspective of this thesis are:

- The kube-apiserver, which exposes the Kubernetes API. It is the front-end for the Kubernetes control plane; User commands are typically directed at and processed by this component.

- Etcd, a distributed high-availability key value store where, among other things, secrets and authentication information are stored.

The important node components include:

- The kubelet, an agent running on each node in the cluster. It mostly monitors the state of any containers running because of kubernetes. These are run through pods, a kubernetes object designated to executing containers. The kubelet also interacts with the master components and reports on the monitoring data.

- The container runtime which is responsible for actually running containers. OCP uses Docker while AKS uses Moby, but any implementation of the CRI is supported.[4]

Figure 2.3 illustrates these components in context. It is to note that users in this illustration are the users of the applications running in the cluster; From a provider standpoint the users would be the people responsible for development and operations.

1 Fou19e.
2 Fou19f.
3 Pro19d.
4 For additional information, refer to section 2.3.4

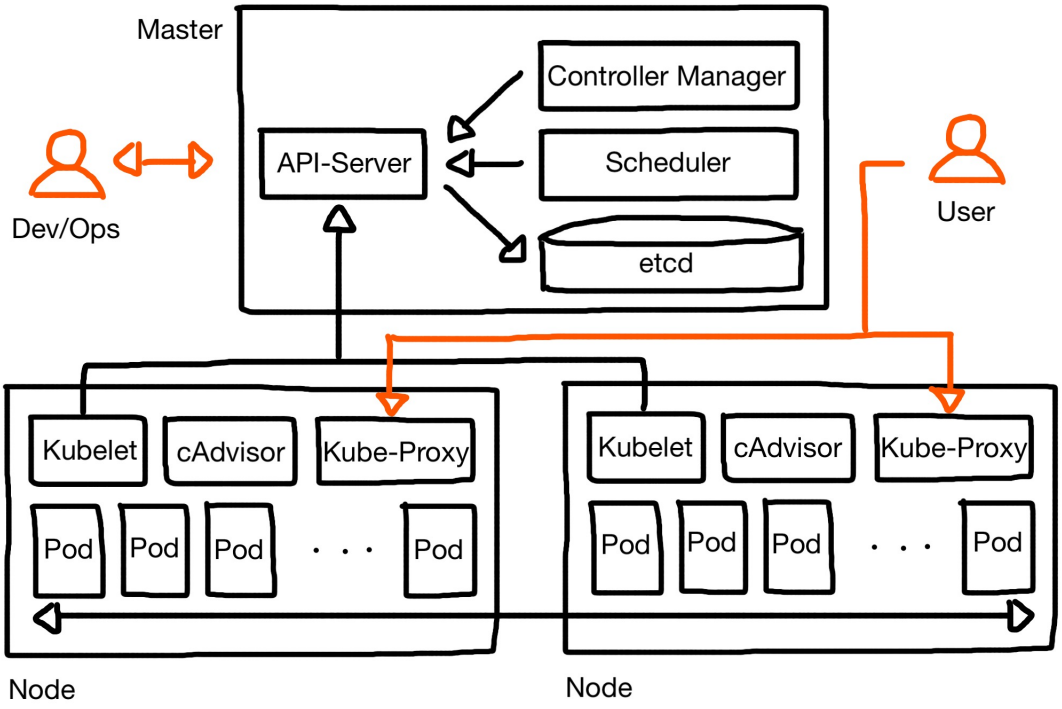



**Figure 2.3** An overview of different k8s components[a]

*a* Mei19.

### Kubernetes objects

Kubernetes objects are persistent entities that represent the desired state of your cluster. They can describe what containers should run, which resources are available to what system and the policies to apply (i.e. automatic restart behaviour and communication restrictions). The intent is to modify these objects in order to change the target state, which the k8s system then works towards by adjusting the current state to match.[1] Some objects, i.e. pods, belong to a specific namespace, meaning a virtual cluster of many in a shared physical cluster. Others, like node objects describing the underlying machine, exist outside of a specific namespace. In order to understand how one can make the k8s system run an application according to its requirements, an understanding of the basic objects is needed.

A pod is the basic k8s object and encapsulates a container with some resources like an IP, storage and policies. Pods are typically each comprised of one container, a single instance of an application in k8s. They may contain more than one container for cases where these are tightly coupled and directly share resources.[2] That's all the pods do. They run.[3]

1 Pro19a.
2 Pro19e.
3 BC18.

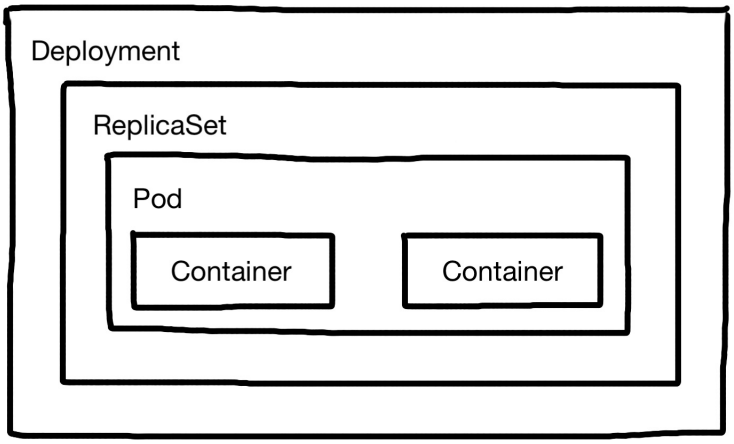



**Figure 2.4** Connection between deployments and other k8s objects down to containers[a]

*a* Mei19.

Pods are usually not created manually, but started and stopped by a ReplicaSet. As a k8s controller, its purpose is to maintain a stable set of replica pods running at any time in order to ensure availability of the function this type of pod provides.[1]

ReplicaSets are in turn managed by deployments. Just as replicaSets control pods, deployments control replicaSets in order to maintain the currently desired state of a cluster. Figure 2.4 illustrates this connection.

Since multiple identical pods may provide the same functionality and instances of this type of pod could be stopped or started at any point, how could a pod or other system address and connect to a pod? This can be solved by configuring a service, which abstracts a set of pods and their access policy. Typically, you talk to services instead of other pods directly. These services might then be published cluster-externally, i.e. through a load balancer provided by a cloud provider as seen in figure 2.5.

1 Pro19f.

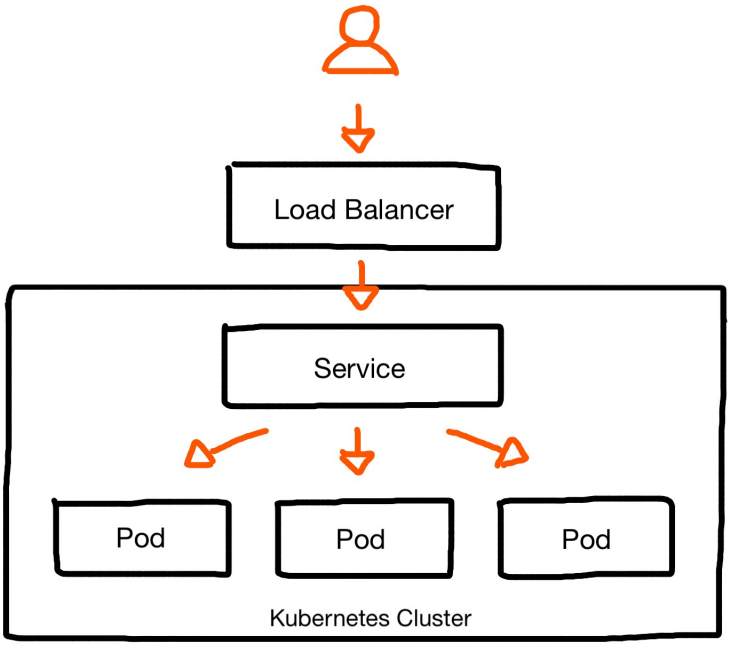



**Figure 2.5** Connection between deployments and other k8s objects down to containers[a]

*a* Mei19.

