July 14, 2023

Container Based Applications

Defense in Depth

Version 1.2

https://ivision.com

Contents

1 **Introduction** . . . 3
1.1 A Brief History of Run Time 3
1.2 The Three Problems 4
1.3 How to Read This Paper 4

2 **Threat Model** . . . 5
2.1 Environments 6
2.2 Components 6
2.3 Flows 7

3 **Host Operating Systems** . . . 8
3.1 Threats 8
3.2 Tools 9

4 **Containers** . . . 10
4.1 Threats 10
4.2 Tools 12

5 **Container Orchestrators** . . . 13
5.1 Threats 13
5.2 Tools 14

6 **Bonus Problems** . . . 15
6.1 Managed Clusters 15

7 **Conclusion** . . . 17

Introduction

> Some people, when confronted with a problem, think "I know, I'll use a container orchestrator." Now they have three problems. - ivision[1]

1.1 A Brief History of Run Time

Computer software evolves like lots of things in life. Various layers accrue over time. These layers make some problems more manageable, but increase the overall complexity in the system.

One of the most complex pieces of software is the foundational layer of Operating Systems[2]. There has been a lot of focus and engineering effort put into evolving OSes for decades. This effort has been from a functional point of view and a security point of view.

From a security point of view, lots of the engineering effort has been spent on how best to enforce the Principle of Least privilege[3]. Historically, in Unix OSes, much of the model was based on what ordinary users and "root" users are allowed to do[4]. However, over time, this model proved to not be granular enough. Your choices are either to have very limited access or access to everything.

In Linux, which is the Unix operating system we will focus on, there have been various attempts to add this granularity. AppArmor[5] and SELinux[6] introduced mandatory access control (MAC)[7] implementations. Namespaces[8] allow partitioning kernel resources like mounts, process IDs, networking stack, and user IDs such that each process thinks they have their own unique resources. Capabilities[9] added a way to control what operations a particular thread can perform (this effectively provides the needed granularity for operations so users don't need root or suid). Cgroups[10] added a way to throttle resource use of memory, CPU, and I/O. Finally, seccomp[11] added the ability to precisely control what system calls are allowed to be invoked by a process.

Many of these features have been used to implement OS-level virtualization[12] in the form of a container. Alternatives for containers on Linux like Docker[13], LXC[14], and rkt[15] are numerous. We will focus on Docker.

---

[1] http://regex.info/blog/2006-09-15/247

[2] https://en.wikipedia.org/wiki/Operating_system

[3] https://en.wikipedia.org/wiki/Principle_of_least_privilege

[4] https://en.wikipedia.org/wiki/Superuser

[5] https://en.wikipedia.org/wiki/AppArmor

[6] https://en.wikipedia.org/wiki/Security-Enhanced_Linux

[7] https://en.wikipedia.org/wiki/Mandatory_access_control

[8] https://en.wikipedia.org/wiki/Linux_namespaces

[9] http://man7.org/linux/man-pages/man7/capabilities.7.html

[10] https://en.wikipedia.org/wiki/Cgroups

[11] https://en.wikipedia.org/wiki/Seccomp

[12] https://en.wikipedia.org/wiki/OS-level_virtualization

[13] https://en.wikipedia.org/wiki/Docker_(software)#Components

[14] https://linuxcontainers.org/lxc/introduction/

[15] https://coreos.com/rkt/

Once a development team has decided to containerize their application, a new problem arises – how are containers deployed and managed? Container orchestration[16] systems were engineered to solve that problem. The primary job of a container orchestrator is to ensure that the desired containers composing a software system are running and healthy.

Now we have our three problems.

1.2 The Three Problems

Modern web applications are increasingly being deployed in containerized platforms. Development teams that build and manage these platforms on their own must manage at least three complex layers of software:

1. The host systems running the containers and cluster nodes.
2. The containers running on the host systems and the tools used to build and configure them.
3. The container orchestration system used to deploy and manage the containers.

The security of all of these layers is crucial as they build on top of each other. It's essential to secure each layer adequately for defense in depth of the containerized platform[17].

This whitepaper will focus on defense in depth measures for these three layers by providing a set of guidelines to think about for each layer. The focus will be on infrastructure and development teams building container-based applications who need guidance on critical threats to consider.

