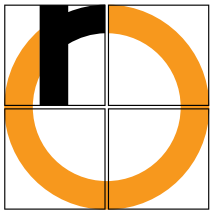
Hochschule **Rosenheim**
University of Applied Sciences



Fakultät für Informatik

Studiengang Informatik B.Sc.

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

Bachelor Thesis

von

Lukas Grams

Datum der Abgabe: TT.MM.JJJJ TODO

Erstprüfer: Prof. Dr. Reiner Hüttl

Zweitprüfer: Prof. Dr. Gerd Beneken

Erklärung

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

Declaration

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

Rosenheim, den XX.XX.2019 TODO

Lukas Grams

Abstract (german)

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

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

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

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

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

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

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

Abstract (english)

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

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

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

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

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

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

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

Contents

List of Figures

1 Wat17.
2 Pro19a.

List of Tables

Listings

List of abbreviations

AKS	Azure Kubernetes Service
CLI	Command Line Interface
CNCF	Cloud Native Computing Foundation
CRI	Container Runtime Interface
IaaS	Infrastructure-as-a-Service
k8s	Kubernetes
LAN	Local Area Network
LXC	LinuX Containers
OCI	Open Container Initiative
OCP	OpenShift Container Platform
OKD	Origin Community Distribution of Kubernetes
OS	operating system
PaaS	Platform-as-a-Service
VM	virtual machine

1 Introduction

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

1.1 Motivation

The increasing amount of Platform-as-a-Service (PaaS) solutions, cloud-hosted environments and microservice architectures introduce new attack scenarios. This creates the need for new defense strategies in both Development and Operations. Especially solutions providing 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

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 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 the Origin Community Distribution of Kubernetes (OKD) 3.11. OKD is a fork of k8s 1.11.0 with additional edits and updates (refer to section 2.4.2 for details). Through AKS, kubernetes v1.13.5 was already available at the same point in time. In order to better compare both implementations, the available k8s version 1.11 was chosen for AKS. Components providing k8s compliance will be the main focus, as these bear the most significance across all Kubernetes Certified solutions. A look at popular tools and frameworks used in such clusters will be avoided in order to keep the scope manageable, though some might be recommended as a mitigation. In order to be applicable to a higher number of use cases, attacks and measures seen in environments with exceptionally high security requirements might be mentioned , but not covered in their entirety. This might entail state-sponsored actors deploying zero-day exploits, which are not applicable to a majority of solutions deployed. This thesis aims to provide insight to the risks of providing a PaaS solution and mitigations thereof. As such, it will look at the capabilities a potential provider has to (mis-)configure such solutions - inherent risks of the technologies themselves are only explored when measures to mitigate them are accessible from a provider standpoint. In short, the goal is to improve the security of your k8s cluster, not k8s itself. To follow the OWASP Risk Rating Methodology[2] down the line, we need to define our threat agents. It is recommended to minimize the number of threat agents by treating them as equivalent classes when applicable[3]. Examining a list of threat actors published through SANS[4], we can will only exclude state-sponsored actors. This leaves us with cyber criminals, hacktivists, systems administrators, end users, executives and partners. Grouping the remaining actors by the factors used to estimate risks in section 4.2, we can identify find three scenarios that encompass all of the actors:

- Malicious third party attacking the solution from within the Local Area Network (LAN) and/or the internet
- Malicious third party attacking from inside a hijacked container, i.e. remotely executing code or commands

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

- Bad user, i.e. a negligent, hijacked or malicious account risking compromise of their own and/or other applications