### Using Kubernetes

There are many ways and solutions to set up a k8s cluster. Anyone can write their own one from scratch, but so-called turnkey solutions for cloud and on-premise environments exist, too, which significantly reduce the time and effort required to set up and run a cluster.[1] Once your cluster is set up, it is typically interacted with through the k8s API. For human interaction, kubectl is a CLI reference implementation which enables remote interaction with it.[2] In order to create or manipulate an object, you need to provide its new specification. This is typcally supplied to kubectl as a .yaml file, an example of which can be seen in listing 2.1.[3]

**Listing 2.1** Exemplary .yaml file of a simple k8s deployment[4]

```
1  apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
2  kind: Deployment
3  metadata:
4    name: nginx-deployment
5  spec:
6    selector:
7      matchLabels:
8        app: nginx
9    replicas: 2 # tells deployment to run 2 pods matching the template
10   template:
11     metadata:
12       labels:
13         app: nginx
14     spec:
15       containers:
16       - name: nginx
17         image: nginx:1.7.9
18         ports:
19         - containerPort: 80
```

1 Pro19g.
2 Pro19h.
3 Pro19a.

### 2.4.2 OpenShift

OpenShift is a commercial Kubernetes container platform by Red Hat. There are several different options to choose, deployable in different environments. OpenShift Online is hosted in a public cloud, while OpenShift Dedicated is hosted on dedicated nodes in a virtual private cloud. The option most relevant to this thesis is OCP, which can be deployed on any infrastructure, including on-premise environments.[1] Red Hat sponsors and supports the development of Fedora, a Linux distribution on which they base their Red Hat Enterprise Linux (RHEL) OS on. RHEL is then used by them commercially by supplying its binaries and updates through commercial licenses. Similar to this, Red Hat develops and supports a k8s distribution called Origin Community Distribution of Kubernetes (OKD), which then serves as a base for their OpenShift products, including OCP.[2] OKD release versions correspond to the k8s version it is based on, i.e. OKD 1.11 is based on k8s 1.11.[3] OCP versions may differ, but Red Hat maintains a list of tested integrations for each major OCP release, giving insight to which k8s version it is based on.[4] OCP is a certified kubernetes distribution, which means it implements the same interfaces as all the others.[5] Despite this, the differences in intended usage become apparent quite quickly, starting with the CLI used to interact with a cluster (oc instead of kubectl).[6] Even basic concepts like namespaces are different, as OCP uses the similar but not identical concept of projects.[7] OCP uses Docker as its container runtime by default, but CRI-O is another option.[8]

### 2.4.3 Azure Kubernetes Service

The Azure Kubernetes Service is a managed k8s solution offered by Microsoft and running on the Azure public cloud. The computing resources used to run the cluster are pre-configured and provisioned VMs in the Azure cloud, which do not need to be configured and run a container runtime based on Moby, a toolkit from which the Docker runtime is built.[9] k8s versions in AKS directly correspond to the k8s version they are based on. As many cloud services do, there are multiple integrations to other cloud services, in case of AKS including an option to integrate your Azure Active Directory to authenticate users.[10]

1 Red19d.
2 Har19.
3 Red19e.
4 Red19f.
5 Fou19e.
6 Red19g.
7 Red19h.
8 Red19i.
9 Cha17.
10 Mic19.

# 3 Deriving the attack surface and security measures

Picking up the three scenarios of section 1.3, we identify the possible attack vectors in this chapter to get an overview of the attack surface that is posed by k8s systems. Vectors and the possible attacks by different threat actors are explained as well as possible security measures introduced.

## 3.1 Defining procedure and approach

The identification of attack vectors within scope turned out to be a big challenge, both in research and preparation for a clear presentation. As new minor versions of k8s release approximately every tree months and the Kubernetes project only maintains release branches for the most recent three minor releases[2], the underlying system evolves continuously. Due to lengthy review processes, accredited literature about the topic are rare and risk being outdated quite quickly. Even less formal sources like blog posts or guidelines published by organizations are both rare, still unfinished at the point of this writing or updated continually, further complicating the creation of a snapshot regarding the current state. Nonetheless, we tried to achieve it on a best-effort basis. These vectors are applicable to all k8s solutions, although the available security measures will vary between different products or implementations. Thus, we will generically mention the applicable measures and mention specific examples for AKS and OCP. In order to give a more comprehensible picture, the scope of these vectors varies. Some areas of interest, i.e. unpatched software and vulnerabilities in the underlying server infrastructure, have a broad scope. Since they are well known security topics applicable almost all systems, we felt comfortable to broeadly encapsulate them for a more clear understanding. If we were to rigorously differentiate between only slightly different vectors, this list would be multiple times as long as it already is.

2 Pro19i.

## 3.2 Identified vectors

The following headlines will each define and explain one of the 17 identified attack vectors of this thesis. For brevity of reference, each has a Vector ID assigned which is formatted as Vxx.

TODO mention these generic measures LOG ALL THE THINGS ELK/EFK-stack or Splunk with log forwarding/aggregation (container AND nodes), k8s audit logging (k8s)

MONITORING AND ALERTING prometheus, grafana (visualization), neuvector, (sysdig) falco, aqua, twistlock

### 3.2.1 V01 - Reconaissance through Kubernetes & platform control plane interfaces

Gather information useful for further attacks through accessible: the Kubernetes dashboard & apiserver as well as potential platform webinterfaces & apiserver(s)

components: kubernetes dashboard, kubernetes apiserver, OCP web console,

TODO

MEASURES SAME AS V02

### 3.2.2 V02 - Read confidentials through Kubernetes & platform control plane interfaces

Gather confidential information through the Kubernetes dashboard & apiserver as well as potential platform webinterfaces & apiserver(s)

TODO

DEACTIVATE OR BLOCK UNUSED INTERFACES administration: uninstall / deactivate configurable interfaces , check RBAC permissions with kubiscan from container: network policies blacklisting APIs/consoles from company network: firewall (bad practice)

LIMIT INTERFACE CAPABILITIES deactivate capabilities to the highest authority level whenever possible/applicable

SECURE ACCESS no unauth, no ABAC, integrate existing RBAC you are managing anyway use 2FA when possible (i.e. for dev and/or admin access to cloud) <- openstack: 2FA may be integratable to AD? TODO

FOLLOW LEAST PRIVILEGE PRINCIPLE limit access to each user in their scope (possibly further - need to know principle?)

SECRETS MANAGEMENT DO NOT load secrets from dockerfile / podconfig / env vars or other shit! DO use kubernetes secrets (or better: external key mgmt like hashicorp vault or azure ad pod identity / openshift vault) TODO: how to enforce?

ENABLE ALERTING on events like adding admin / granting admin privs / creating namespace / touching things deemed critical. TODO: how?

### 3.2.3 V03 - Change configuration through Kubernetes & platform control plane interfaces

Change the existing configuration through the Kubernetes dashboard & apiserver as well as potential platform webinterfaces & apiserver(s) TODO

V06 LIMIT CAPABILITIES

EVERYTHING FROM V02

### 3.2.4 V04 - Compromise internal k8s control plane components (etcd, scheduler, controller-manager)

This vector comprises reconaissance, leaks of confidentials and configuration changes through Kubernetes components not intended to be accessible: etcd stores, kube-scheduler and kube-controller-manager

TODO

CLUSTER ISOLATION

FOLLOW ADDITIONAL TIPS BY SUPPLIER

### 3.2.5 V05 - Supply compromised container (base) image

Supplying a malicious container image leading to security violations on the cluster (remote access for an attacker, resource misuse, data leakage, ...). Most easily done untargeted (dockerhub images or dockerfiles on tutorials/help forums), but can be done targeted, too. Additionally, an image build process typically runs as root, leading to compromise possibilities to compromise the node where an image is built from a rogue dockerfile. (TODO: provoking stale image usage to exploit vulns, too?)

TODO

POLICIES FOR CONTAINER IMAGES Define what makes an image safe (enough) for usage. (light: only allow verified dockerhub images. Heavy: run images through a vetting process or build them from scratch) Define in what intervals both policy and set of base images are reviewed

RESTRICT THE SET OF POTENTIALLY USED IMAGES Use a dedicated private image registry which only gets vetted images OR enforce image whitelisting from public sources (including restricting versions, only allow the ones you vetted!)

VET IMAGES BEFORE USAGE Run your defined image vetting process for every container before it becomes available for use Re-vet new versions before they are admitted for usage scan with fossa, clair, microscanner, OPA policies

RE-ASSESS ALL VERSIONS OF ALL IMAGES REGULARLY regularly scan all available images in all their versions and disable images that are (or have become) vulnerable. Especially old image versions should be purged, since those can be vulnerable to well-documented exploits.