These guidelines will use Docker as the reference implementation for containers and Kubernetes for the reference implementation of container orchestration. However, these guidelines apply to other container implementations and orchestration systems as well.

1.3 How to Read This Paper

We will start by providing a very basic threat model for the three aforementioned layers in the next section. The following sections will detail the enumerated threats and guidelines for managing them. We also provide some pointers to useful tools that can be used for finding specific instances of these threats.

The intended path is for development, security, and infrastructure engineers building their own clusters to read the threat model to get acquainted with the general architecture and then walk through each of the three Host Operating System, Container, and Container Orchestrator sections and consider how the threats detailed there relate to what they are building.

That being said, the three threat sections are also useful in isolation. For example, if you are just working on containerizing your Application, then the Container section is still beneficial.

[16] https://en.wikipedia.org/wiki/Orchestration_(computing)

[17] https://en.wikipedia.org/wiki/Defense_in_depth_(computing)

Threat Model

Threat modeling is the art of describing and documenting the security-relevant aspects of a complex system in a simplified form. It often starts with a good diagram, and we recommend using a relatively strict and simplistic type of diagram called a Data Flow Diagram (DFD) for developers and teams relatively new to threat modeling. The data flow diagram identifies the components in a system, which then allows for a per component analysis. For each component, the threat model then aims to understand "what can go wrong?".

In this section, we will develop a basic threat model for a Kubernetes deployment. We will use the "STRIDE" [1] threat enumeration technique:

Threat	Desired Property
Spoofing	Authenticity
Tampering	Integrity
Repudiation	Non-repudiation
Information Disclosure	Confidentiality
Denial of Service	Availability
Elevation of Privilege	Authorization
Outdated Software	All of the above

The threats will be enumerated against the following DFD:

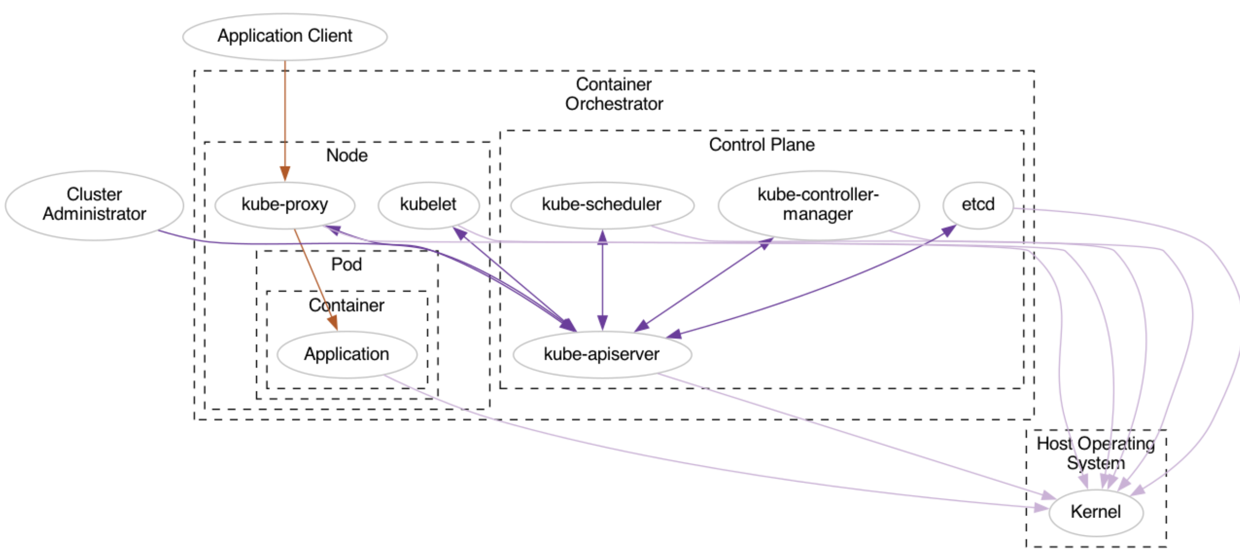

Figure # 2.1: Dataflow Diagram

NOTE:

[1] https://en.wikipedia.org/wiki/STRIDE_(security)

- The Host Operating System in this diagram could be one or several physical or virtual host machines.
- The Pod in this diagram could be one or more instances that may or may not communicate with each other.