2 Theory

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

2.1 Infrastructure-as-a-Service

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

2 ST11.

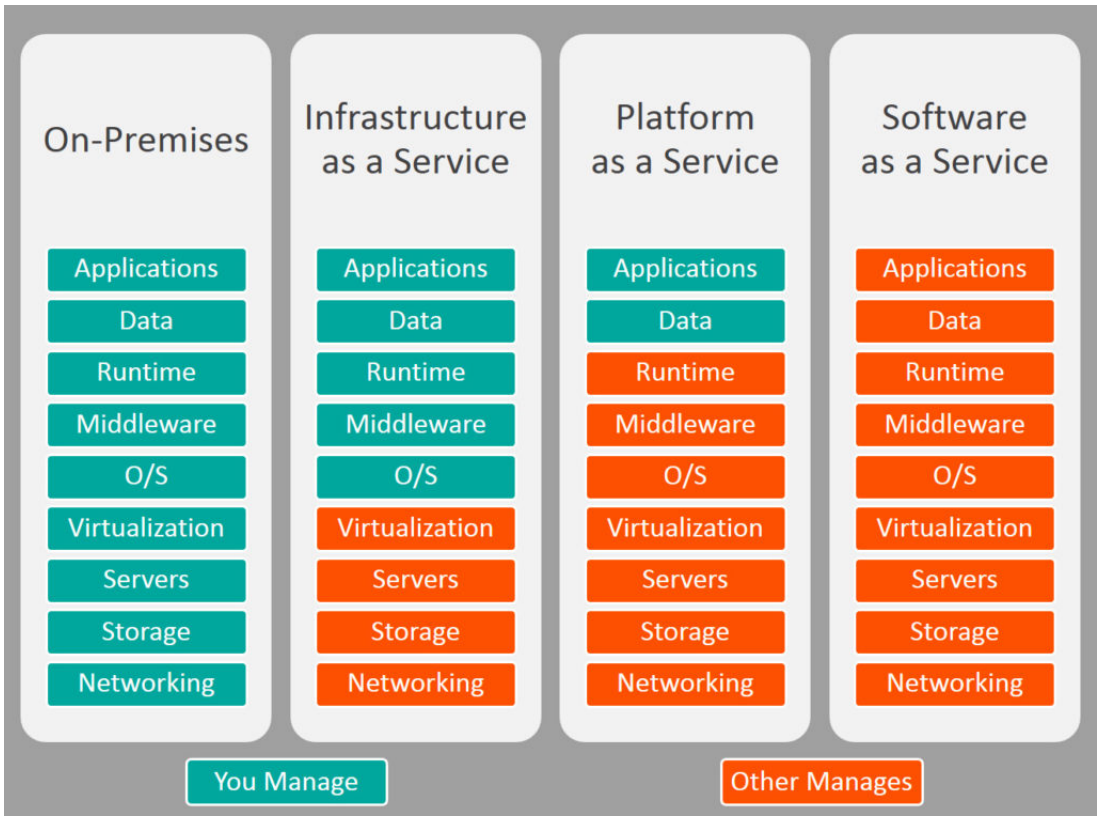


Figure 2.1 Comparison of responsibilities in different service models[a]

a Wat17.

2.2 Platform-as-a-Service

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

1 ST11.
2 Cor19a.

also have to extensively configure 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

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?

As the Docker website puts it, 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 on a host OS. The host system shares the kernel with all containers on it and might share other resources, but containers are built from container images which include any code and dependencies needed in order to make them independent of the infrastructure they are running on.[3]

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

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

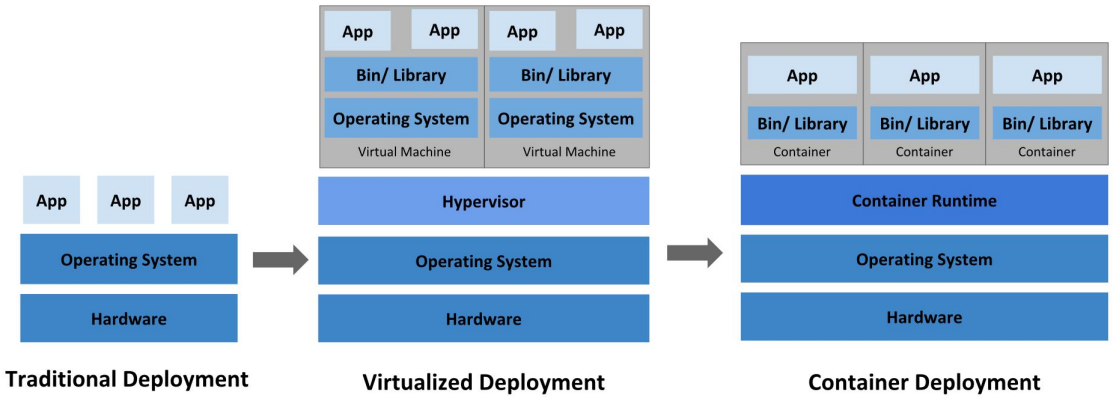


Figure 2.2 Comparison of different application deployments on the same hardware[a]

a Pro19a.