### 3.2.6 V06 - Supply compromised k8s configuration

Supplying a malicious kubernetes configuration leading to security violations on the cluster (remote access for an attacker, resource misuse, data leakage, ...). Most easily done untargeted (tutorials/ help forums), but can be done targeted, too.

TODO

RAISE OPERATOR AWARENESS Train operators (cluster administrators & users) regularly Make sure that operators are aware of the risks of copy-pasting

OTHER ORGANIZATIONAL MEASURES write guidelines with good security policies (what to use by default, what exceptions could be allowed and what processes have to be followed before admitted)

LIMIT CAPABILITIES K8s PodSecurityPolicies Enable and configure global ( & potentially additional namespace-identified) Pod Security Policies, which define default values and enforce specific settings if defined. ATTENTION: currently in preview for AKS! (openshift SCCs) (recommentations: TODO! RunAsUser=MustRunAsNonRoot , RunAsGroup=MustRunAsNonRoot, AllowPrivilegeEscalation=false (<- careful, might break setuid binaries! Not-as-good-alternative: DefaultAllowPrivilegeEscalation=false), privileged=false)

restrict communication no traffic to oder projects (OCP) or namespaces (AKS) restrict images use private repo / dockerhub-official images only All of these: either implemented by dev or enforced by admin. Suggestion: enforced by cluster admin (scc / networkpolicy in podsecuritypolicy) with exceptions granted when needed restrict host impact no priviledged containers, no priviledge escalation (asking to become priviledged), restrict kernel capabilities, no root containers (maybe yes, but with user namespace remapping only?), procmounttype default (apply restrictions by container runtimes in /proc), restrict mount volume types and host volume paths, (maybe use sandboxing through gvisor / kata containers?) verify with tool / 3rd party put through kubesec.io offline/ kube-bench/ kubeaudit / kubeatf (enforce: k8guard) or 4-eyes-principle or security clearance on config changes (probably too overkill)

### 3.2.7 V07 - Compromise other application components (lateral movement)

Once an attacker gains access to a container, he may try to access more lucrative application components or information, i.e. sniffing traffic or accessing databases or containers with more confidential information/traffic.

TODO

ENCRYPR CROSS-CONTAINER TRAFFIC istio with mTLS for traffic encryption and signing OR guideline for devs to use TLS everywhere OR enforce with OPA?

AUTHORIZATION/AUTHENTICATION FOR ACCESS, EVEN WITHIN namespace isolation, network restriction, ...appSec problem?

### 3.2.8 V08 - Container breakout (R/W, Privilege Escalation)

Once inside a container, an attacker may try to gain access to the underlying host by a multitude of means. This includes invoking syscalls, accessing the host file system and elevation priviledges within or outside of the container environment

TODO

V06 LIMIT CAPABILITIES -> RESTRICT HOST IMPACT

### 3.2.9 V09 - Compromise local image cache

If the cached image of a container can be manipulated, another container (which might even seem to fulfill the same function) violating security principles could be started.

TODO

ENSURE IMAGE PROVENANCE docker content trust, code signing

### 3.2.10 V10 - Modify running container

Once inside a container, an attacker may try to modify the container to exfiltrate data or better suit their needs for further intrusion

TODO

LIMIT CAPABILITIES like V13, but during runtime

### 3.2.11 V11 - Hoard resources (sabotage)

The provided resources may be misconfigured or misused to disrupt service availability (i.e. through fork bombing or misconfiguration)

TODO

ISOLATION OF COMPONENTS different namespaces (with different resource limits) for different parts of app, depending on criticality

LIMIT RESOURCE USAGE per-app / per-namespace resource limits on CPU, Storage, RAM, Bandwidth, cloud budget, what else?

LIMIT ACCOUNT CAPABILITIES least privilege for dev accounts to limit misconfiguration (i.e. to his own namespace) <- no creation of new namespaces except for within his own resource limits!

LOGGING OR MONITORING CONFIG CHANGES see generic -> logging/monitoring

### 3.2.12 V12 - Misuse resources (cryptojacking)

The provided resources may be misconfigured or misused for financial gains (mining cryptocurrencies)

TODO

MEASURES SAME AS V11 EXCEPT COMPONENT ISOLATION

### 3.2.13 V13 - Add malicious container

A malicious container may be started within the cluster TODO

LIMIT ADMISSION AND CAPABILITIES k8s: V06 -> limit capabilities container runtime: V16->additional restrictions + V17->minimal node setup

### 3.2.14 V14 - Add malicious node

A malicious node may be added to the cluster TODO

REQUIRE ADMIN TO AUTHORIZE NODE ADD Source: NIST-SP800-190, chapter 4.3.5 TODO: how to in ocp / AKS?

### 3.2.15 V15 - Bad user practice (outside of cluster)

This vector comprises user practices outside of the cluster that lead to risks within it. Examples include phishing, openly publishing keys/tokens to public code repositories and more.

TODO

ORGANIZATIONAL MEASURES inform users & admins of risks regarding... ... phishing (classic awareness) ... openly publishing stuff (open source (repos), presentations, talking to external people, troubleshooting threads on stackoverflow, potential partners/vendors, ...) TODO: are there infosec modules/courses/resources for this?

### 3.2.16 V16 - Incufficient base infrastructure hardening

The underlying nodes could allow an attacker easy entry, even if the containers themselves are hardened. This includes Side-Channel attacks like Spectre & Meltdown

TODO

"CONVENTIONAL" HARDENING established resources like CIS benchmarks

V17 NODE PATCH MGMT MEASURES

V17 NODE MINIMAL OS MEASURES

ADDITIONAL RESTRICTIONS AVAILABLE disable SSH access to hosts whenever possible (access through physical when bare-metal / hypervisor interfaces if VM nodes). If needed and risk accepted, enable ssh access through bastion host close exposed ports configured, but not needed (dangling container runtime daemon?) CSVS chapter: infrastructure CIS Docker Benchmark chapters: Chapters 1-3 & 5 (+ Docker Content trust) CIS Kubernetes Benchmark chapters: Worker Node Security Configuration & Configuration Files NIST-SP800-190 chapters: 3.5, 4.5, 4.6 cloud config: check with ScoutSuite and analyze/enforce with cloud-custodian, Azure: check with azurite?

### 3.2.17 V17 - Entry through known, unpatched vulnerabilities

Every system has to be kept up to date with security patches. Publicly known vulnerabilities might otherwise be exploited, leading to potentially devastating violations of security principles

TODO

PATCH MANAGEMENT MEASURES (Components: Hosts, Kubernetes cluster, containers, other cluster tools (Jenkins, . . . ) Define responsibilities for Patch Management Guidelines and their execution (while accounting for vacation time and sick days) Define Patch Management Guideline(s) covering all components, I.e. when to patch, when to upgrade, when to take offline Executing people: subscribe to vulnerability notification feeds or set up automated patching AKS-specific: Security Updates are not rolled out automatically if that would result in downtime! Someone has to restart nodes manually or set up automated process! OpenShift-specific: execute guidelines Regularly assess all components for conformity with the desired state Host: Conventional patch management measures Kubernetes: execute guidelines Container: automated scanning (i.e. with clair) or regular manual assessment, subscribe to alerts (new vulns im container images / code OR other components!)

MINIMAL SETUP Container-optimised node OS (i.e. CoreOS, Google Container-optimized OS) Use mimimal base images (slim is good, alpine is better) for your containers Install only tools required for the operation each specific container on the images used Avoid and eliminate tool functionality overlap (i.e. you don't need two different tools to gather the same logs, each might be at risk through vulns)

# 4 Assessing the attack surface risk

With the attack vectors identified, we introduce a customized risk estimation model for our purpose and explain the challenges of developing it. The model is then used to estimate the relative risk of each vector.

## 4.1 Defining procedures and approach

In order to achieve a view on the risks more accurately resembling situations where a solution might be implemented, several assumptions are made:

- People in contact with the solution are familiar with conventional security principles and measures, but are new to the technologies used in the solutions within scope. They might even be new to container and cloud solutions in general. This includes both users of the solution like developers and project managers, as well as operators and administrators.
- It is assumed that no special requirements like industry-specific compliance requirements have to be followed and the workloads processed within the solution have no exceptionally high security requirements.
- Regarding the design and implementation of applications running on the solutions, it is assumed that conventional application security measures have been implemented, i.e. against the OWASP Top 10.[2] The extent of those measures is assumed to be in accordance to moderate criticality of the data and service provided by the application.
- Some risks increase or decrease drastically, depending on many specific configurations. Considering the high system complexity, "getting it to work" is hard enough for users and operators new to these technologies.[3] When setting up and configuring a setup, the default configurations are left as-is whenever possible. Guidelines of specific implementations are followed, but whenever measures are presented as optional and not required, they will be skipped. Before recommending

2 Fou17.
3 Gee17.

security measures, the setup will be modified just enough to become functional, without regards to the security implementations.