The objects of interest in these diagrams are **environments** (the dashed boxes), **components** (the ovals), and **flows** (the arrows):

- **Environment**: A container with one or more components or other sub-environments. An environment may be a physical element such as a data center or a software environment such as a virtual machine. Environments are either trusted or untrusted. Trusted environments require an attacker to breach protections, such as data validation or locked doors, in order to influence the components contained within the system. Environments are also used to indicate ownership boundaries between different organizations.
- **Component**: A data processing element that is evaluated as a non-divisible entity in the threat model. Components may be processes, servers, or a software function implemented by a number of servers in coordination with each other.
- **Flow**: Represents the transfer of data between components. Flows that cross from an untrusted to a trusted environment (and vice versa) are of particular interest when threat modeling.

The specific objects used in this model will be defined in the following subsections.

For more details about threat modeling see Threat Modeling[2]: Designing for Security (Shostack 2014)

2.1 Environments

The environments in this model define the trust boundaries of the Host Operating System, Container, and Container Orchestrator.

Environment	Description
Container	Contains Operating System objects such as processes, mounts, and networking stacks.
Container Orchestrator	Contains all entities related to the management and deployment of containers. Also referred to as a cluster.
Control Plane	Contains all entities related to managing the cluster itself.
Host Operating System	Contains all the kernel and user space components that make up the Operating System.
Node	Contains all entities related to running the containers.
Pod	Contains one or more container instances.

2.2 Components

There are two classes of components we will consider: User Agents and Software Agents. User agents are people that interact with other components through software APIs. Software Agents are the primary software components in the system.

2.2.1 User Agents

Component	Description
Cluster Administrator	A user that administers the Container Orchestrator environment.
Application Client	A user that interacts with the Application through the Application API Call.

2.2.2 Software Agents

Component	Description
Application	A software program running in a Container.
etcd	A distributed key-value store that holds critical data for the cluster.
Kernel	The Host Operating System kernel.

[2] Shostack, A. (2014). Threat Modeling: Designing for Security. Wiley.

Component	Description
kube-apiserver	The core API server used to manage the cluster. Access is through a REST API.
kube-controller-manager	A daemon made up of control loops that maintain the desired state of the cluster.
kube-proxy	A network proxy responsible for proxying traffic from the kube-apiserver to a Node.
kube-scheduler	A scheduling service that is responsible for picking the optimal Node for a Pod to run on.
kubelet	An agent running in a Node that ensures that the desired containers are running and healthy.

2.3 Flows

Flows in this model represent API calls from the various components in the Container Orchestrator environment.

Flows	Description
Application API Call	An API call from the Application Client to the Application.
Kubernetes API Server Call	An API call from one of the Container Orchestrator components to the kube-apiserver.
syscall	A system call from an Application to a Kernel.

Host Operating Systems

Host Operating Systems should be architectured in a way to make it challenging for an attacker to raise their privileges or pivot to other systems.

Elevation of privileges could empower an attacker to overtake the Control Plane and gain full control of the cluster. It could also enable an attacker to have full control of the Applications running on the Nodes.

A Host Operating System may be subject to diverse forms of attacks. The following subsections illustrate some of the principal threats.

3.1 Threats

3.1.1 Spoofing

- ☐ Confirm that robust passwords and hashing algorithms (like SHA512 with 200000 iterations) are implemented to **authenticate** users.
- ☐ If SSH access is enabled to allow remote access, then guarantee that the SSHD configuration provides robust **authentication** options:
  - ☐ `PermitEmptyPasswords` should be configured to `NO`.
  - ☐ `PermitRootLogin` should be configured to `NO`.
  - ☐ `IgnoreRhosts` should be configured to `YES`.
  - ☐ `HostbasedAuthentication` should be configured to `NO`.
  - ☐ `RhostsRSAAuthentication` should be configured to `NO`.

3.1.2 Tampering

- ☐ Ensure that the **integrity** of directories and files containing data such as keys, passwords, and configuration settings are **not** world writable or writable by an high number of users through group settings. Such settings can allow an attacker to tamper with files (under `/etc/`, for instance), potentially leading to access or control of essential services.

3.1.3 Repudiation

- ☐ Guarantee that suitable system-level logging with `syslogd`[1] is enabled and configured accurately. Precise logging can help determine the type of attacks, where they originated from, and be used as argument for **non-repudiability**.