2.3.2 Differentiating docker

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

2.3.3 Images, image building and Dockerfiles

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

1 Cha17.
2 Inc19b.
3 Red19.
4 Red18b.
5 Inc18.

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

2.3.4 Container standards and interfaces

Without going into the nuances and historical developments, there are a multitude of programs mostly implementing three interfaces for container management. The two basic interfaces are the runtime and image specifications under the Open Container Initiative (OCI) which standardize how containers and container images should be formatted and executed.[1] The OCI also maintains a 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 some containers fail, the need for automated systems to manage these containers arises. Orchestrators can also provide many other advantages, like load balancing and automated scaling. Kubernetes systems are popular orchestrators currently in use. The sum of hosts running and controlling containers and the orchestrator are called a cluster.

2.4.1 Kubernetes

At its base, 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. When a user wants to deploy his application, it is intended for him to manipulate the target state and let the k8s system take care of the rest.[5]

There are many k8s distributions, many of which are certified kubernetes offerings, meaning they all comply to the same standards set by the Linux Foundation through the Cloud Native Computing

1 Fou19c.
2 Lew19a.
3 Lew19b.
4 Pro16.
5 Pro19b.

Foundation (CNCF), which oversees the project. The kubernetes code base is open source and maintained on Github, but any implementation fulfilling the (publicised) requirements can become k8s certified, regardless of how much code they changed.[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 functioning k8s cluster, multiple binary components are needed. Master components provide the control plane, while node components are run on each underlying machine in order to maintain and provide the environment to execute the containers you want to run eventually. Master components are often exclusively run on machines dedicated to them, which are called master nodes - in contrast to worker nodes, which run the containers your applications consist of.[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; User commands are typically directed at and processed by this component.

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

The important node components include:

- The kubelet, an agent running on each node in the cluster. It mostly monitors the state of any containers running because of kubernetes and interacts with the master components.

- The container runtime which is responsible for actually running containers. OCP uses Docker while AKS uses Moby, but any implementation of the CRI is supported.[3]

Kubernetes objects

TODO higher-level idea behind objects Pods, Services, Volumes, Namespaces

1 Fou19d.
2 Pro19c.
3 For additional information, refer to section 2.3.4

Kubernetes Controllers

TODO

higher-level idea behind controllers ReplicaSets, Deployments, StatefulSets, DaemonSets, Jobs

Using Kubernetes

TODO

build from scratch, buy CaaS/PaaS/IaaS, cloud vs. on-prem, different scopes/features from different products

https://kubernetes.io/docs/setup/pick-right-solution/

2.4.2 OpenShift

TODO

2.4.3 Azure Kubernetes Service

TODO

2.5 TODO: Others?

TODO

3 Deriving the attack surface

3.1 Defining procedure and approach

3.2 Identified vectors

3.2.1 Reconaissance through Kubernetes and platform control plane interfaces

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

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

TODO

3.2.2 Read confidentials through Kubernetes control plane interfaces

Gather confidential information through the Kubernetes dashboard & apiserver (platform interfaces are evaluated separately) TODO

3.2.3 Change configuration through Kubernetes control plane interfaces

Change the existing configuration through the Kubernetes dashboard & apiserver (platform interfaces are evaluated separately) TODO

3.2.4 Read confidentials through platform interfaces (mgmt console/API)

Gather confidential information through platform webinterfaces & apiserver(s) (Kubernetes interfaces are evaluated separately) TODO

3.2.5 Change configuration through platform interfaces (mgmt console/API)

Change the existing configuration through platform webinterfaces & apiserver(s) (Kubernetes interfaces are evaluated separately) TODO

3.2.6 Compromise internal k8s control plane components (etcd, scheduler, controller-manager)

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

3.2.7 Supply compromised container (base) image

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

3.2.8 Supply compromised k8s configuration

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

3.2.9 Compromise other application components (lateral movement)

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

3.2.10 Container breakout (R/W, Privilege Escalation)

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

3.2.11 Compromise local image cache

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

3.2.12 Modify running container

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

3.2.13 Misuse node resources (sabotage, cryptojacking)

The resources of a single node are used to run a container and may be misconfigured or misused for financial gains (mining cryptocurrencies) or to disrupt service availability (i.e. through fork bombing or misconfiguration) TODO