- Multiple tenants like different customers, teams or projects are separated by k8s namespaces or OCP projects respectively, not by clusters.

Focus on three scenarios: attack through network, hijacked container, bad user

Research-Freeze: May 3rd, 2019! (Pre-KubeCon19, check git commit dates for stuff like OWASP-documents etc!) Some newer information might be used for big outliers, but everything else just gets a side note

RISK ASSESSMENT METHODOLOGY: difficulties with: A how generically should vectors be set? B how to structure vectors (into categories?)

Solution to A: vectors split and merged after risk assessment sketch; if there were considerable differences in the estimated values, they were split. If no diffs, they were merged. Solution to B: Considerable time invested, no optimal solution was found. Ultimately ignored this, since more time investment wasnt feasible or added much value to the goals. TODO: Explain process, what was looked at and why it wasnt good, how we arrived at the end structure.

Risk assessment formula: Risk = Probability * Impact. Impact was taken as single value of 1 through 3 and estimated through None/Theoretical (0), Low/Intermediate-Step (1), non-severe security principle violation (2), severe security principle violation (3)

Probability was a bitch to define proberly. Therefore split into four factors: Vantage Point, Required Access Level (RAL), Detectability and Exploitability. Those initially had values between 0 though 3, but outlier values of 4 were defined for RAL and Vantage Point (in sync with existing assessment methodologies within HvS). The average of these four values is taken as the total propability value, ranging from 0.25 through 3.5 (low <= 1.25, medium <= 2.25). Vantage Point: physical access (0, see above since your own or the cloud providers hardware-accessing employees can also do whatever they want); node or management-interface (1); within container (2); within company network (3); from public www (4) Required Access Level: cloud/infrastructure admin (0, since a rogue employee with super-admin can do whatever they want and this is about baseline security); cluster/system-admin (1); cluster/system user with read/write access (2); cluster/system user with read-only access (3); unauthenticated (4) Detectability: Difficult (1) since it needs custom tools for environment-specific vuln detction; Average (2) since it is either generic but needs simple custom tools or its individualized but can be identified with some slight tool individualization; Easy (3) since there are generic script-kiddie tools / GUI-paths to find the vuln Exploitability: Theoretical (0, since this is the level of unpublished 0-days and we are still doing baseline security); difficult (1) needs custom tools for environment-specific

exploitation; Average(2), since its a generic exploit but needs simple custom tools or its inividualized but can be exploited with some slight tool individualization; Easy (3) since there are generic script-kiddie tools / GUI-paths to exploit the vuln

This leads to a total risk of 0 through 10.5, which is then rounded to full integers and capped at 10. In accordance to HvS internal models, total risk values <= 3 are defined as low, <= 6 as medium and values above that are defined as high.

specific values for each vector are estimated in a context with multiple assumptions: TODO: callback to assumptions in scope limitation

- If multiple techniques can be used / impacts can occur to leverage a vector, all factor values of the one with the highest total risk are taken - Values might decrease through the implementation of security measures, leading to a lower total value. (If multiple techniques could be used to leverage a vector and only one gets its total risk reduced, the new maximum risk value of that vector becomes the vector value(s)

goal is to reduce values above threshold X to below threshold X by applying security measures. This aims to ensure a basic security level, not something against APT groups / zero-day protection / targeted attacks with a lot of resources and competence. (no online banking, user data of average confidentiality etc)

Default values are defined as the following:

SETUP OF PRACTICAL PART:

Version freeze: OCP 3.11 -> OKD 3.11 -> k8s-version = 1.11(.0 with fixes, is a fork. see: https://github.com/openshift/orig ) AKS (on May 3rd 2019) => k8s-version <= v1.13.5 available, but only for k8s-v1.11 only 1.11.8 or 1.11.9!

Considerable changes between 1.11.0 and 1.11.9 (Source: https://github.com/kubernetes/kubernetes/blob/master/CHANG 1.11.md): - action required: the API server and client-go libraries have been fixed to support additional non-alpha-numeric characters in UserInfo "extra" data keys. Both should be updated in order to properly support extra data containing "/" characters or other characters disallowed in HTTP headers. (#65799, @dekkagaijin) - https://github.com/kubernetes/autoscaler/releases/tag/cluster-autoscaler-1.3.1 - ACTION REQUIRED: Removes defaulting of CSI file system type to ext4. All the production drivers listed under https://kubernetes-csi.github.io/docs/Drivers.html were inspected and should not be impacted after this change. If you are using a driver not in that list, please test the drivers on an updated test cluster first. “‘ (#65499, @krunaljain) - kube-apiserver: the Priority admission plugin is now enabled by default when using –enable-admission-plugins. If using –admission-control to fully specify the set of admission plugins, the Priority admission plugin should be added if using the PodPriority feature, which

is enabled by default in 1.11. (#65739, @liggitt) The system-node-critical and system-cluster-critical priority classes are now limited to the kube-system namespace by the PodPriority admission plugin. (#65593, @bsalamat)

no major changes => OCP 3.11, AKS-k8s 1.11.9!

Dependency compatibilities: OCP 3.11: docker 1.13, CRI-O 1.11 (Source: LINK COMMENTED k8s-1.11: docker 1.11.2 to 1.13.1 (Source: LINK COMMENTED Azure-AKS: uses moby, NOT docker! (moby = pluggable container runtime based on docker, automatically updated in background whenever no node restart needed)

-> take defaults for all dependencies on install, document and apply all AKS node updates needing manual restart! TODO: apply OCP updates when incoming?

## 4.2 Estimating the risk

The resulting risk estimation is illustrated in Table 4.1. The individual values these results are derived from can be seen in tables 4.2 and 4.3. The details on how these values were determined are explained hereafter.

**Table 4.1** A comparison of the resulting risk for the OCP and AKS environment

| Vector ID | Vector | OCP Risk | AKS Risk |
|---|---|---|---|
| V01 | Reconaissance through Kubernetes & platform control plane interfaces | **3** | **3** |
| V02 | Read confidentials through Kubernetes & platform control plane interfaces | **6** | **7** |
| V03 | Change configuration through Kubernetes & platform control plane interfaces | **7** | **8** |
| V04 | Compromise internal k8s control plane components (etcd, scheduler, controller-manager) | **5** | **6** |
| V05 | Supply compromised container (base) image | **7** | **7** |
| V06 | Supply compromised k8s configuration | **10** | **10** |
| V07 | Compromise other application components (lateral movement) | **7** | **7** |
| V08 | Container breakout (R/W, Privilege Escalation) | **5** | **7** |
| V09 | Compromise local image cache | **3** | **3** |
| V10 | Modify running container | **5** | **5** |
| V11 | Hoard resources (sabotage) | **8** | **5** |
| V12 | Misuse resources (cryptojacking) | **5** | **8** |
| V13 | Add malicious container | **5** | **6** |
| V14 | Add malicious node | **6** | **6** |
| V15 | Bad user practice (outside of cluster) | **6** | **6** |
| V16 | Incufficient base infrastructure hardening | **9** | **9** |
| V17 | Entry through known, unpatched vulnerabilities | **10** | **10** |

