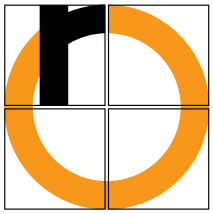
Hochschule **Rosenheim**
University of Applied Sciences



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, containerisierter Datenverarbeitung und Microservice-Architekturen bieten neue Angriffsszenarien. Dies erhöht den Bedarf an neuen Verteidigungs-Strategien in der IT-Sicherheit von Entwicklungs- und Produktiv-Betriebsumgebungen. Gerade Lösungen auf Basis von (Kubernetes konformer) Container-Orchestrierung sind stark gefragt und haben einen sehr unterschiedlichen Aufbau im Kontrast zu konventionellen und etablierten 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 selbst betriebene 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.
- Angriff von böswilligen Dritten sowie böswilliger oder fahrlässiger 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 zielführende 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, containerized workloads 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 high in 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 company network 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, result oriented 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, section 'Original reference image'.
2 Aut19c, section 'Going back in time'.
3 Mei19.
4 Mei19.
5 Mei19.

List of Tables

Listings

1 Aut19a.

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
LXC	LinuX Containers
OCI	Open Container Initiative
OCP	OpenShift Container Platform
OKD	Origin Community Distribution of Kubernetes
OWASP	Open Web Application Security Project
PaaS	Platform-as-a-Service
RBAC	Role-based access control
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, containerized workloads and microservice architectures introduce new attack scenarios. This creates the need for new defense strategies in both Development and Operations. Especially solutions providing Kubernetes (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 (see figure A.1), the available v1.11 was chosen to improve comparability.

The components of these two solutions 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 includes state-sponsored actors deploying zero-day exploits, which are not applicable to a majority of solutions deployed and thus disregarded for the given context. This thesis aims to provide insight to the risks of providing a PaaS solution and mitigation 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 a given 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 actors 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 company network 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 Inc18a, see headline and date.
2 Fou19a.
3 Fou19b, Section 'Define all possible threats'.
4 Irw14, p. 12 to 17.

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 work in progress 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 k8s in general[9] as well as OCP[10] and AKS[11] specifically. In addition to that, recorded and openly available talks by different people in the industry are available. Especially the talks at KubeCon, the annual k8s conference, are accessible through the Cloud Native Computing Foundation (CNCF) YouTube presence[12]. The 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 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

1 RH18.
2 SMS11.
3 Int19a.
4 Int19b.
5 RH18.
6 SMS11.
7 RF19.
8 Fou19c.
9 Aut19b.
10 Inc19a.
11 Cor18.
12 Channel Link: https://www.youtube.com/channel/UCvqbFHwN-nwalWPjPUKpvTA/videos

even intended to be followed in full, but just as basis for security considerations.[1] In accordance to this, the findings can only be provided on a best-effort basis and not without reservation towards possible changes through future software versions or new information. This should be an information basis and approach for rating your own implementation while taking the business risk into consideration, not a definitive guide to all aspects in this regard. 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 the limitations of this research has to be emphasized.

1 McC18, starting at 29:18.

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 delegates the management and control of the infrastructure needed to deploy his applications to his IaaS provider. 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 availability. As shown in Figure 2.1, consumers are responsible for the OS layer and everything above it.

2 ST11, p. 2 to 3.

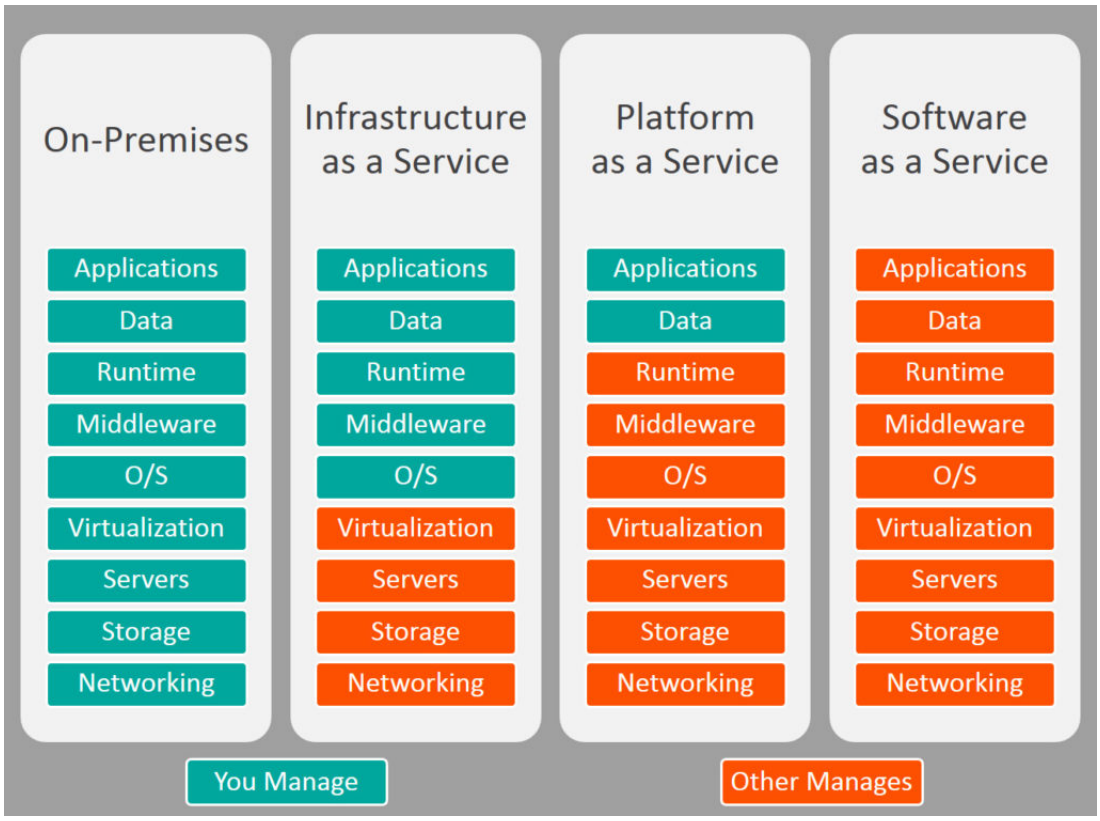


Figure 2.1 Comparison of responsibilities in different service models[a]

a Wat17, section 'Original reference image'.

2.2 Platform-as-a-Service

In Platform-as-a-Service (PaaS) environments, a consumer delegates the management and control of even more resources needed to deploy his applications to his PaaS provider, 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 database management or log 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.

1 ST11, p. 2 to 3.
2 Cor19a, section 'Advantages of PaaS'.

A consumer might also 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 applications 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, section 'Process Linux node updates and reboots using kured'.
2 Inc19b, section 'Package Software into Standardized Units for Development, Shipment and Deployment'.
3 Raa18, slide 13.
4 Osn18, section '2013: Docker'.

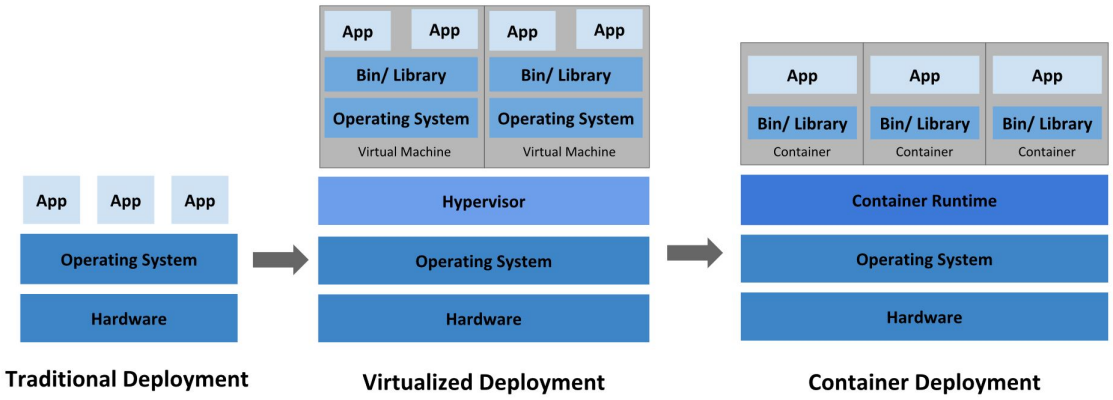


Figure 2.2 Comparison of different application deployments on the same hardware[a]

a Aut19c, section 'Going back in time'.