3.2.14 Hoard orchestration resources (sabotage)

The resources of the whole cluster may be misconfigured or misused to disrupt service availability (i.e. through fork bombing or misconfiguration) TODO

3.2.15 Misuse orchestration resources (cryptojacking)

The resources of the whole cluster may be misconfigured or misused for financial gains (mining cryptocurrencies) TODO

3.2.16 Add malicious container

A malicious container may be started within the cluster TODO

3.2.17 Add malicious node

A malicious node may be added to the cluster TODO

3.2.18 Bad user practice (outside of cluster)

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

3.2.19 Incufficient base infrastructure hardening

The underlying nodes could allow an attacker easy entry, even if the containers themselves are hardened TODO

3.2.20 Entry through known, unpatched vulnerabilities

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

4 Assessing the attack surface risk

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 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.

2 Fou17.
3 Gee17.

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

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

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

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

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

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

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

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

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

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

Default values are defined as the following:

SETUP OF PRACTICAL PART:

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

Considerable changes between 1.11.0 and 1.11.9 (Source: https://github.com/kubernetes/kubernetes/blob/master/CHANG
1.11.md): - action required: the API server and client-go libraries have been fixed to support additional non-alpha-numeric characters in UserInfo "extra" data keys. Both should be updated in order to properly support extra data containing "/" characters or other characters disallowed in HTTP headers. (#65799, @dekkagaijin) - https://github.com/kubernetes/autoscaler/releases/tag/cluster-autoscaler-1.3.1 - ACTION REQUIRED: Removes defaulting of CSI file system type to ext4. All the production drivers listed under https://kubernetes-csi.github.io/docs/Drivers.html were inspected and should not be impacted after this change. If you are using a driver not in that list, please test the drivers on an updated test cluster first. “‘ (#65499, @krunaljain) - kube-apiserver: the Priority admission plugin is now enabled by default when using –enable-admission-plugins. If using –admission-control to fully specify the set of admission plugins, the Priority admission plugin should be added if using the PodPriority feature, which is enabled by default in 1.11. (#65739, @liggitt) The system-node-critical and system-cluster-critical priority classes are now limited to the kube-system namespace by the PodPriority admission plugin. (#65593, @bsalamat)

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

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

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

4.2 Estimating the risk

4.2.1 Reconaissance through Kubernetes & platform control plane interfaces

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

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

4.2.2 Read confidentials through Kubernetes control plane interfaces

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

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

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

4.2.3 Change configuration through Kubernetes control plane interfaces

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

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

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

4.2.4 Read confidentials through platform interfaces (mgmt console/API)

Same as V02, TODO: merge

Same as V02, TODO: merge

4.2.5 Change configuration through platform interfaces (mgmt console/API)

Same as V03, TODO: merge

Same as V03, TODO: merge

4.2.6 Compromise internal k8s control plane components (etcd, scheduler, controller-manager)

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

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

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

4.2.7 Supply compromised container (base) image

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

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

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

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

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

Exactly the same as OCP

4.2.8 Supply compromised k8s configuration

Similar to V07, this can be done either untargeted by spraying to tutorials / help forums or targeted, similar to spear phising. If a cluster administrator does not fully analyze or understand the configuration

he gets from public sources, the cluster could be compromised fully, i.e. by implementing backdoors through malicious containers with special access and ability to be remotely accessed by the attacker.

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

Exactly the same as OCP

4.2.9 Compromise other application components (lateral movement)

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

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

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

4.2.10 Container breakout (R/W, Privilege Escalation)

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

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

a project access to the ‘privileged’ SCC permissions. This would significantly increase the risk of a container breakout!)

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

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

4.2.11 Compromise local image cache

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

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

Same as OCP.

4.2.12 Modify running container

Instead of deploying a container with malicious contents, an attacker can try to modify and use an already running container to its needs by loading additional tools/binaries, changing configurations or exfiltrating data. This could be done through an RCE vuln, ssh access or others, just like any compromised linux machine.