**Table 4.2** The risk estimation of all vectors for an OCP 3.11 cluster

| Vector ID | Vector | Vantage Point | RAL | Detectability | Exploitability | Probability | Impact | Resulting risk |
|---|---|---|---|---|---|---|---|---|
| V01 | Reconaissance through Kubernetes & platform control plane interfaces | 3 | 3 | 3 | 2 | 2.75 | 1 | **3** |
| V02 | Read confidentials through Kubernetes & platform control plane interfaces | 3 | 3 | 3 | 3 | 3 | 2 | **6** |
| V03 | Change configuration through Kubernetes & platform control plane interfaces | 3 | 1 | 3 | 2 | 2.25 | 3 | **7** |
| V04 | Compromise internal k8s control plane components (etcd, scheduler, controller-manager) | 3 | 1 | 3 | 0 | 1.75 | 3 | **5** |
| V05 | Supply compromised container (base) image | 4 | 4 | 3 | 2 | 3.25 | 2 | **7** |
| V06 | Supply compromised k8s configuration | 4 | 4 | 3 | 2 | 3.25 | 3 | **10** |
| V07 | Compromise other application components (lateral movement) | 2 | 2 | 3 | 2 | 2.25 | 3 | **7** |
| V08 | Container breakout (R/W, Privilege Escalation) | 2 | 2 | 3 | 0 | 1.75 | 3 | **5** |
| V09 | Compromise local image cache | 1 | 1 | 2 | 2 | 1.5 | 2 | **3** |
| V10 | Modify running container | 2 | 2 | 3 | 2 | 2.25 | 2 | **5** |
| V11 | Hoard resources (sabotage) | 3 | 2 | 3 | 2 | 2.5 | 3 | **8** |
| V12 | Misuse resources (cryptojacking) | 3 | 2 | 3 | 2 | 2.5 | 2 | **5** |
| V13 | Add malicious container | 3 | 2 | 3 | 2 | 2.5 | 2 | **5** |
| V14 | Add malicious node | 3 | 1 | 2 | 2 | 2 | 3 | **6** |
| V15 | Bad user practice (outside of cluster) | 3 | 4 | 2 | 2 | 2.75 | 2 | **6** |
| V16 | Incufficient base infrastructure hardening | 4 | 4 | 2 | 2 | 3 | 3 | **9** |
| V17 | Entry through known, unpatched vulnerabilities | 4 | 4 | 3 | 2 | 3.25 | 3 | **10** |

**Table 4.3** The risk estimation of all vectors for an OCP 3.11 cluster

| Vector ID | Vector | Vantage Point | RAL | Detectability | Exploitability | Probability | Impact | Resulting risk |
|---|---|---|---|---|---|---|---|---|
| V01 | Reconaissance through Kubernetes & platform control plane interfaces | 4 | 3 | 3 | 2 | 3 | 1 | **3** |
| V02 | Read confidentials through Kubernetes & platform control plane interfaces | 4 | 3 | 3 | 3 | 3.25 | 2 | **7** |
| V03 | Change configuration through Kubernetes & platform control plane interfaces | 4 | 1 | 3 | 2 | 2.5 | 3 | **8** |
| V04 | Compromise internal k8s control plane components (etcd, scheduler, controller-manager) | 4 | 1 | 3 | 0 | 2 | 3 | **6** |
| V05 | Supply compromised container (base) image | 4 | 4 | 3 | 2 | 3.25 | 2 | **7** |
| V06 | Supply compromised k8s configuration | 4 | 4 | 3 | 2 | 3.25 | 3 | **10** |
| V07 | Compromise other application components (lateral movement) | 2 | 2 | 3 | 2 | 2.25 | 3 | **7** |
| V08 | Container breakout (R/W, Privilege Escalation) | 2 | 2 | 3 | 2 | 2.25 | 3 | **7** |
| V09 | Compromise local image cache | 1 | 1 | 2 | 2 | 1.5 | 2 | **3** |
| V10 | Modify running container | 2 | 2 | 3 | 2 | 2.25 | 2 | **5** |
| V11 | Hoard resources (sabotage) | 3 | 2 | 3 | 2 | 2.5 | 2 | **5** |
| V12 | Misuse resources (cryptojacking) | 3 | 2 | 3 | 2 | 2.5 | 3 | **8** |
| V13 | Add malicious container | 4 | 2 | 3 | 2 | 2.75 | 2 | **6** |
| V14 | Add malicious node | 4 | 1 | 1 | 2 | 2 | 3 | **6** |
| V15 | Bad user practice (outside of cluster) | 4 | 4 | 2 | 2 | 3 | 2 | **6** |
| V16 | Incufficient base infrastructure hardening | 4 | 4 | 2 | 2 | 3 | 3 | **9** |
| V17 | Entry through known, unpatched vulnerabilities | 4 | 4 | 3 | 2 | 3.25 | 3 | **10** |

### 4.2.1 V01 - Reconaissance through Kubernetes & platform control plane interfaces

A user with access to the apiserver / webinterface(s) and read access can scout out information. By default, each account (project admin or project user, but not cluster admin) can only see information about his own project, a cluster admin can see all namespaces. This could show outdated software versions, running systems / containers / pods / user account privileges / misconfigurations and may support in planning and confirming effectiveness of further attacks.

The information gathering processes and interfaces are known and documented pretty well, but the information gathered has to be analyzed specific to the environment. Same as OCP, except accessible from anywhere (cloud, duh). -> doesn't change total risk value

### 4.2.2 V02 - Read confidentials through Kubernetes & platform control plane interfaces

In addition to V01, a user with access to the apiserver / webinterface(s) and read access can gather confidential secrets like certs, tokens or passwords which are intended to be used by automated systems and/or users to authenticate themselves to cluster components and gain privileged access like pull/push images, trigger actions in other applications / containers, ... These can be gathered and used for further access by an attacker.

Kube-hunter is a readily available tool and checks for this automatically.

Same as OCP, except accessible from anywhere (cloud, duh). -> increases total risk value slightly, pushing it just over the edge from medium to high

### 4.2.3 V03 - Change configuration through Kubernetes & platform control plane interfaces

In adition to both V01 and V02, a user with access to the apiserver / webinterface(s) and write access can change configurations on the cluster. By default, each account (project admin or project user, but not cluster admin) can only change the configuration of namespaces resources (i.e. access to project-specific resources like pods, services, routes, but not cluster-global resources like nodes, SCCs or interface/authorization configurations).

The capabilities can be looked up through the API, what you can achieve with it has to be analyzed environment-specifically though.

Same as OCP, except accessible from anywhere (cloud, duh). -> increases total risk value slightly, both are still rated high in the end

### 4.2.4 V04 - Compromise internal k8s control plane components (etcd, scheduler, controller-manager)

Misconfiguration of internal Kubernetes components (accessible by systems it is not assigned to be accessible by) could lead to a full cluster compromise. The cluster configuration and all secrets / authorization credentials are stored in the etcd instance(s). One would need to seriously fuck up the setup, since OCP configures everything through ansible and you would have to knowingly change some internal settings not intended to be changed in order to achieve this. Configurations are maintained by red hat, meaning config changes will be applied in updates and additionally sent out to notify relevant people subscribed to those alerts.

Kube-hunter checks for misconfiguration, but can't find any (non-false-positive) openings with default settings. Would need zero-day / known vuln in Microsoft or red hat configs

Same as OCP, except accessible from anywhere (cloud, duh). The master components are updated, configured and maintained by Microsoft, only when a restart is required the cluster administrator has to trigger it manually. -> increases total risk value slightly, both are still rated medium in the end

### 4.2.5 V05 - Supply compromised container (base) image

This has two facettes: it can be untargeted (image spraying) and targeted (compromising a specific image known to be used by the target).

The untargeted version needs the least access, since it simply needs a (free) dockerhub account to upload malicious images that could or couldn't fulfil the function they are advertised to do. This is done in the hopes of someone downloading that image for use in his own environment, thus starting attacker-supplied containers within their cluster. This could allow an attacker remote access to a container in the cluster and/or exfiltrate information. Even without injecting malware, an attacker could mislabel old software versions as newer ones so software with known vulnerabilities is deployed because it is though to be up to date.

The targeted version could be specialized uploads to docker hub (similar to broad phishing vs. spear phishing) or "poisoning" an internal container image repository.

Image builds run as root, which could further be exploited – but this would need a vulnerability in the OCP / Azure build process.

These methods are publically known and both the docker container runtime and docker hub actively try to mitigate this, but malicious images are only deleted when reported by enough users and the security settings within the container runtime are not set by default. Base containers and malware / known vulnerable versions are readily available from public sources, but need some technical expertise to plug together.

Exactly the same as OCP

### 4.2.6 V06 - Supply compromised k8s configuration

Similar to V07, this can be done either untargeted by spraying to tutorials / help forums or targeted, similar to spear phising. If a cluster administrator does not fully analyze or understand the configuration he gets from public sources, the cluster could be compromised fully, i.e. by implementing backdoors through malicious containers with special access and ability to be remotely accessed by the attacker.