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[2], which will be talked about in the section 2.3.4. Some sources also talk about docker-formatted containers when those technologies 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, section 'What is Moby?'
2 Inc19c, section 'Develop, Ship and Run Any Application, Anywhere'.
3 Inc19d, section '1.11. Working with Docker formatted containers'.
4 Inc18b, section 'Docker Images'.
5 Inc18c, section 'Dockerfile reference'.

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 adjust 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 the Linux

1 Fou19d, first paragraph.
2 Lew19a, section 'Examples of Low-Level Container Runtimes'.
3 Lew19b, Intro and section 'Examples of High-Level Runtimes'.
4 Aut19d, section 'Purpose'.
5 Aut19a, section 'Understanding Kubernetes Objects'.

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

Kubernetes components

In order to deliver a working 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.[2]

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; Cluster user or Administrator 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 started by the k8s cluster. 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 as depicted in figure 2.2, which is responsible for actually running containers. Examples include OCP using Docker while AKS uses Moby, but any implementation of the CRI is supported.[3]

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, section 'There are over 80 Certified Kubernetes offerings.'
2 Aut19e, section 'Master Components'.
3 For additional information, refer to section 2.3.4

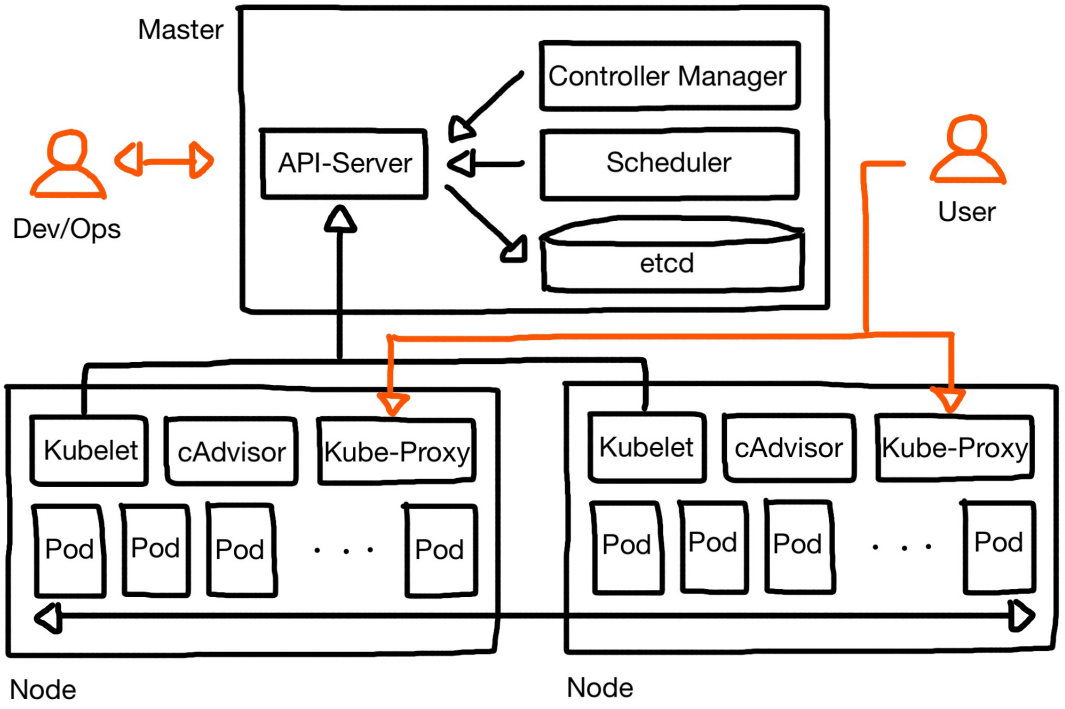


Figure 2.3 An overview of different k8s components[a]

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 behavior 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 Aut19a, section 'Understanding Kubernetes Objects'.
2 Aut19f, section 'Understanding Pods'.
3 BC18, p. 4.

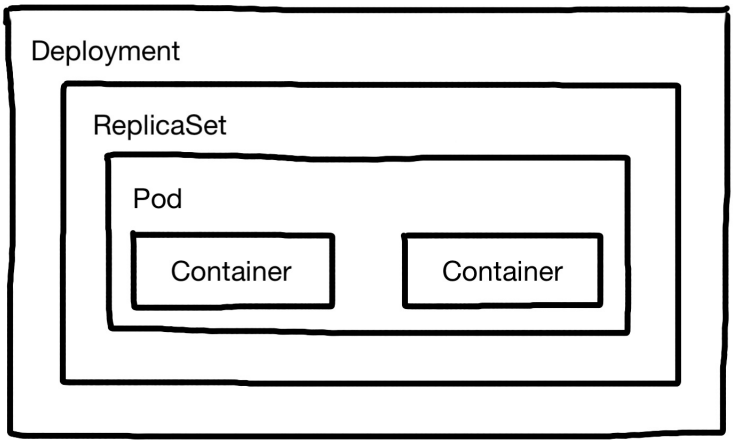


Figure 2.4 Connection between deployments and other k8s objects down to containers[a]

a Mei19.

Pods are usually not created manually, but started and stopped by a replicaSet. Their purpos as a k8s controller 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 Aut19g, introductory sentence.

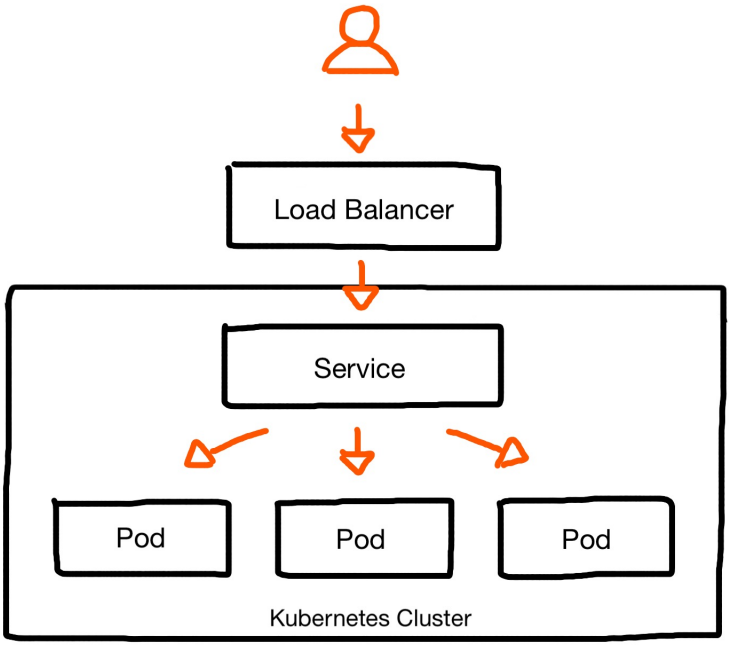


Figure 2.5 Connection between deployments and other k8s objects down to containers[a]

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 typically 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 Aut19h, sections 'Turnkey Cloud Solutions' and 'On-Premises turnkey cloud solutions'.
2 Aut19i, section 'The Kubernetes API'.
3 Aut19a, section 'Describing a Kubernetes Object'.

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 in the Azure public cloud. As it does not differ from the k8s reference implementation compared to OCP, its functionality is identical to the general k8s mechanics described in section 2.4.1. 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 Inc19e, section 'OpenShift plans and pricing'.
2 Har19, section 'OKD vs Red Hat OpenShift'.
3 Inc19f, section 'What is OKD?'.
4 Inc19g, table 'Platform Components'.
5 Fou19e, table 'Platform - Certified Kubernetes - Distribution'.
6 Inc19h, section 'Basic Setup and Login'.
7 Inc19i, section 'Overview'.
8 Inc19j, section 'Getting CRI-O'.
9 Cha17, section 'What is Moby?'
10 Cor19c, first paragraph.

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 are released 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 may provide specific examples for AKS and OCP for further illustration. 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 broadly 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 Aut19j, section 'Supported versions'.

3.2 Identified vectors and security measures

The following headlines will each respectively 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 V*xx*.