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

Same as OCP.

4.2.13 Misuse node resources (sabotage, cryptojacking)

This is a vector in contrast to orchestration resources. Assuming a cluster suitable for production (more than one worker node, probably more than a handful), the failure or misuse of a single node may be of use to the attacker, but has very limited impact to operations. This is because a significant part of the cluster is built to heal from failures of any node and/or container. TODO: absorb this with V14 and V15, since the risk is always lower than them and they cover everything?

Mining containers/binaries and/or fork-bombing tools with accompanying tutorials are easy to find publically. Cryptojacking is regularly cited as an up-and-coming attack.

Same as OCP.

4.2.14 Hoard orchestration resources (sabotage)

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

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

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

4.2.15 Misuse orchestration resources (cryptojacking)

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

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

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

4.2.16 Add malicious container

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

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

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

4.2.17 Add malicious node

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

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

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

4.2.18 Bad user practice (outside of cluster)

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

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

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

4.2.19 Incufficient base infrastructure hardening

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

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

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

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

4.2.20 Entry through known, unpatched vulnerabilities

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

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

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

5 Managing the attack surface risk

5.1 Defining procedures and approach

5.2 Managing the risk of V20 - Entry through known, unpatched vulnerabilities

5.2.1 Demonstrating the successful attack without security measures

5.2.2 Selecting security measures

5.2.3 Demonstration with implemented security measures

5.2.4 Risk reassessment

5.3 Managing the risk of V10 - Container Breakout

5.3.1 Demonstrating the successful attack without security measures

5.3.2 Selecting security measures

5.3.3 Demonstration with implemented security measures

5.3.4 Risk reassessment

5.4 Managing the risk of V09 - lateral movement

5.4.1 Demonstrating the successful attack without security measures

5.4.2 Selecting security measures

5.4.3 Demonstration with implemented security measures

5.4.4 Risk reassessment

5.5 How to proceed

6 Conclusion

6.0.1 On-premise and public cloud environment comparison

6.0.2 Multi-tenant isolation

6.0.3 Summary

Bibliography

Books

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

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

Online sources

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

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

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

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

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

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

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

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

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

[Pro19a] The Kubernetes Project. *What is Kubernetes*. 2019. URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/#going-back-in-time (visited on 07/08/2019) (cit. on p. 8).

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

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

[Red19] Inc. Red Hat. *Getting Started with Containers*. 2019. URL: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_atomic_host/7/html-single/getting_started_with_containers/index#working_with_docker_formatted_containers (visited on 07/08/2019) (cit. on p. 8).

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

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

[Fou19c] The Linux Foundation. *Open Container Initiative*. 2019. URL: https://www.opencontainers.org/ (visited on 07/08/2019) (cit. on p. 9).

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

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

[Pro16] The Kubernetes Project. *Container Runtimes Part 3: High-Level Runtimes*. 2016. URL: https://github.com/kubernetes/kubernetes/blob/242a97307b34076d5d8f5bbeb docs/devel/container-runtime-interface.md (visited on 07/08/2019) (cit. on p. 9).

[Pro19b] The Kubernetes Project. *What is Kubernetes*. 2019. URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (visited on 07/09/2019) (cit. on p. 9).

[Fou19d] The Linux Foundation. *Sustaining and Integrating Open Source Technologies*. 2019. URL: https://www.cncf.io/ (visited on 07/09/2019) (cit. on p. 10).

[Pro19c] The Kubernetes Project. *Kubernetes Components*. 2019. URL: https://kubernetes.io/docs/concepts/overview/components/ (visited on 07/09/2019) (cit. on p. 10).

[Fou17] OWASP Foundation. *OWASP Top 10 -2017 The Ten Most Critical Web Application Security Risks*. 2017. URL: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf (visited on 04/03/2019) (cit. on p. 17).

[Gee17] Brad Geesaman. *Hacking and Hardening Kubernetes Clusters by Example [I] - Brad Geesaman, Symantec [3:05]*. 2017. URL: https://www.youtube.com/watch?v=vTgQLzeBfRU&feature=youtu.be&t=185 (visited on 04/16/2019) (cit. on p. 17).