Examples are readily available from public sources, but need some technical expertise to plug together.

Exactly the same as OCP

### 4.2.7 V07 - Compromise other application components (lateral movement)

Once an attacker sits within a container, he can scan the network for other containers, hosts, services, apis or similar interfaces to further his access. By default, all containers in all projects (except master & infra components) are put in the same subnet, allowing everyone to communicate with anyone else. This is especially troubling for securing an environment with multiple tenants – even if the DB is not publically accessible, unauthorized access can be leveraged by anyone in the cluster.

Scanning tools like nmap etc. to find components to talk to are readily available, but their results are cluster-specific (everyone runs something different). Therefore some technical expertise is needed to leverage the network access needed.

Same as OCP. The worst AKS-specific problem with this is the mitigation. This risk is not clearly documented in the setup section of the documentation. If one stumbles upon this information in further sections of the docs after setting up his cluster, he might postpone or deny changing the setting to isolate different projects by default. This is because a full cluster rebuild is needed to change this setting!

### 4.2.8 V08 - Container breakout (R/W, Privilege Escalation)

A deployed container poses the risk of allowing access to the node it is running on, thus allowing an attacker to "break out" of the container and perform actions on the node. This poses a considerable threat, since any container may run on any node by default, allowing an attacker full access to any containers running on the node he controls, which will – especially over time – have a great chance to include containers belonging to other projects.

The OCP default settings limit the possibility of this dramatically, the risk lies more in organizations relaxing the defaults in favour of easy usability. (A majority of container images straight from docker hub require UID 0, which is denied by the default SCC 'restricted' in OCP during admission. This results in crashlooping and non-functional containers, developers would need to customize any image themselves. The easiest way to stop those problems this is to permit the default service account within a project access to the 'privileged' SCC permissions. This would significantly increase the risk of a container breakout!)

This is probably the most-talked about attack vector regarding containers, but techniques are not obviously documented and breakout methods would have to be customized to the restrictions applied within a cluster.

Difference to OCP: containers can be run with UID 0 and more relaxed settings in general by default. User-namespace remapping not in place by default, vastly increasing the risk of a container breakout! This is more on the usability>security side of things. -> raises risk, jumping from medium to high.

### 4.2.9 V09 - Compromise local image cache

If you can swap out the cached container image on a host, the swapped-in version will run the next time this node spins up this container. This is a very sneaky way to inject a malicious container, but within the default settings, access to the host file system is required.

Not well known and not entirely trivial to do (sneakily).

Same as OCP.

### 4.2.10 V10 - Modify running container

Instead of deploying a container with malicious contents, an attacker can try to modify and use an already running container to its needs by loading additional tools/binaries, changing configurations

or exfiltrating data. This could be done through an RCE vuln, ssh access or others, just like any compromised linux machine.

-> Common sense to do this, same technical level as any command line interaction with a linux system.

Same as OCP.

### 4.2.11 V11 - Hoard resources (sabotage)

With enough access or restrictions too lax, an attacker may be able to seriously halt the availability of all workloads processed by the cluster by misconfiguration, conducting DOS attacks or wiping nodes or cluster configurations. Since it is a complex system, finding the sabotaged component can take considerable know-how and time if done well, increasing the impact – especially in on-premise environments, where resources are limited.

Wiping is common sense, sabotaging the cluster in a complex and effective way may take deeper knowledge and be customized to the environment.

Difference to OCP: you can easily spin up more resources in the cloud -> less impact -> risk decreases by a considerable margin, high to medium

### 4.2.12 V12 - Misuse resources (cryptojacking)

In contrast to V14, an attacker will try to be sneaky if done well. The goal here is to (ab)use the computing resources not belonging to and payed for by him to achieve monetary gain though mining cryptocurrencies.

Cryptojacking is regularly cited as an up-and-coming attack, but to do it with a low risk of being detected needs some technical skill.

Difference to OCP: an attacker can easily spin up more resources in the cloud -> more impact -> risk increases by a considerable margin, medium to high

### 4.2.13 V13 - Add malicious container

Instead of manipulating running containers, an attacker with user access and permissions to spin up containers may start their own ones. (BYOC – bring-your-own-container?) This is still restricted by container admission restrictions on the user/project, but at least he can install all needed binaries beforehand and his shell doesn't die whenever the underlying container might be stopped.

Doing this is common sense, as before some technical skill is required to prepare a malicious container

Same as OCP, except accessible from anywhere (cloud, duh). -> increases total risk value slightly, both are still rated medium in the end

### 4.2.14 V14 - Add malicious node

An attacker could try to add a malicious node to the cluster and inspect or manipulate data in or exfiltrate data from containers scheduled on it. Since any container may run anywhere, there is a high chance of all containers eventually being run on a given node over time, exposing the whole cluster to an attacker. This could be sped up by manipulating the reports of remaining resources on the node towards the scheduler. By design, cluster administrator access is needed to add a node within OCP.

This technique is not talked about that much, but still available in public resources and possible in all clusters. Docs are publically available to add nodes to a cluster, basic linux server administration skills are needed to follow them.

Accessible from anywhere (cloud). -> total risk value unchanged In contrast to OCP, you can spin up additional nodes more easily in AKS if configured on creation, but to access/control/manipulate them you still need cluster administrator access. A tutorial on getting ssh access is available, but that's lengthy and not trivial.

### 4.2.15 V15 - Bad user practice (outside of cluster)

This vector comprises user practices outside of the cluster that lead to risks within it. Examples include phishing, openly publishing keys/tokens to public code repositories, password reuse, scouting specific software or container images used, publishing logs with information valuable to an attacker and more. Could be done targeted (i.e. specific OSINT) or untargeted through github crawlers, scanning account/password dumps, ... Whatever you get could be used to access the cluster with the permissions granted by service-/user-accounts or as a reconnaissance base for further attacks.

There are tools available to do this, using them effectively requires some technical skill.

Same as OCP, except accessible from anywhere (cloud, duh). -> doesn't change total risk value

### 4.2.16 V16 - Incufficient base infrastructure hardening

The underlying nodes could allow an attacker easy entry, even if the containers themselves are hardened. This includes Side-Channel attacks like Spectre & Meltdown, open ports on the servers exposed by other stuff running on it, being available from the public www, ... Vector exists mostly to sink all "classic" infra security measures in it, since those are researched and available everywhere and very much not the focus of the thesis.

Among worst case: unauthenticated access to run commands which is hosted publically on the internet for anyone to access and indexed by shodan. Bye bye cluster. (Too many scenarios to hypothesize here, I'll just point the finger at conventional server & infra hardening standards and guidelines)

-> Well known, still needs some technical skill to find vulns and exploit them

Suprisingly, same as OCP (despite the azure promise of PaaS-we-manage-your-infra)! That's the case since security updates on nodes that require a reboot are not done automatically, but have to be triggered manually or configured to trigger automatically. Remediation is far less work though.

### 4.2.17 V17 - Entry through known, unpatched vulnerabilities

Sinkhole vector for patch management. Would be a measure against every preceding vector otherwise. Worst case could be anything, thus maximum risk. (See kubernetes CVE with 9.8 / 10)

To check for this is common sense, some technical skill may be needed to find and exploit unpatched stuff.

Same as OCP. Even infra still needs user interaction to be patched, see preceding vector.

# 5 Managing the attack surface risk

With this chapter, the OWASP Risk Rating Methodology[2] is continued. As completing the risk management process would exceed the scope of this thesis, the full process will be described and the risk management steps will be demonstrated through two exemplary vectors.

## 5.1 Defining procedures and approach

Normally one should start with the vector currently having the highest overall risk. We chose two other vectors, since these were more representative of the environments within scope and concise enough to properly present the process. Securing the underlying physical servers or VMs is not an optimal example representing the process of securing a solution based on orchestrated containers.

TODO would start with highest vector, check possible measures and implement when acceptable/applicable. reassess vector risk and either implement more XOR accept residual risk if still highest. otherwise put back into queue -> pick up now highest vector and check possible measures, ... -> repeat until risks are either all below target threshold or all above are accepted.

## 5.2 Managing the risk of V07 - Lateral movement from container

TODO this was done in both AKS and OCP, is possible with both default configs!