3.2.1 V01 - Reconnaissance through interface components

An attacker could gather information through the available interfaces in order to prepare further attacks and look for vulnerabilities to exploit. A k8s cluster has many interfaces, some of which are intended to be accessed by users or other non-master-components. These components include the kube-apiserver introduced in section 2.4.1, the popular web-based k8s user interface called Dashboard[1] as well as solution-specific interfaces like the OCP web console[2] or the Azure Portal for AKS[3]. Other interfaces are intended to be accessible from inside a cluster. These may include the components accessible from cluster-external sources, but also APIs for the integration of additional functionality. A common use case for such APIs is the provisioning of additional cloud resources in order to scale up the cluster. One example of reconnaissance against the kube-apiserver was demonstrated at KubeCon 2019[4], while an example of leveraging the cloud api can be found in a blog post[5].

Following the security principle of limiting the attack surface[6], the measures against this include blocking access to or outright deactivating interfaces. If interfaces are unused, i.e. the k8s Dashboard where only CLI tools are used for cluster interaction, one can easily deactivate the Dashboard and thus reduce the attack surface without further work needed. If an interface is used, but alternatives exist, it may be advisable to evaluate whether a change in workflows or software architectures in order to decommission the interface may provide an overall benefit. Least privilege, another security principle[7], can be followed by restricting access to used interfaces. Authentication can be required of both humans and automated processes and their access may be restricted to the information and capabilities strictly needed to operate. KubiScan, an open source tool to scan the Role-based access control (RBAC) permissions of a k8s cluster for permissions considered risky, is publicly available to assess a given cluster. Users might be required to log in through a two factor authentication to hinder access through stolen credentials. Network restrictions can be put in place to restrict the vantage points from which an attacker may gain access. Following Defense in Depth[8], this should not only be done from the

1 Aut19k, first paragraph.
2 Inc19k, section 'Overview'.
3 Cor19d, section 'Create an AKS cluster'.
4 Ric19, starting at 8:45.
5 Gom18.
6 RH18, p. 4 to 5.
7 RH18, p. 3 to 4.
8 RH18, p. 3.

outside, but access from inside the cluster, too. As an example, most containers don't need access to k8s master components, so these requests can be blocked by k8s Egress Network Policies[1]. The impact of successful attacks could be limited through Alerting and restriction of the attacks 'blast radius'. Logging and alerting processes may be implemented through k8s audit policies[2] for requests that are considered suspicious and do not occur in normal operations, examples of which can be found in online sources[3]. The 'blast radius' may be reduced by isolating different applications or teams trough multiple namespaces and restricting access to non-namespaced resources to cluster administrators, thus isolating an attacker to a single namespace instead of the whole cluster. An even more drastic isolation could be achieved through isolation by cluster, resulting in separate k8s master components per isolated entity.

3.2.2 V02 - Reading confidentials through interface components

The components mentioned in section 3.2.1 could also be used by an attacker to gather confidential information, which can be either useful in and of itself or used to gain further access. Extracted private keys, tokens or passwords may be leveraged to authenticate with additional privileges in order to steal or manipulate container images, adjacent applications, source code and interact with other interfaces.

Despite presenting a more serious damage potential, the measures of section 3.2.1 also apply here since they target the same interfaces. In addition to that, special attention should be paid to the way secrets are passed into code running under k8s. Embedding secret information into container images should be avoided, since those are not only unencrypted and difficult to change, but may be extracted by an attacker reading the Dockerfile or inspecting an image themselves. Supplying them through environment variables is also not recommended, as those might be exposed in unconsidered ways. It is recommended to supply them to containers at runtime through k8s secrets or external tools like those integrated into cloud platforms or offered by third-party providers. Popular third-party solutions include HashiCorp Vault and CyberArk Conjur.[4]

3.2.3 V03 - Configuration manipulation through interface components

With write access to the components mentioned in section 3.2.1, an attacker may change configurations of the cluster or its environment, i.e. security controls set in place through the cloud provider platform. Examples include disabling security restrictions, rendering alerting processes dysfunctional by blocking their communication or swapping container images with functional, but compromised alternatives.

1 Aut19l, section 'The NetworkPolicy Resource'.
2 Aut19m, section 'Audit Policy'.
3 San17, section 'Alerting on the Kubernetes infrastructure'.
4 RH18, chapter 7.

As this has a higher potential for damage, special care should be taken. The measures of section 3.2.1 are applicable here, too. Special thought may be put into restricting write permissions to the most privileged user accounts, i.e. cluster or cloud administrators for actions that can influence systems outside of a namespace or cluster. In order to automatically deny actions, k8s Pod Security Policies[1] may be leveraged. Since these are non-namespaced k8s objects themselves, they could be circumvented by anyone with permissions to manipulate non-namespaced objects, i.e. cluster administrators.

3.2.4 V04 - Compromise internal master components

In contrast to the components mentioned in section 3.2.1, there are others which are only intended to be interacted with by other k8s master components. They may also be compromised by an attacker. We will refer to these as internal components. The most promising target is the etcd key value store introduced in section 2.4.1, as through it an attacker may gain any permission they want[2].

The available security measures are much more straightforward than those of the interface components, which is why all (ab)use cases have been united in one vector. Master components can be run on any machine in the cluster, but are often run on dedicated master nodes.[3] Therefore, k8s solutions can restrict access to these by either not exposing them to anyone outside the pool of master nodes or, in case the components themselves are deployed as containers, restricting write access to their namespace they are running in as well as communication to the containers themselves. Synonymous to section 3.2.1, isolating through separate clusters per team or application can restrict the damage potential through separate master components.

3.2.5 V05 - Image poisoning and baiting

An attacker may try and lead a cluster user to run containers from a container image that is either controlled by the attacker or insecure.[4] This could be done as an untargeted attack by offering images on popular public image repositories like Docker Hub or providing examples of similarly bad Dockerfiles in tutorials or support forums. Since building new containers from scratch is a lot of work, container images for popular applications are often sought to either build upon or use out-of-the-box. Applications for which no official container image is maintained are an especially common niche for such attacks. In contrast to this, an attacker could also try a more targeted attack by uploading bad images to the (possibly private) repository used by their target. Even previously trusted images may become insecure

1 Aut19n, section 'What is a Pod Security Policy?'
2 RH18, chapter 'Running etcd Safely'.
3 Aut19e, section 'Master Components'.
4 SMS11, p. 13 to 14.

over time as new vulnerabilities are publicized, thus becoming "stale".[1] It should be noted that providing Dockerfiles instead of images may be easier to identify as malicious, but when a user builds an image from it anyway, the build process will commonly run under a root user.[2]

In order to mitigate all of the above, only safe images should be used to run containers within a cluster. Although this sounds simple in theory, it may prove to be far more complex in practice. One would first have to define what constitutes a safe image in their context before implementing any measures ensuring their usage. Docker Hub introduced measures that help identifying more trustworthy images by labeling them as Verified Publisher Images or Certified Images respectively[3]. In order to restrict the amount of possible images used, a private registry could be used and the cluster restricted to only using images from that registry. User may then be allowed to build new images based on those in the registry and upload those to it. The damage potential of images may be limited by restricting the capabilities a container may gain through the Pod Security Policies introduced in section 3.2.3. Images could also go through a (either manual or automated) vetting process before being admitted to a registry. Images in a registry may be reassessed on a regular basis, too, so images with known vulnerabilities can be identified and removed. Multiple tools that provide automated image scans exist and can be found online[4], some of which provide integration into k8s clusters or cloud platforms.

3.2.6 V06 - Configuration poisoning and baiting

An attacker may try and lead a cluster user or even administrator to configure the cluster in such a way that the attacker gains control, the cluster is left insecure, or both.[5] This vector has a lot in common with the one described in section 3.2.5. A bad configuration might include bad containers, but does not have to, which is why we kept this as a separate vector. Attacks like this are far easier to do untargeted by offering tutorials and posts in support forums, but might also be conducted in a targeted way, similar to spear phishing attacks. The number of possibilities for specific bad configurations is near endless. Easy and effective misconfigurations may be achieved by swapping out popular image names with similar ones controlled by the attacker (i.e. 'nodejs' instead of 'node'), preconfiguring weak or attacker known default passwords, simply omitting or misconfiguring security measures so they do not work effectively (i.e. defining Network Policies, but not applying them) or making internal interfaces available to the public network.

1 SMS11, p. 14.
2 While k8s currently doesn't provide image building capabilities out-of-the-box, many k8s certified solutions implement this functionality.
3 MI18, sections 'Verified Publisher Images and Plugins' and 'Certified Images and Plugins'.
4 RH19, section 'Securing your container images'.
5 SMS11, p. 13 to 14.