3.1.4 Information Disclosure

- ☐ Guarantee that directories and files containing **confidential** data such as keys, passwords, and configuration settings are **not** world readable or readable by a large number of users through group settings. Such settings can provide an attacker with vital information that can be used to gain access to other parts of the system.

[1] https://linux.die.net/man/5/syslog.conf

- ☐ Confirm that passwords are hashed with a cryptographically safe hashing algorithms with key stretching[2]. The increased number of rounds will ensure that brute force attacks take more time and reduce the likelihood that the **confidentiality** of user passwords is violated.
- ☐ Limit software name and version data from userspace programs that disclose **confidential** implementation details from appearing in things like banners and HTTP headers. These details can give attackers additional information required to construct more targeted attacks.
- ☐ Limit the number of open TCP and UDP ports. Keeping unnecessary ports open broadens the attack surface and can leak **confidential** information about the services being used on the system.

3.1.5 Denial of Service

- ☐ Ensure that resources used by critical processes are limited by appropriately using Control Groups[3] to prevent DoS attacks and maintain **availability**. Some components like `systemd` already utilize cgroups and have configuration options for it[4].

3.1.6 Elevation of Privilege

- ☐ Considerably reduce the number of users that need `sudo` access as it equates to **authorization** as root.
- ☐ Stay clear of setuid[5] binaries. Processes operating with setuid have effective privileges equivalent to an **authorized** root user.
- ☐ Restrict the permissions of `/tmp` to non-execute to obstruct **unauthorized** users from saving and executing harmful code from `/tmp`.

3.1.7 Outdated Software

- ☐ All host operating systems should be kept updated frequently. Any kernel exploits can enable an attacker to manipulate instances of all threats such as spoofing, tampering, repudiation, information disclosure, and elevation of privileges. Particularly serious vulnerabilities could enable an attacker to escape containers and escalate privileges to the root user.

3.2 Tools

The following scripts are handy in identifying some of the threats described in the earlier section. They are suitable for setting up a **baseline** for parts of your system that should be examined in greater detail.

- The **Linux Basic Security Audit (LBSA)** script[6] is useful to find weak SSH configurations, poor file permission settings, and other vulnerabilities.
- The **Linux Privilege Escalation Check** script[7] is helpful for detecting common privilege escalation paths.
- **Lynis** [8] searches for common vulnerabilities, poor configuration settings, and improper permission settings.

[2] https://en.wikipedia.org/wiki/Key_stretching

[3] https://en.wikipedia.org/wiki/Cgroups

[4] https://www.freedesktop.org/software/systemd/man/systemd.resource-control.html

[5] https://en.wikipedia.org/wiki/Setuid

[6] http://wiki.metawerx.net/wiki/LBSA

[7] https://github.com/sleventyeleven/linuxprivchecker/blob/master/linuxprivchecker.py

[8] https://cisofy.com/lynis/

Containers

Docker containers and their environment should be configured so that an attacker cannot pivot from a compromised container to exploit further backend services via the network interface, the host operating system, or adjacent containers.

Container compromise happens when an attacker successfully gains full or partial control of a running container. This can result from flaws in public-facing application logic that allows code injection, the use of vulnerable container images, or the injection of malicious images into the CI/CD pipeline.

A Docker container may be susceptible to several forms of attack.[1] The following subsections outline some of the primary threats.

4.1 Threats

4.1.1 Spoofing

- ☐ Ensure that the **identity** of a container image is always verified. An image[2] can be spoofed by replacing an image in a registry or through interception when an image is being transmitted to the registry from a CI/CD pipeline. Consider using something like Docker Content Trust[3] to sign your images.

4.1.2 Tampering

- ☐ The **integrity** of the image should be verified. An image[4] could be tampered with by compromising a registry or through interception when it is being transmitted to a container orchestrator. Consider using Docker Content Trust[5] for signing your images.

- ☐ Try to make the container filesystem read-only[6] to help enforce the **integrity** of the filesystem contents. Alternatively, only make writable those locations that are absolutely necessary for the implementation of the microservice, or consider using tmpfs mounts (such as Docker tmpfs mounts[7]) for writable storage.