### 5.2.1 Demonstrating the successful attack without security measures

LATERAL MOVEMENT OCP COMMANDS - SET STAGE Display projects: oc projects Lookup service ip from server (look for user-db ip) : oc get service -n sock-shop Show container used + scc's needed for it: cat network-utils.yaml oc adm policy scc-review -z default -f network-utils.yaml Start container and enter it: oc apply -f network-utils.yaml oc rsh network-utils

2 Fou19a.

LATERAL MOVEMENT OCP SET STAGE DUMP (see latex source file here)

LATERAL MOVEMENT OCP COMMANDS

LATERAL MOVEMENT OCP TEXT DUMP LATERAL MOVEMENT AKS COMMANDS

Display projects: kubectl get namespace Lookup service ip from server (look for user-db ip) : kubectl get service -n sock-shop Show container used: cat network-utils.yaml Start container and enter it: kubectl apply -f network-utils.yaml kubectl exec -it network-utils – /bin/bash

LATERAL MOVEMENT AKS SET STAGE DUMP LATERAL MOVEMENT COMMANDS AKS LATERAL MOVEMENT AKS TEXT

### 5.2.2 Selecting security measures

TODO use multitenant networkplugin (OCP) and network policies (k8s)

LATERAL MOVEMENT OCP REMEDIATION DUMP LATERAL MOVEMENT AKS REMEDIATION COMMANDS

LATERAL MOVEMENT AKS REMEDIATION DUMP

### 5.2.3 Demonstration with implemented security measures

TODO move pics, listings from above here

### 5.2.4 Risk reassessment 1

TODO updated table of risk for ocp AND aks

## 5.3 Managing the risk of V08 - Container Breakout

TODO this was done on AKS and OCP, but needed changes in default OCP settings to work.

### 5.3.1 Demonstrating the successful attack without security measures

BREAKOUT DEMO OCP COMMANDS CONTAINER BREAKOUT OCP TEXT DUMP CONTAINER BREAKOUT OCP RESULT DUMP CONTAINER BREAKOUT AKS PREPARATION - SSH TO NODE CONTAINER BREAKOUT AKS COMMANDS CONTAINER BREAKOUT AKS TEXT DUMP CONTAINER BREAKOUT AKS RESULT DUMP

### 5.3.2 Selecting security measures

TODO dont allow privileged containers (scc in OCP, podsecuritypolicies in AKS)

CONTAINER BREAKOUT OCP REMEDIATION DUMP CONTAINER BREAKOUT AKS REMEDIATION CMDs CONTAINER BREAKOUT AKS REMEDIATION DUMP

### 5.3.3 Demonstration with implemented security measures

TODO move pics, listings from above here

### 5.3.4 Risk reassessment 2

TODO SECOND updated table of risk for ocp AND aks <- this one starts with first updated one

## 5.4 How to proceed

TODO would start with highest vector, check possible measures and implement when acceptable/applicable. reassess vector risk and either implement more XOR accept residual risk if still highest. otherwise put back into queue -> pick up now highest vector and check possible measures, ... -> repeat until risks are either all below target threshold or all above are accepted.

# 6 Conclusion

This chapter aims to provide answers to the remaining questions posed in chapter 1.2.

### 6.0.1 On-premise and public cloud environment comparison

TODO 2 subsubsections, generic and solution-example specific?

TODO both have pros and cons, regardless of specific solution picked. Specific solutions have their pros and cons, too. cloud risk may be higher initially (both in the solutions we looked at and in general), but there are measures to reduce that to acceptable level for baseline security. some usecases benefit from buy-iaas-provide-paas, since more control with you as provider. high-security requirements on-prem probably better (and may be needed for compliance). -> our POV both work, would have to be decided on a case-by-case basis as they depend on many factors (integration with other offers or services very valuable for cloud-specific, pre-existing on-prem datacenters make on-prem easier).

### 6.0.2 Multi-tenant isolation

TODO namespace/project isolation vs. cluster isolation. refer to separate-tenants-by-cluster measure in vector identification section. Is possible, but more complex and tedious ATM. refer to growing projects for cluster federation and clusters-as-cattle (instead of container-as-cattle and cluster-as-pet) -> may be better in future, only do today for super-high-security apps in their own cluster. costs more resources, at which point one may simply run the stuff on isolated and dedicated servers anyway (except when needed for workflow, high availability or mircoservice arch offsets complexity needed)?

### 6.0.3 Summary

TODO summary of thesis process - huge scope, could have been better defined beforehand. risk estimate formula still isnt perfect, but time would probably still be better allocated to practical stuff or more research, both for thesis and others trying to follow the process. no use in better knowing which

one is the biggest problem if no time left to reduce risk in the end. hope that more sources are published and industry standards crystallize so not everyone has to do that much research on their own.

TODO outlook - stuff will continue to move fast, hopefully with security in mind. we are currently far better with security than i.e. pre k8s-1.8. more security features will be introduced or go stable (refer to future k8s pod sandbox option in YAML files). as mentioned in multi-tenant isolation, cluster federation might help and will introduce new solutions, but probably also new problems. rootless builds, currently through buildah etc, becoming more mainstream will probably be the next big security thing here.

TODO commentary - hope this helps someone and spares some time. am excited for future tools and platform developments.

# Bibliography

## Books


[Irw14] Stephen Irwin. *Creating a Threat Profile forYour Organization*. SANS Institute, 2014, 12 to 17 (cit. on p. 2).

[Liz18] Michael Hausenblas Liz Rice. *Kubernetes Security - How to Build and Operate Applications Securely in Kubernetes*. O'Reilly Media, Inc., 2018 (cit. on p. 3).

[Mur11] Karen Scarfone Murugiah Souppaya John Morello. *NIST Special Publication 800-190 Application Container Security Guide*. National Institute of Standards and Technology, U.S. Department of Commerce, 2011 (cit. on p. 3).

[Red19b] Inc. Red Hat. *OpenShift Container Platform 3.11 Container Security Guide*. Red Hat, Inc., 2019 (cit. on p. 3).

[ST11] National Institute of Standards and Technology. *SP 800-145 The NIST Definition of Cloud Computing*. U.S. Department of Commerce, 2011, 2 to 3 (cit. on pp. 5, 6).

[BC18] Matt Butcher and Karen Chu. *Phippy goes to the Zoo*. The Linux Foundation, 2018, p. 4 (cit. on p. 11).


## Online sources


[Pro19a] The Kubernetes Project. *Understanding Kubernetes Objects*. 2019. URL: `https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/` (visited on 07/10/2019) (cit. on pp. ix, 11, 14).

[Red18a] Inc. Red Hat. *Generally Available today: Red Hat OpenShift Container Platform 3.11 is ready to power enterprise Kubernetes deployments*. 2018. URL: `https://www.redhat.com/en/blog/generally-available-today-red-hat-openshift-container-platform-311-ready-power-enterprise-kubernetes-deployments` (visited on 07/08/2019) (cit. on p. 2).

[Fou19a] OWASP Foundation. *OWASP Risk Rating Methodology*. 2019. URL: https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology (visited on 04/05/2019) (cit. on pp. 2, 41).

[Fou19b] OWASP Foundation. *Threat Modeling Cheat Sheet*. 2019. URL: https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Threat_Modeling_Cheat_Sheet.md#define-all-possible-threats (visited on 07/05/2019) (cit. on p. 2).

[Int19a] Center for Internet Security. *CIS Docker Community Edition Benchmark*. 2019. URL: https://github.com/OWASP/Docker-Security (visited on 07/11/2019) (cit. on p. 3).

[Int19b] Center for Internet Security. *CIS Kubernetes Benchmark*. 2019. URL: https://github.com/OWASP/Docker-Security (visited on 07/11/2019) (cit. on p. 3).

[Red19a] OWASP Foundation RedGuard. *Container Security Verification Standard*. 2019. URL: https://github.com/OWASP/Container-Security-Verification-Standard (visited on 04/24/2019) (cit. on p. 3).

[Fou19c] OWASP Foundation. *Docker Security*. 2019. URL: https://github.com/OWASP/Docker-Security (visited on 04/24/2019) (cit. on p. 3).

[Mic18] Microsoft. *Best practices for cluster security and upgrades in Azure Kubernetes Service (AKS)*. 2018. URL: https://docs.microsoft.com/en-us/azure/aks/operator-best-practices-cluster-security (visited on 07/11/2019) (cit. on p. 3).