As there are many potential ways for misconfiguration, there are also many measures to mitigate them. Raising user and administrator awareness for the risk of using a configuration supplied by outsiders is recommended. Continuously training them to provide the knowledge needed to thoroughly understand and assess these configurations may help in mitigating this problem, too. Free and helpful tools in assessing configurations are kubesec.io, kube-bench[1] and kubeaudit[2]. Defining guidelines and best practices for configurations could be done and enforced by regular manual reviews or automated checks through tools like the Open Policy Agent admission controls[3] or k8Guard[4]. As bad configurations may include bad containers, the measures introduced in section 3.2.5 are applicable, too.

3.2.7 V07 - Lateral movement through cluster

Once an attacker gains access to a container, he may try to access more lucrative information, applications or cluster components. Ways to achieve this could be sniffing network traffic or scanning the network for other containers, hosts, services, apis or similar interfaces.

Applicable security measures include network isolation through Network Policies[5] and mutual TLS authentication for network communication through tools like Istio[6].

3.2.8 V08 - Container breakout

A popular phrase about container security is that "[c]ontainers do not contain"[7]. Once an attacker is able to execute commands in a container within the cluster, he may try influence components outside its isolation confinements. If an attacker successfully gains access to the underlying system, they might gain visibility into or control over any container running on it, including those of other applications. Additionally, they could gain access to internal cluster components like the kubelet[8]. Ways to achieve this include the invocation of syscalls, accessing mounted parts of the host file system and requesting additional capabilities. More ways to achieve this are being discussed at the time of this writing as the OWASP Docker Top 10 are worked on[9] and openly demonstrated[10].

1 https://github.com/aquasecurity/kube-bench
2 https://github.com/Shopify/kubeaudit
3 con19, section 'Wrap Up'.
4 https://k8guard.github.io/
5 Aut19o, section 'The NetworkPolicy Resource'.
6 Aut19p, section 'Mutual TLS authentication'.
7 Wal14, first headline.
8 refer to section 2.4.1 for additional details
9 Fou19f, dialogue by GitHub users 'drwetter', 'gramsimamsi' and 'wurstbrot'.
10CB19.

This can again be mitigated by restricting the capabilities and host access a given container is provided through the Pod Security Policies mentioned in section 3.2.3. "Sandboxed" containers may be leveraged here, too, which provide more isolation between a container and its host at the cost of performance. Popular implementations include gVisor[1] and Kata Containers[2].

3.2.9 V09 - Image cache compromise

If the container image cached by a container runtime on the underlying host system is swapped out by an attacker, another container controlled by the attacker might be started.

In order to mitigate this, container runtimes may support capabilities to check for the integrity of a container image. An example of such features is Docker Content Trust, which enables integrity verification at runtime[3].

3.2.10 V10 - Container modification at runtime

Instead of starting bad containers as discussed in section 3.2.13, an attacker may try to modify it in order to suit their needs and support further attacks. This may be done by downloading and running additional binaries, establishing connections to Command and Control Servers or manipulating the software and files in place.

Measures against this build include limiting the capabilities a container has as discussed in section 3.2.3. Limiting network connections to cluster-external sources may mitigate ways to download additional software or exfiltrate data. This could be done via Egress Network Policies, which were introduced in section 3.2.1 Monitoring tools could be used to detect this, of which several commercial products are offered and often integrated into (cloud) provider platforms[4].

3.2.11 V11 - Resource hoarding (sabotage)

An attacker with the ability to use resources or manipulate configuration may misuse or misconfigure them in order to disrupt the availability of specific applications or the cluster as a whole in DOS attacks. This may be done in a myriad of ways. Exhausting the available resources through fork bombs[5] or intensive use of memory, RAM, network bandwidth, processing power or available network ports

1 https://github.com/google/gvisor
2 https://katacontainers.io/
3 Inc19l, section 'About Docker Content Trust (DCT)'.
4 TK18, slide 40 to 41.
5 RH18, chapter 'Fork Bombs and Resource-Based Attacks'.

may be a straightforward technique. They could also delete data and binaries, i.e. k8s objects and node programs and operating system components, wherever they have access to. If an attacker tries to achieve disruption for a long period of time, they could misconfigure the setup in a way that is either difficult to debug or reverse.

Measures against this include dedicated master nodes, which are a common practice anyway, in order to keep entities without administrator access from affecting the master nodes an master components running on them. Synonymous to section 3.2.1, isolating applications or teams through different namespaces or even clusters would reduce the 'blast radius' of a successful attack. Once this isolation is in place, resource quotas[1] could be set in place to constrain the total resource consumption of a namespace. Tools to monitor usage[2] could be set in place, as well as alerting mechanisms for unusually high resource consumption. Misconfiguration could be mitigated by the same logging and alerting measures introduced in section 3.2.1. Deployment management tools like Helm could also be used, which simplify persisting the k8s object states in a version-control system.

3.2.12 V12 - Resource misuse (cryptojacking)

Instead of sabotaging the cluster and/or applications, an attacker may use resources they can allocate to achieve monetary gain. This is often done by mining cryptocurrencies. The most commonly cited example for this is the incident of electric car manufacturer Tesla, where attackers used the control gained over their cloud infrastructure to run mining software in order to gain cryptocurrency with resources paid for by the company[3].

The measures against this include those introduced in section 3.2.11. As this vector might be more difficult to detect if done well, a focus on detection may be of use. More advanced measures are presented by RedLock, the company which uncovered the Tesla incident, on their blog[4].

3.2.13 V13 - Adding rogue containers

In contrast to sections 3.2.5 and 3.2.10, an attacker or malicious user capable of starting containers in the cluster may simply start their own malicious one. Since they could try to configure it in any way they want, this could help them in further attacks, leveraging vectors like those described in sections 3.2.7 or 3.2.8.

1 Aut19q, section 'Viewing and Setting Quotas'.
2 Aut19r, first paragraph.
3 Tea18, section 'The Latest Victim: Tesla'.
4 Tea18, section 'Preventing Such Compromises'.

The mitigations of section 3.2.10 mostly apply here, except for those against downloading additional binaries at runtime. An attacker could simply supply these within a Dockerfile or container image so they would already be installed. Against this, the detection and mitigation measures against bad images described in section 3.2.5 are applicable.

3.2.14 V14 - Adding rogue nodes

An attacker could try and add a host as a cluster node which is controlled by them. If successful, containers will be scheduled on the new node, allowing an attacker access to these containers in order to read, exfiltrate or manipulate data. They would also have control over the kubelet on this node, as described in section 3.2.8. In a sufficient time period, the chance of any container in the cluster running on the node at least once increases. This process could be sped up by manipulating the reporting data sent by the kubelet to the kube-scheduler, faking a lot of unused resources on it so more containers are scheduled.

In order to mitigate this, nodes would have to authenticate themselves to the apiserver and permissions to add new nodes should be limited to the highest possible user accounts like cluster administrators.

3.2.15 V15 - Leveraging bad user practice

An attacker may leverage bad user practice to gain access to or permissions within a cluster. This vector comprises user practices outside of the cluster that lead to risks within it. Examples include phishing for user accounts, gathering keys/tokens published to public code repositories, gathering passwords published in credential leaks or dumps, scouting specific software or container images used as well as gathering logs published with information valuable to an attacker. This could be done in a targeted way (i.e. specific Open Source Intelligence gathering or sending spearphishing emails) or untargeted by searching large repositories like Github for information formatted in a specific way. A practical example would include looking for published '.kube\config' files, where authentication information and kube-apiserver addresses are saved to[1]. Gaining access through tokens in public logs has been demonstrated, too[2].

3.2.16 V16 - Leveraging bad infrastructure

Even if the cluster itself is secured properly, an attacker may gain access through the underlying infrastructure if it isn't. This includes the configuration of cloud infrastructure as well as on-premise

1 Aut19s, section 'Define clusters, users, and contexts'.
2 EdO19, section 'Results and notable findings'.

configurations. Servers may open insecure or unnecessary ports towards the public, as well as interfaces that allow someone any sort of control, i.e. allowing public access to the Docker remote API[1]. Internally, an attacker may access data belonging to other applications by leveraging side channel attacks like Spectre and Meltdown[2].

Measures against this include well known and documented security measures and best practices for traditional server infrastructure, including the full range of CIS benchmarks available[3]. Keeping the OS and programs of the underlying nodes minimal by only installing the necessary tools, functions and binaries may aid by reducing potential vulnerabilities on top of reducing the resource overhead. Establishing a 'bastion host'[4] as single point of remote entry to the cluster may be advisable if such functionality is required. Measures can also be found in the OWASP CSVS[5], CIS Docker Benchmark[6], the CIS Kubernetes Benchmark[7] and the NIST publication[8]. Cloud configuration may be assessed and/or enforced with automated tools like ScoutSuite[9], cloud-custodian[10] or environment-specific tools like azurite[11].