4.1.3 Repudiation

- ☐ Configure logging and regularly monitor the logs so that suspicious behaviors and attack patterns can be identified by ivision_admins[8]. This will allow **non-repudiation** to be enforced. Docker has several logging drivers[9] that make it easy to integrate with other services to simplify logging and monitoring.

---

[1] This threat overview is adapted from theOWASP Docker Top Ten project as of March 20, 2019. Please bear in mind that this documentation project is new and under active development.

[2] Docker Image. `https://docs.docker.com/v17.09/glossary/?term=image`

[3] `https://docs.docker.com/engine/security/trust/content_trust/`

[4] Docker Image. `https://docs.docker.com/v17.09/glossary/?term=image`

[5] `https://docs.docker.com/engine/security/trust/content_trust/`

[6] Docker read-only option. https://docs.docker.com/engine/reference/commandline/run/#mount-volume–v—read-only

[7] Docker tmpfs mounts. `https://docs.docker.com/storage/tmpfs/`

[8] `https://docs.docker.com/config/containers/logging/`

[9] `https://docs.docker.com/config/containers/logging/configure/`

4.1.4 Information Disclosure

- ☐ **Never** store secrets in plain sight within a container. This includes in environment variables and on the filesystem. Storing secrets in plain sight in a container can allow an attacker to access **confidential** information. If the container was compromised in any way, an attacker would easily have access to them. Consider using a secrets manager like Docker secrets[10] or HashiCorp Vault[11].

- ☐ Only expose ports that are **absolutely** necessary for the execution of your application. For example, microservices **should** normally only be using **80** or **443**. Opening unnecessary ports in a container can allow attackers to compromise or glean **confidential** information about a microservice.

- ☐ Only mount host filesystems that are **absolutely** necessary for the functioning of the microservice. Mounting host filesystems inside a container can allow attackers to modify or extract **confidential** information about the host.

4.1.5 Denial of Service

- ☐ Ensure that resources within a container are always limited by appropriately using Control Groups[12] to prevent DoS attacks and maintain **availability**. An indifferent container administrator may incorrectly configure the container, thus allowing an attacker to exhaust resources such as CPU cycles and available memory. Docker offers several options to enforce these limits[13].

4.1.6 Elevation of Privilege

- ☐ Strictly limit the number of users needing access to the Docker socket. Access to the Docker socket equates to **unauthorized** root access on the **host** machine[14].

- ☐ Only mount the Docker socket inside a container when **absolutely** necessary. A compromised container possessing access to the Docker socket can easily escalate privileges to **unauthorized** root access on the **host** machine[15].

- ☐ Always follow the principle of least privilege and **never** grant your containers more capabilities than are absolutely necessary using tools like SELinux or AppArmor profiles. Poorly managed Linux capabilities[16] for Docker containers can substantially increase the attack surface of the host operating system and allow **unauthorized** access to system resources.

- ☐ Avoid creating privileged Docker containers with the `--privileged` flag. This flag grants the container nearly identical access on the host as processes running outside of the container[17]. This increased level of privilege could allow **unauthorized** agents access to system resources.

4.1.7 Outdated Software

- ☐ Regularly rebuild the Docker images to refresh the base Operating System and userspace components to the latest known good versions. Docker images that have not been appropriately updated allow an attacker to leverage threats such as spoofing, tampering, repudiation, information disclosure, and privilege escalation.

- ☐ Update the Docker software itself regularly. Docker implementations that have not been properly updated can be susceptible to threats such as spoofing, tampering, repudiation, information disclosure, and privilege escalation.

---

[10] https://docs.docker.com/engine/swarm/secrets/

[11] https://www.vaultproject.io/

[12] https://en.wikipedia.org/wiki/Cgroups

[13] Docker resource constraints. https://docs.docker.com/config/containers/resource_constraints/

[14] Docker Socket. https://docs.docker.com/engine/security/https/

[15] Don't expose the Docker socket (not even to a container). https://www.lvh.io/posts/dont-expose-the-docker-socket-not-even-to-a-container.html

[16] Linux capabilities. http://man7.org/linux/man-pages/man7/capabilities.7.html

[17] https://docs.docker.com/engine/reference/run/#runtime-privilege-and-linux-capabilities

4.2 Tools

The following scripts are helpful in identifying some of the threats detailed in this section. They are useful in establishing a **baseline** for parts of your system that should be analyzed in more detail.