[McC18] Rory McCune. *A Hacker's Guide to Kubernetes and the Cloud - Rory McCune, NCC Group PLC (Intermediate Skill Level) [29:18]*. 2018. URL: https://youtu.be/dxKpCO2dAy8?t=1758 (visited on 04/25/2019) (cit. on p. 4).

[Wat17] Stephen Watts. *SaaS vs PaaS vs IaaS: What's The Difference and How To Choose*. 2017. URL: https://www.bmc.com/blogs/saas-vs-paas-vs-iaas-whats-the-difference-and-how-to-choose/ (visited on 04/02/2019) (cit. on p. 6).

[Cor19a] Microsoft Corporation. *What is PaaS? Platform as a Service*. 2019. URL: https://azure.microsoft.com/en-us/overview/what-is-paas/ (visited on 04/02/2019) (cit. on p. 6).

[Cor19b] Microsoft Corporation. *What is PaaS? Platform as a Service*. 2019. URL: https://docs.microsoft.com/en-us/azure/aks/operator-best-practices-cluster-security?view=azuremgmtbilling-1.1.0-preview%5C#process-linux-node-updates-and-reboots-using-kured (visited on 04/03/2019) (cit. on p. 7).

[Inc19a] Docker Inc. *What is a Container?* 2019. URL: https://www.docker.com/resources/what-container (visited on 07/08/2019) (cit. on p. 7).

[Raa18] Mike Raab. *Intro to DockerContainers*. 2018. URL: https://static.rainfocus.com/oracle/oraclecode18/sess/1513810380873001uIiU/PF/docker-101-ny_1520531065990001vPkT.pdf (visited on 07/08/2019) (cit. on p. 7).

[Osn18] Rani Osnat. *A brief history of containers: From the 1970s to 2017*. 2018. URL: https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016 (visited on 07/08/2019) (cit. on p. 7).

[Pro19b] The Kubernetes Project. *What is Kubernetes*. 2019. URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/#going-back-in-time (visited on 07/08/2019) (cit. on p. 8).

[Cha17] Nick Chase. *OK, I give up. Is Docker now Moby? And what is LinuxKit?* 2017. URL: https://www.mirantis.com/blog/ok-i-give-up-is-docker-now-moby-and-what-is-linuxkit/ (visited on 07/08/2019) (cit. on pp. 8, 15).

[Inc19b] Docker Inc. *About Docker Engine*. 2019. URL: https://docs.docker.com/engine/ (visited on 07/08/2019) (cit. on p. 8).

[Red19c] Inc. Red Hat. *Getting Started with Containers*. 2019. URL: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_atomic_host/7/html-single/getting_started_with_containers/index#working_with_docker_formatted_containers (visited on 07/08/2019) (cit. on p. 8).

[Red18b] Inc. Red Hat. *Containers and Images*. 2018. URL: https://docs.openshift.com/enterprise/3.0/architecture/core_concepts/containers_and_images.html#docker-images (visited on 07/08/2019) (cit. on p. 8).

[Inc18] Docker Inc. *Dockerfile Reference*. 2018. URL: https://docs.docker.com/engine/reference/builder/ (visited on 07/08/2019) (cit. on p. 8).

[Fou19d] The Linux Foundation. *Open Container Initiative*. 2019. URL: https://www.opencontainers.org/ (visited on 07/08/2019) (cit. on p. 9).

[Lew19a] Ian Lewis. *Container Runtimes Part 2: Anatomy of a Low-Level Container Runtime*. 2019. URL: https://www.ianlewis.org/en/container-runtimes-part-2-anatomy-low-level-contai (visited on 07/08/2019) (cit. on p. 9).

[Lew19b] Ian Lewis. *Container Runtimes Part 3: High-Level Runtimes*. 2019. URL: https://www.ianlewis.org/en/container-runtimes-part-3-high-level-runtimes (visited on 07/08/2019) (cit. on p. 9).

[Pro16] The Kubernetes Project. *Container Runtimes Part 3: High-Level Runtimes*. 2016. URL: https://github.com/kubernetes/kubernetes/blob/242a97307b34076d5d8f5bbeb1docs/devel/container-runtime-interface.md (visited on 07/08/2019) (cit. on p. 9).

[Pro19c] The Kubernetes Project. *What is Kubernetes*. 2019. URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (visited on 07/09/2019) (cit. on p. 9).

[Fou19e] The Linux Foundation. *There are over 80 Certified Kubernetes offerings.* 2019. URL: https://www.cncf.io/certification/software-conformance/ (visited on 07/10/2019) (cit. on pp. 10, 15).

[Fou19f] The Linux Foundation. *Sustaining and Integrating Open Source Technologies*. 2019. URL: https://www.cncf.io/ (visited on 07/09/2019) (cit. on p. 10).

[Pro19d] The Kubernetes Project. *Kubernetes Components*. 2019. URL: https://kubernetes.io/docs/concepts/overview/components/ (visited on 07/09/2019) (cit. on p. 10).

[Pro19e] The Kubernetes Project. *Pod Overview*. 2019. URL: https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/#understanding-pods (visited on 07/10/2019) (cit. on p. 11).

[Pro19f] The Kubernetes Project. *ReplicaSet*. 2019. URL: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/ (visited on 07/10/2019) (cit. on p. 12).

[Pro19g] The Kubernetes Project. *Picking the Right Solution*. 2019. URL: https://v1-13.docs.kubernetes.io/docs/setup/pick-right-solution/# (visited on 07/10/2019) (cit. on p. 14).

[Pro19h] The Kubernetes Project. *The Kubernetes API*. 2019. URL: https://kubernetes.io/docs/concepts/overview/kubernetes-api/ (visited on 07/10/2019) (cit. on p. 14).

[Red19d] Inc. Red Hat. *OpenShift plans and pricing*. 2019. URL: https://www.openshift.com/products/pricing/ (visited on 07/10/2019) (cit. on p. 15).

[Har19] Brian Harrington. *OpenShift and Kubernetes: What's the difference?* 2019. URL: https://www.redhat.com/en/blog/openshift-and-kubernetes-whats-difference (visited on 07/10/2019) (cit. on p. 15).

[Red19e] Inc. Red Hat. *The Origin Community Distribution of Kubernetes that powers Red Hat OpenShift.* 2019. URL: https://www.okd.io/ (visited on 07/12/2019) (cit. on p. 15).

[Red19f] Inc. Red Hat. *OpenShift Container Platform 3.x Tested Integrations*. 2019. URL: https://access.redhat.com/articles/2176281 (visited on 04/09/2019) (cit. on p. 15).

[Red19g] Inc. Red Hat. *Get Started with the CLI*. 2019. URL: https://docs.openshift.com/container-platform/3.11/cli_reference/get_started_cli.html (visited on 07/10/2019) (cit. on p. 15).

[Red19h] Inc. Red Hat. *Managing Projects*. 2019. URL: https://docs.openshift.com/container-platform/3.11/admin_guide/managing_projects.html (visited on 07/10/2019) (cit. on p. 15).

[Red19i] Inc. Red Hat. *Using the CRI-O Container Engine*. 2019. URL: https://docs.openshift.com/container-platform/3.11/crio/crio_runtime.html (visited on 07/10/2019) (cit. on p. 15).

[Mic19] Microsoft. *Integrate Azure Active Directory with Azure Kubernetes Service*. 2019. URL: https://docs.microsoft.com/en-us/azure/aks/azure-ad-integration (visited on 07/10/2019) (cit. on p. 15).

[Pro19i] The Kubernetes Project. *Kubernetes version and version skew support policy*. 2019. URL: https://kubernetes.io/docs/setup/release/version-skew-policy/#supported-versions (visited on 07/11/2019) (cit. on p. 17).

[Fou17] OWASP Foundation. *OWASP Top 10 -2017 The Ten Most Critical Web Application Security Risks*. 2017. URL: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf (visited on 04/03/2019) (cit. on p. 27).

[Gee17] Brad Geesaman. *Hacking and Hardening Kubernetes Clusters by Example [I] - Brad Geesaman, Symantec [3:05]*. 2017. URL: https://www.youtube.com/watch?v=vTgQLzeBfRU&feature=youtu.be&t=185 (visited on 04/16/2019) (cit. on p. 27).

## Other sources

[Mei19] Nico Meisenzahl. *docker-drawings*. A collection of graphics given to Lukas Grams by Nico Meisenzahl at the Global Azure Bootcamp Rosenheim on 2019-04-27. 2019 (cit. on pp. 11–13).