3.2.17 V17 - Leveraging bad patch management

An attacker may leverage publicly known vulnerabilities in software that is not up to date on any component used in the environment. This may even include systems assumed to be responsible for by other parties. Even in PaaS environments like AKS, servers have to be restarted by the cloud customer in order for some security updates to take effect. An example for this can be seen in an email sent to an Azure cloud account holder, a screenshot of which is provided in figure A.2.

Mitigating this is straightforward in theory - apply security updates, fixes and intermediary workarounds whenever they become available. In practice this proves to be an ongoing real-world problem, as the WannaCry epidemic very publicly demonstrated[12]. Any and all unpatched components could pose a risk. Thus, responsibilities and guidelines should be defined for when and what to patch. Fallback plans accounting for vacation time and sick days should be implemented, too. The employees responsible for updating should set up or subscribe to their relevant vulnerability notification feeds, i.e. alerts

1 SN19, section 'Publicly Accessible Docker Hosts'.
2 Tec19, section 'Which cloud providers are affected by Meltdown?'
3 Int19c, refer to the list presented.
4 Sco19, section 'What is a bastion host, and do I need one?'
5 RF19, section 'Infrastructure'.
6 Int19a, chapters 1 through 3 and 5.
7 Int19b, chapters 'Worker Node Security Configuration' and 'Configuration Files'.
8 SMS11, chapters 3.5, 4.5 and 4.6.
9 https://github.com/nccgroup/ScoutSuite
10 https://github.com/cloud-custodian/cloud-custodian
11 https://github.com/mwrlabs/Azurite
12 Ked19, p. 2.

for new vulnerabilities[1]. Servers should follow conventional patch management processes, as these are similar to conventional system infrastructure. Kubernetes distributions should be kept up to date with updates provided by the k8s distributor. Of note is the short time of guaranteed support by the k8s reference implementation. The Kubernetes project maintains the most recent three minor releases with a new minor version planned approximately every three months[2]. This would result in a cluster upgrade to be required every nine months in order to stay supported. Other distributions may have longer time periods of guaranteed support, i.e. OCP[3]. Applying updates to containers by updating container images should be done regularly. This includes both incrementing the version numbers of their base images to a version without currently known vulnerabilities as well as updating binaries directly downloaded during the build process. As an 'update' command run by the package manager during the build process works dynamically, security updates to those packages are included any time an image is built again. In consequence, images should be rebuilt periodically. Using the image version tag 'latest' is not recommended, as this could automatically propagate bad images pushed to the image repository through to production environments[4]. Automated scanning may be implemented by open source tools like clair[5]. Limiting the probability for new vulnerabilities by reducing the software installed as described in section 3.2.16 may help, too.

1 Dio19, first paragraph.
2 Aut19j, section 'Supported versions'.
3 Inc19m, section 'OpenShift Container Platform v3'.
4 RH18, p. 40.
5 https://github.com/coreos/clair

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, p. 4.
3 Gee17, starting at 3:05.

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.

The formula for estimating risk values of the vectors identified in section 3.2 was developed with the three scenarios in mind that were introduced in section 1.3.

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 wast feasible or added much value to the goals. TODO: Explain process, what was looked at and why it wast 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 properly. 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 probability 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 detection; 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 individualized 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, values above that are defined as high with the exception of 10, which is critical.

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/origin/releases/tag/v3.11.0) 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/CHANGELOG-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.

Vector ID	Vector	OCP Risk	AKS Risk
V01	Reconnaissance through interface components	**3**	**3**
V02	Reading confidentials through interface components	**6**	**7**
V03	Configuration manipulation through interface components	**7**	**8**
V04	Compromise internal master components	**5**	**6**
V05	Image poisoning and baiting	**7**	**7**
V06	Configuration poisoning and baiting	**10**	**10**
V07	Lateral movement through cluster	**7**	**7**
V08	Container breakout	**5**	**7**
V09	Image cache compromise	**3**	**3**
V10	Container modification at runtime	**5**	**5**
V11	Resource hoarding (sabotage)	**8**	**5**
V12	Resource misuse (cryptojacking)	**5**	**8**
V13	Adding rogue containers	**5**	**6**
V14	Adding rogue nodes	**6**	**6**
V15	Leveraging bad user practice	**6**	**6**
V16	Leveraging bad infrastructure	**9**	**9**
V17	Leveraging bad patch management	**10**	**10**

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

Vector ID	Vector	Vantage Point	RAL	Detectability	Exploitability	Probability	Impact	Resulting risk
V01	Reconnaissance through interface components	3	3	3	2	2.75	1	**3**
V02	Reading confidentials through interface components	3	3	3	3	3	2	**6**
V03	Configuration manipulation through interface components	3	1	3	2	2.25	3	**7**
V04	Compromise internal master components	3	1	3	0	1.75	3	**5**
V05	Image poisoning and baiting	4	4	3	2	3.25	2	**7**
V06	Configuration poisoning and baiting	4	4	3	2	3.25	3	**10**
V07	Lateral movement through cluster	2	2	3	2	2.25	3	**7**
V08	Container breakout	2	2	3	0	1.75	3	**5**
V09	Image cache compromise	1	1	2	2	1.5	2	**3**
V10	Container modification at runtime	2	2	3	2	2.25	2	**5**
V11	Resource hoarding (sabotage)	3	2	3	2	2.5	3	**8**
V12	Resource misuse (cryptojacking)	3	2	3	2	2.5	2	**5**
V13	Adding rogue containers	3	2	3	2	2.5	2	**5**
V14	Adding rogue nodes	3	1	2	2	2	3	**6**
V15	Leveraging bad user practice	3	4	2	2	2.75	2	**6**
V16	Leveraging bad infrastructure	4	4	2	2	3	3	**9**
V17	Leveraging bad patch management	4	4	3	2	3.25	3	**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	Reconnaissance through interface components	4	3	3	2	3	1	**3**
V02	Reading confidentials through interface components	4	3	3	3	3.25	2	**7**
V03	Configuration manipulation through interface components	4	1	3	2	2.5	3	**8**
V04	Compromise internal master components	4	1	3	0	2	3	**6**
V05	Image poisoning and baiting	4	4	3	2	3.25	2	**7**
V06	Configuration poisoning and baiting	4	4	3	2	3.25	3	**10**
V07	Lateral movement through cluster	2	2	3	2	2.25	3	**7**
V08	Container breakout	2	2	3	2	2.25	3	**7**
V09	Image cache compromise	1	1	2	2	1.5	2	**3**
V10	Container modification at runtime	2	2	3	2	2.25	2	**5**
V11	Resource hoarding (sabotage)	3	2	3	2	2.5	2	**5**
V12	Resource misuse (cryptojacking)	3	2	3	2	2.5	3	**8**
V13	Adding rogue containers	4	2	3	2	2.75	2	**6**
V14	Adding rogue nodes	4	1	1	2	2	3	**6**
V15	Leveraging bad user practice	4	4	2	2	3	2	**6**
V16	Leveraging bad infrastructure	4	4	2	2	3	3	**9**
V17	Leveraging bad patch management	4	4	3	2	3.25	3	**10**

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

4.2.1 V01 - Reconnaissance through interface components

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 - Reading confidentials through interface components

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 - Configuration manipulation through interface components

In addition 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 master components

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 cant 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 - Image poisoning and baiting

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 fulfill 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 publicly 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 - Configuration poisoning and baiting

Similar to V07, this can be done either untargeted by spraying to tutorials / help forums or targeted, similar to spear phishing. 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 - Lateral movement through cluster

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 publicly 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

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 favor 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 - Image cache compromise

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 - Container modification at runtime

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 - Resource hoarding (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 - Resource misuse (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 - Adding rogue containers

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 - Adding rogue nodes

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 publicly 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 - Leveraging bad user practice

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 - Leveraging bad infrastructure

The underlying nodes could allow an attacker easy entry, even if the containers themselves are hardened. This includes Side-Channel attacks like Spectre and 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 publicly on the internet for anyone to access and indexed by shodan. Bye bye cluster. (Too many scenarios to hypothesize here, Ill just point the finger at conventional server and infra hardening standards and guidelines) -> Well known, still needs some technical skill to find vulns and exploit them

Surprisingly, 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 - Leveraging bad patch management

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 through cluster

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

TODO: show demo on OCP and AKS, or just one and put the other in appendix?
TODO: show getting in and damage, refer to ppt for commands, console I/O and screenshots

5.2.2 Selecting security measures

TODO: use multitenant networkplugin (OCP) and network policies (k8s) TODO: show implementing measures through commands in ppt

2 Fou19a.

5.2.3 Demonstration with implemented security measures

TODO show remediation through demo of unsuccessful attack

5.2.4 Risk reassessment

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

TODO: show demo on OCP and AKS, or just one and put the other in appendix?
TODO: show getting in and damage, refer to ppt for commands, console I/O and screenshots

5.3.2 Selecting security measures

TODO dont allow privileged containers (scc in OCP, podsecuritypolicies in AKS) TODO: show implementing measures through commands in ppt

5.3.3 Demonstration with implemented security measures

TODO show remediation through demo of unsuccessful attack

5.3.4 Risk reassessment revisited

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

5.4 Continuing the risk management process

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 Comparing on-premise and public cloud

TODO reference table 4.1
TODO explain problem of generically comparing, without going into specific solutions -> just comparing OCP vs. AKS. 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 use cases 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 data centers make on-prem easier).