- **Docker Bench** [18] is a set of auditing scripts used to find common flaws like the absence of resource limits, privileged containers, missing MAC profiles, and unsafe volume mounts.

[18] https://github.com/docker/docker-bench-security

Container Orchestrators

Container Orchestrators should be appropriately configured to utilize proper **authentication**, **authorization**, and encrypted information channels that preserve **integrity** and **confidentiality**.

A breach in **authentication** could allow an attacker to access the kube-apiserver, potentially reading sensitive information from the cluster or causing significant alterations. For similar reasons, sufficient **authorization** should be implemented to restrict the scope of accessible resources. Lastly, all Kubernetes API Server Calls should occur over encrypted channels to maintain **integrity** and **confidentiality**.

The orchestrator being examined here is Kubernetes[1]. It is primarily accessed through the `kubectl`[2] command. The subsections below outline some of the primary threats.

5.1 Threats

5.1.1 Spoofing

- ☐ Enforce **authentication** to the kube-apiserver to ensure that only verified parties can interact with the APIs. Kubernetes provides several methods for authentication[3], with Client Certificates and Service Account Tokens being the most common.
- ☐ Avoid enabling `--insecure-port` in production[4]. This feature is intended solely for testing and results in the **authentication** and **authorization** modules being disabled.

5.1.2 Tampering

- ☐ Utilize TLS for all API traffic to protect the **integrity** of transmitted data[5].

5.1.3 Repudiation

- ☐ Enable auditing[6] so that Cluster Administrators can identify suspicious behaviors and attack patterns. This allows for the enforcement of **non-repudiation**.

5.1.4 Information Disclosure

- ☐ Utilize TLS for all API traffic to protect the **confidentiality** of transmitted data[7].

---

[1] Kubernetes. https://kubernetes.io/

[2] kubectl. https://kubernetes.io/docs/tasks/tools/install-kubectl/

[3] https://kubernetes.io/docs/reference/access-authn-authz/authentication/

[4] https://kubernetes.io/docs/reference/access-authn-authz/controlling-access/#api-server-ports-and-ips

[5] https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/#use-transport-layer-security-tls-for-all-api-traffic

[6] https://kubernetes.io/docs/tasks/debug-application-cluster/audit/

[7] https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/#use-transport-layer-security-tls-for-all-api-traffic

5.1.5 Denial of Service

☐ Implement resource quotas and limit ranges to control the size and capacity of resources allocated to a namespace[8]. Establishing limits on resources assists in assuring the continued **availability** of the cluster.

5.1.6 Elevation of Privilege

☐ Use Role-based access control (RBAC) to enforce **authorization** for all APIs. Kubernetes provides an integrated RBAC component[9].

☐ Minimize read and write access to `etcd` as much as possible. According to the Kubernetes documentation, write access is nearly equivalent to granting admin access to the cluster and read access provides an easy path for **unauthorized** users to escalate their privileges[10].

☐ Use user impersonation[11] (`--as`) strictly when necessary and audit all existing uses of it accurately. Improper use of this functionality can potentially grant lesser privileged users **authorized** access to a broader range of resources in the cluster.

☐ Utilize security contexts[12] to set the minimum permissions that Pods and Containers are **authorized** to have in order to function correctly.

5.1.7 Outdated Software

☐ Regularly update all host operating systems. Any kernel exploits can allow an attacker to leverage instances of all threats such as spoofing, tampering, repudiation, information disclosure, and escalation of privileges. More serious vulnerabilities could potentially allow an attacker to seize control of the cluster.

5.2 Tools

The following scripts and Kubernetes plugins are useful in identifying some of the threats detailed in the previous section. They are effective in establishing a **baseline** for the system parts that need a more detailed analysis.

- **kube-hunter** [13] is beneficial for mapping the attack surface and discovering known vulnerabilities within Kubernetes clusters.
- **access-matrix** [14] is a krew[15] plugin that uncovers and visualizes the RBAC authorization matrix.
- The `kubectl auth can-i`[16] command is a simple and efficient method to determine the actions that the current user can execute.