6.0.2 Summary

TODO summary of thesis process - huge scope, could have been better defined beforehand. risk estimate formula still inst 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.

A Appendix

A.1 AKS versions available on May 3rd, 2019

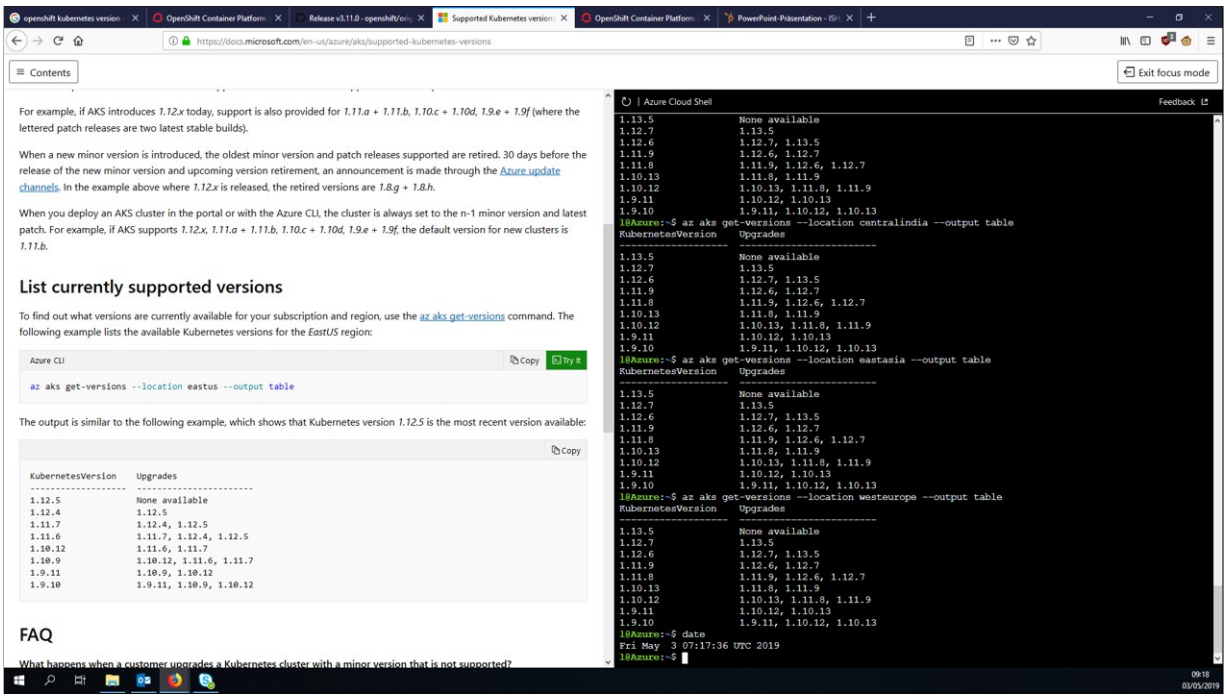
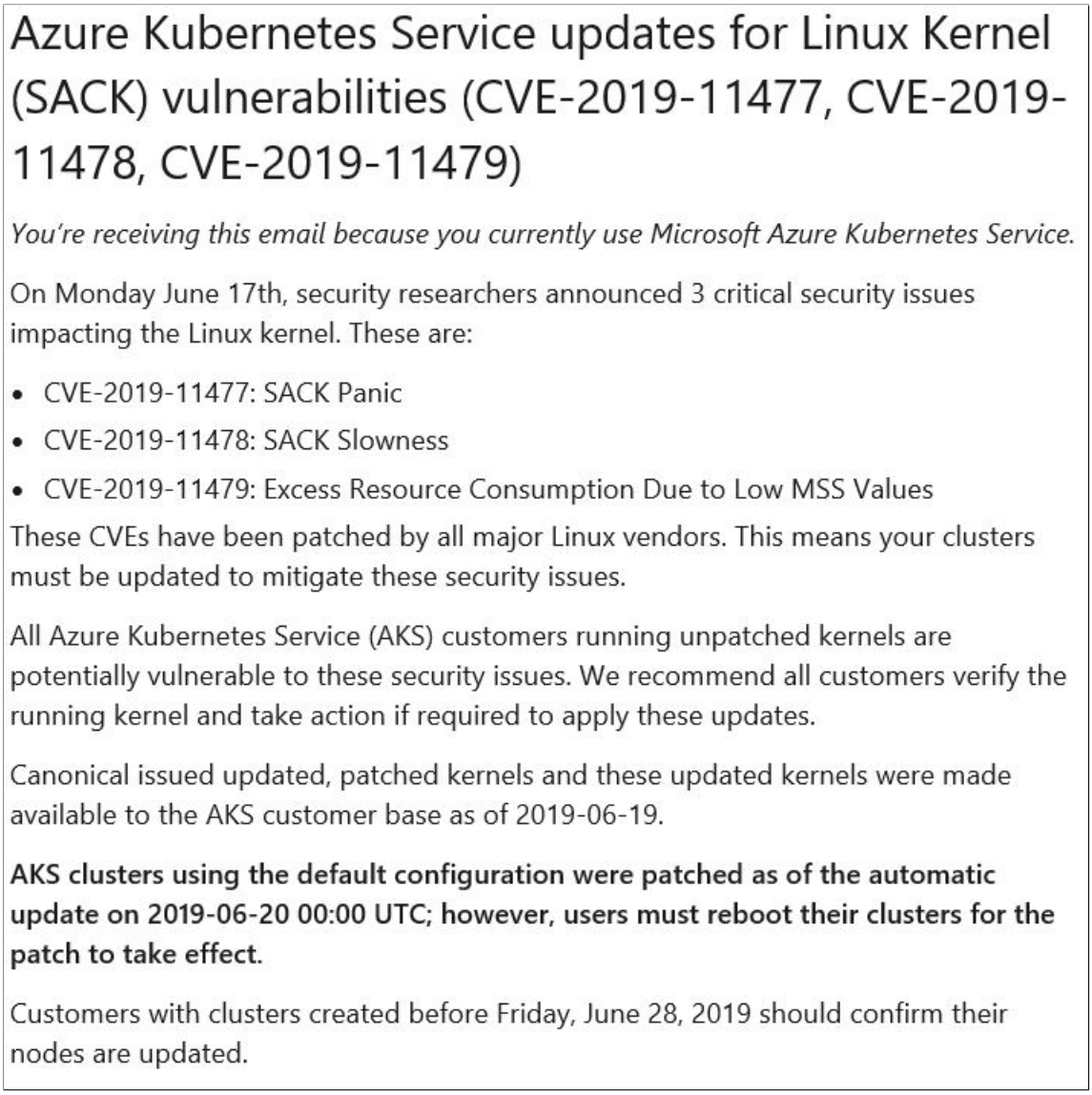


Figure A.1 List of k8s versions available in AKS during the practical part of the thesis on May 3rd, 2019. Screenshot taken by Lukas Grams.

A.2 Security advisory email from Microsoft



Azure Kubernetes Service updates for Linux Kernel (SACK) vulnerabilities (CVE-2019-11477, CVE-2019-11478, CVE-2019-11479)

You're receiving this email because you currently use Microsoft Azure Kubernetes Service.