[8] https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/#limiting-resource-usage-on-a-cluster

[9] https://kubernetes.io/docs/reference/access-authn-authz/rbac/

[10] https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/#restrict-access-to-etcd

[11] https://kubernetes.io/docs/reference/access-authn-authz/authentication/#user-impersonation

[12] https://kubernetes.io/docs/tasks/configure-pod-container/security-context/

[13] https://github.com/aquasecurity/kube-hunter

[14] https://github.com/corneliusweig/rakkess

[15] https://github.com/kubernetes-sigs/krew

[16] https://kubernetes.io/docs/reference/access-authn-authz/authorization/#checking-api-access

Bonus Problems

As we have seen, there are Host Operating Systems, Containers running on Host Operating Systems, and Container Orchestrators managing Containers. Surely that is the end of the line – no. As always, another layer has grown to manage and create Container Orchestrators.

Several popular platforms are used to implement this bonus layer. We will focus on Amazon Elastic Kubernetes Service (EKS), Google Kubernetes Engine (GKE), and Azure Kubernetes Service (AKS). As with all software, these new layers mitigate some threats and create new ones. The following sections will give a brief overview of these platforms and how they can help mitigate some of the threats discussed in previous sections.

A detailed treatment of the security properties of these platforms could easily fill another whitepaper and is not the central focus of this paper. As such, the following sections are strictly meant to be a brief overview of how these platforms relate to the layers and threats that have been explored in detail in the preceding sections.

6.1 Managed Clusters

Each of the platforms discussed allows clients to easily create managed Kubernetes Clusters. Instead of having to directly manage your own Control Plane and Nodes, the platform provides an API to create them for you. The Control Plane runs in an account fully managed by the platform provider. The Nodes run in the creating user's account.

6.1.1 Amazon Web Services EKS

The following table lists AWS services that help mitigate threats discussed in the preceding sections:

Threat	Services
Spoofing	IAM
Tampering	KMS
Repudiation	CloudWatch
Information Disclosure	KMS
Denial of Service	AWS Shield
Elevation of Privilege	IAM integration with RBAC and ConfigMap
Outdated Software	All of the above

When deploying Applications in EKS the above service integrations should be employed and carefully configured. Additionally, Amazon's documentation on securing your cluster should be reviewed[1].

6.1.2 Google Cloud Platform GKE

The following table lists Google services that help mitigate threats discussed in the preceding sections:

Threat	Services
Spoofing	Cloud IAM, GKE Cluster Trust
Tampering	Cloud KMS, GKE Cluster Trust

[1] https://docs.aws.amazon.com/eks/latest/userguide/security.html

Threat	Services
Repudiation	Cloud Monitoring, Cloud Logging
Information Disclosure	Cloud KMS, GKE Cluster Trust
Denial of Service	Cloud Armor
Elevation of Privilege	Cloud IAM integration with RBAC, GKE Sandbox
Outdated Software	All of the above

When deploying Applications in GKE, the above-listed service integrations should be employed and carefully configured. Additionally, Google's documentation on hardening your cluster's security should be reviewed[2].

6.1.3 Azure AKS

The following table lists Microsoft services that can help mitigate threats we discussed in the preceding sections:

Threat	Services
Spoofing	Azure Active Directory
Tampering	Azure KeyVault
Repudiation	Azure Monitor
Information Disclosure	Azure KeyVault
Denial of Service	Azure Monitor, Azure DDoS Protection
Elevation of Privilege	Azure Active Directory integration with RBAC
Outdated Software	All of the above

When deploying Applications in AKS, the above-listed service integrations should be employed and carefully configured. Additionally, Microsoft's documentation on hardening your cluster's security should be reviewed[3].

[2] https://cloud.google.com/kubernetes-engine/docs/how-to/hardening-your-cluster

[3] https://docs.microsoft.com/en-us/azure/aks/concepts-security

Conclusion

As we have seen, there are several complex layers involved when creating modern, container-based applications. Each of these layers requires a significant commitment to security in order to ensure the system as a whole is secure. Such a commitment is **not** a one-time analysis, but rather an ongoing one.

The threats and tools presented in this paper will aid you in this ongoing analysis. Furthermore, they hopefully will enable the reader to enumerate their own threats and apply proper security engineering practices[1] to their constantly evolving software architectures.

[1] https://c.com/wp-content/uploads/2020/05/ivision-White-Paper-Security-Engineering.pdf