On Monday June 17th, security researchers announced 3 critical security issues impacting the Linux kernel. These are:

- CVE-2019-11477: SACK Panic
- CVE-2019-11478: SACK Slowness
- CVE-2019-11479: Excess Resource Consumption Due to Low MSS Values

These CVEs have been patched by all major Linux vendors. This means your clusters must be updated to mitigate these security issues.

All Azure Kubernetes Service (AKS) customers running unpatched kernels are potentially vulnerable to these security issues. We recommend all customers verify the running kernel and take action if required to apply these updates.

Canonical issued updated, patched kernels and these updated kernels were made available to the AKS customer base as of 2019-06-19.

AKS clusters using the default configuration were patched as of the automatic update on 2019-06-20 00:00 UTC; however, users must reboot their clusters for the patch to take effect.

Customers with clusters created before Friday, June 28, 2019 should confirm their nodes are updated.

Figure A.2 An email from July 15th, 2019, advising users to reboot their AKS clusters in order to apply security patches. Screenshot taken by Lukas Grams.

A.3 AKS cluster default confs?

TODO? <- maybe add screenshots about aks default config situation, leading to rebuild needed to get netpols?

Bibliography

Books

[RH18] Liz Rice and Michael Hausenblas. *Kubernetes Security - How to Build and Operate Applications Securely in Kubernetes*. O'Reilly Media, Inc., 2018. URL: https://info.aquasec.com/kubernetes-security (cit. on pp. 3, 18–20, 23, 27).

Online text-based sources

[Aut19a] The Kubernetes Authors. *Understanding Kubernetes Objects*. 2019. URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/ (visited on 07/16/2019) (cit. on pp. ix, 9, 11, 14).

[Inc18a] Red Hat Inc. *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 07/16/2019) (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 07/16/2019) (cit. on pp. 2, 3).

[Fou19b] OWASP Foundation. *Threat Modeling Cheat Sheet*. 2019. URL: https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Threat_Modeling_Cheat_Sheet.md (visited on 07/16/2019) (cit. on p. 2).

[Irw14] Stephen Irwin. *Creating a Threat Profile for Your Organization*. 2014. URL: https://www.sans.org/reading-room/whitepapers/threats/creating-threat-profile-organization-35492 (visited on 07/16/2019) (cit. on p. 2).

[SMS11] Murugiah Souppaya, John Morello, and Karen Scarfone. *NIST Special Publication 800-190 Application Container Security Guide*. 2011. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-190.pdf (visited on 07/16/2019) (cit. on pp. 3, 20, 21, 26).

[Int19a] Center for Internet Security. *CIS Docker Community Edition Benchmark*. 2019. URL: https://www.cisecurity.org/benchmark/docker/ (visited on 07/16/2019) (cit. on pp. 3, 26).

[Int19b] Center for Internet Security. *CIS Kubernetes Benchmark*. 2019. URL: https://www.cisecurity.org/benchmark/kubernetes/ (visited on 07/16/2019) (cit. on pp. 3, 26).

[RF19] RedGuard and OWASP Foundation. *Container Security Verification Standard*. 2019. URL: https://github.com/OWASP/Container-Security-Verification-Standard (visited on 07/16/2019) (cit. on pp. 3, 26).

[Fou19c] OWASP Foundation. *Docker Security*. 2019. URL: https://github.com/OWASP/Docker-Security (visited on 07/16/2019) (cit. on p. 3).

[Aut19b] The Kubernetes Authors. *Overview of Cloud Native Security*. 2019. URL: https://kubernetes.io/docs/concepts/security/overview/ (visited on 07/16/2019) (cit. on p. 3).

[Inc19a] Red Hat Inc. *OpenShift Container Platform 3.11 Container Security Guide*. 2019. URL: https://access.redhat.com/documentation/en-us/openshift_container_platform/3.11/pdf/container_security_guide/OpenShift_Container_Platform-3.11-Container_Security_Guide-en-US.pdf (visited on 07/16/2019) (cit. on p. 3).

[Cor18] Microsoft Corporation. *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 07/16/2019) (cit. on p. 3).

[ST11] National Institute of Standards and Technology. *SP 800-145 The NIST Definition of Cloud Computing*. 2011. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf (visited on 07/16/2019) (cit. on pp. 5, 6).

[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 07/16/2019) (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 07/16/2019) (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 (visited on 07/16/2019) (cit. on p. 7).

[Inc19b] Docker Inc. *What is a Container?* 2019. URL: https://www.docker.com/resources/what-container (visited on 07/16/2019) (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 07/16/2019) (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 07/16/2019) (cit. on p. 7).

[Aut19c] The Kubernetes Authors. *What is Kubernetes*. 2019. URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (visited on 07/16/2019) (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 07/16/2019) (cit. on pp. 8, 15).

[Inc19c] Docker Inc. *About Docker Engine*. 2019. URL: https://docs.docker.com/engine/ (visited on 07/16/2019) (cit. on p. 8).

[Inc19d] Red Hat Inc. *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 (visited on 07/16/2019) (cit. on p. 8).

[Inc18b] Red Hat Inc. *Containers and Images*. 2018. URL: https://docs.openshift.com/enterprise/3.0/architecture/core_concepts/containers_and_images.html (visited on 07/16/2019) (cit. on p. 8).

[Inc18c] Docker Inc. *Dockerfile Reference*. 2018. URL: https://docs.docker.com/engine/reference/builder/ (visited on 07/16/2019) (cit. on p. 8).

[Fou19d] The Linux Foundation. *Open Container Initiative*. 2019. URL: https://www.opencontainers.org/ (visited on 07/16/2019) (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 07/16/2019) (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 07/16/2019) (cit. on p. 9).

[Aut19d] The Kubernetes Authors. *Container Runtimes Part 3: High-Level Runtimes*. 2019. URL: https://github.com/kubernetes/cri-api/ (visited on 07/16/2019) (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 07/16/2019) (cit. on pp. 10, 15).

[Aut19e] The Kubernetes Authors. *Kubernetes Components*. 2019. URL: https://kubernetes.io/docs/concepts/overview/components/ (visited on 07/16/2019) (cit. on pp. 10, 20).

[Aut19f] The Kubernetes Authors. *Pod Overview*. 2019. URL: https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/ (visited on 07/16/2019) (cit. on p. 11).

[BC18] Matt Butcher and Karen Chu. *Phippy goes to the Zoo*. 2018. URL: https://www.cncf.io/wp-content/uploads/2018/12/Phippy-Goes-To-The-Zoo.pdf (visited on 07/16/2019) (cit. on p. 11).

[Aut19g] The Kubernetes Authors. *ReplicaSet*. 2019. URL: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/ (visited on 07/16/2019) (cit. on p. 12).

[Aut19h] The Kubernetes Authors. *Picking the Right Solution*. 2019. URL: https://v1-13.docs.kubernetes.io/docs/setup/pick-right-solution/# (visited on 07/16/2019) (cit. on p. 14).

[Aut19i] The Kubernetes Authors. *The Kubernetes API*. 2019. URL: https://kubernetes.io/docs/concepts/overview/kubernetes-api/ (visited on 07/16/2019) (cit. on p. 14).

[Inc19e] Red Hat Inc. *OpenShift plans and pricing*. 2019. URL: https://www.openshift.com/products/pricing/ (visited on 07/16/2019) (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 07/16/2019) (cit. on p. 15).

[Inc19f] Red Hat Inc. *The Origin Community Distribution of Kubernetes that powers Red Hat OpenShift*. 2019. URL: https://www.okd.io/ (visited on 07/16/2019) (cit. on p. 15).

[Inc19g] Red Hat Inc. *OpenShift Container Platform 3.x Tested Integrations*. 2019. URL: https://access.redhat.com/articles/2176281 (visited on 07/16/2019) (cit. on p. 15).

[Inc19h] Red Hat Inc. *Get Started with the CLI*. 2019. URL: https://docs.openshift.com/container-platform/3.11/cli_reference/get_started_cli.html (visited on 07/16/2019) (cit. on p. 15).

[Inc19i] Red Hat Inc. *Managing Projects*. 2019. URL: https://docs.openshift.com/container-platform/3.11/admin_guide/managing_projects.html (visited on 07/16/2019) (cit. on p. 15).

[Inc19j] Red Hat Inc. *Using the CRI-O Container Engine*. 2019. URL: https://docs.openshift.com/container-platform/3.11/crio/crio_runtime.html (visited on 07/16/2019) (cit. on p. 15).

[Cor19c] Microsoft Corporation. *Integrate Azure Active Directory with Azure Kubernetes Service*. 2019. URL: https://docs.microsoft.com/en-us/azure/aks/azure-ad-integration (visited on 07/16/2019) (cit. on p. 15).

[Aut19j] The Kubernetes Authors. *Kubernetes version and version skew support policy*. 2019. URL: https://kubernetes.io/docs/setup/release/version-skew-policy/ (visited on 07/16/2019) (cit. on pp. 17, 27).

[Aut19k] The Kubernetes Authors. *Web UI (Dashboard)*. 2019. URL: https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/ (visited on 07/17/2019) (cit. on p. 18).

[Inc19k] Red Hat Inc. *Create and Build an Image Using the Web Console*. 2019. URL: https://docs.openshift.com/container-platform/3.11/getting_started/developers_console.html (visited on 07/17/2019) (cit. on p. 18).

[Cor19d] Microsoft Corporation. *Quickstart: Deploy an Azure Kubernetes Service (AKS) cluster using the Azure portal*. 2019. URL: https://docs.microsoft.com/en-us/azure/aks/kubernetes-walkthrough-portal (visited on 07/17/2019) (cit. on p. 18).

[Gom18] Paulo Gomez. *How a naughty docker image on AKS could give an attacker access to your Azure Subscription?* 2018. URL: https://itnext.io/how-a-naughty-

docker-image-on-aks-could-give-an-attacker-access-to-your-azure-subscription-6d05b92bf811 (visited on 07/17/2019) (cit. on p. 18).

[Aut19l] The Kubernetes Authors. *Network Policies*. 2019. URL: https://kubernetes.io/docs/concepts/services-networking/network-policies/ (visited on 07/17/2019) (cit. on p. 19).

[Aut19m] The Kubernetes Authors. *Auditing*. 2019. URL: https://kubernetes.io/docs/tasks/debug-application-cluster/audit/ (visited on 07/17/2019) (cit. on p. 19).

[San17] Jorge Salamero Sanz. *Monitoring Kubernetes (part 2): Best practices for alerting on Kubernetes*. 2017. URL: https://sysdig.com/blog/alerting-kubernetes/ (visited on 07/17/2019) (cit. on p. 19).

[Aut19n] The Kubernetes Authors. *Pod Security Policies*. 2019. URL: https://kubernetes.io/docs/concepts/policy/pod-security-policy/ (visited on 07/17/2019) (cit. on p. 20).

[MI18] Jeff Morgan and Docker Inc. *Introducing the New Docker Hub*. 2018. URL: https://blog.docker.com/2018/12/the-new-docker-hub/ (visited on 07/17/2019) (cit. on p. 21).

[RH19] Liz Rice and Michael Hausenblas. *Kubernetes Security*. 2019. URL: https://kubernetes-security.info/ (visited on 07/17/2019) (cit. on p. 21).

[con19] Open Policy Agent contributors. *Kubernetes Admission Control*. 2019. URL: https://kubernetes-security.info/ (visited on 07/18/2019) (cit. on p. 22).

[Aut19o] The Kubernetes Authors. *Network Policies*. 2019. URL: https://kubernetes.io/docs/concepts/services-networking/network-policies/ (visited on 07/18/2019) (cit. on p. 22).

[Aut19p] Istio Authors. *What is Istio?* 2019. URL: https://istio.io/docs/concepts/what-is-istio/ (visited on 07/18/2019) (cit. on p. 22).

[Wal14] Daniel J Walsh. *Are Docker containers really secure?* 2014. URL: https://opensource.com/business/14/7/docker-security-selinux (visited on 07/18/2019) (cit. on p. 22).

[Fou19f] OWASP Foundation. *Addition to the threat mindmap might be needed*. 2019. URL: https://github.com/OWASP/Docker-Security/issues/9 (visited on 07/18/2019) (cit. on p. 22).

[CB19] Dominik Czarnota and Trail of Bits. *Understanding Docker container escapes*. 2019. URL: https://blog.trailofbits.com/2019/07/19/understanding-docker-container-escapes/ (visited on 07/29/2019) (cit. on p. 22).

[Inc19l] Docker Inc. *Content trust in Docker*. 2019. URL: https://docs.docker.com/engine/security/trust/content_trust/ (visited on 07/18/2019) (cit. on p. 23).

[TK18] Jen Tong and Maya Kaczorowski. *Kubernetes Runtime Security*. 2018. URL: https://static.sched.com/hosted_files/kccnceu18/72/20180524%20-%20KubeCon%20EU%20-%20Kubernetes%20Runtime%20Security.pdf (visited on 07/18/2019) (cit. on p. 23).

[Aut19q] The Kubernetes Authors. *Resource Quotas*. 2019. URL: https://kubernetes.io/docs/concepts/policy/resource-quotas/ (visited on 07/19/2019) (cit. on p. 24).

[Aut19r] The Kubernetes Authors. *Tools for Monitoring Resources*. 2019. URL: https://kubernetes.io/docs/tasks/debug-application-cluster/resource-usage-monitoring/ (visited on 07/19/2019) (cit. on p. 24).

[Tea18] RedLock CSI Team. *Lessons from the Cryptojacking Attack at Tesla*. 2018. URL: https://redlock.io/blog/cryptojacking-tesla (visited on 07/19/2019) (cit. on p. 24).

[Aut19s] The Kubernetes Authors. *Configure Access to Multiple Clusters*. 2019. URL: https://kubernetes.io/docs/tasks/access-application-cluster/

configure-access-multiple-clusters/ (visited on 07/19/2019) (cit. on p. 25).

[EdO19] EdOverflow. *"CI Knew There Would Be Bugs Here" — Exploring Continuous Integration Services as a Bug Bounty Hunter*. 2019. URL: https://edoverflow.com/2019/ci-knew-there-would-be-bugs-here/ (visited on 07/19/2019) (cit. on p. 25).

[SN19] Vitaly Simonovich and Ori Nakar. *Hundreds of Vulnerable Docker Hosts Exploited by Cryptocurrency Miners*. 2019. URL: https://www.imperva.com/blog/hundreds-of-vulnerable-docker-hosts-exploited-by-cryptocurrency-miners/ (visited on 07/21/2019) (cit. on p. 26).

[Tec19] Graz University of Technology. *Meltdown and Spectre*. 2019. URL: https://meltdownattack.com/ (visited on 07/21/2019) (cit. on p. 26).

[Int19c] Center for Internet Security. *CIS Benchmarks*. 2019. URL: https://www.cisecurity.org/cis-benchmarks/ (visited on 07/21/2019) (cit. on p. 26).

[Sco19] Stuart Scott. *AWS Security: Bastion Host, NAT instances and VPC Peering*. 2019. URL: https://cloudacademy.com/blog/aws-bastion-host-nat-instances-vpc-peering-security/ (visited on 07/21/2019) (cit. on p. 26).

[Ked19] Mark Kedgley. *The Problem with Running Outdated Software*. 2019. URL: https://www.newnettechnologies.com/whitepaper/Outdated-Software-Whitepaper.pdf (visited on 07/21/2019) (cit. on p. 26).

[Dio19] Yuri Diogenes. *Using Azure Monitor to send an Email Notification for Azure Security Center Alerts*. 2019. URL: https://blogs.technet.microsoft.com/yuridiogenes/2018/08/01/using-azure-monitor-to-send-an-email-notification-for-azure-security-center-alerts/ (visited on 07/21/2019) (cit. on p. 27).

[Inc19m] Red Hat Inc. *Red Hat OpenShift Container Platform Life Cycle Policy*. 2019. URL: https://access.redhat.com/support/policy/updates/openshift (visited on 07/21/2019) (cit. on p. 27).

[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 07/16/2019) (cit. on p. 29).

Online video sources

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

[Ric19] Liz Rice. *DIY Pen-Testing for Your Kubernetes Cluster - Liz Rice, Aqua Security*. 2019. URL: https://www.youtube.com/watch?v=fVqCAUJiIn0&feature=youtu.be&t=525 (visited on 07/17/2019) (cit. on p. 18).

[Gee17] Brad Geesaman. *Hacking and Hardening Kubernetes Clusters by Example [I] - Brad Geesaman, Symantec*. 2017. URL: https://www.youtube.com/watch?v=vTgQLzeBfRU&feature=youtu.be&t=185 (visited on 07/16/2019) (cit. on p. 29).

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).