Kubernetes Security Best Practices

A Definitive Guide

by

Ben Hirschberg

CTO and Co-Founder, ARMO

ARMO

Table of contents

Introduction 3

Kubernetes Security Best Practices: The 4C Model 4

Cloud 4

Prevent Access to KubeAPI Server 4

Encryption and Firewalling 6

Enforce TLS Between Cluster Components 6

Protect Nodes 7

Isolate the Cluster and API with Proper Configurations 7

Cluster 9

Using Kubernetes Secrets for Application Credentials 9

Apply Least Privilege on Access 9

Use Admission Controllers and Network Policy 10

Container 11

Use Verified Images with Proper Tags 11

Limit Application Privileges 12

Code 13

Scan For Vulnerabilities 13

What is Kubernetes Security? 14

The importance of Kubernetes Security 15

Implementing Kubernetes Security Best Practices 15

Security Updates for the Environment 15

Security Context 16

Resource Management 16

Shift Left Approach 17

Kubernetes Security Frameworks: An Overview 17

Conclusion 18

Introduction

Kubernetes, an open-source container orchestration engine, is well known for its ability to automate the deployment, management, and, most importantly, scaling of containerized applications.

Running an individual microservice in a single container is almost always safer than running it as processes in the same VM. To run a container, Kubernetes use the Pod concept (Point Of Deployment) which is a non-empty set of containers. When a Pod is launched in **Kubernetes**, it is hosted in a Kubernetes Node (the abstraction of a machine). A Kubernetes Node can behave as a “Worker” or a “Master” or both.. A Worker node(s) hosts the application Pods in the cluster, the Master Node(s) hosts the Kubernetes control plane components like API server, Scheduler and Controller manager. Together, they **constitute a cluster**.

Nodes provide CPU, memory, storage, and networking resources on which the control plane can place the Pods. A Kubernetes Nodes must run a variety of components, such as the Kubelet, the Kube-Proxy, and the container runtime that help Kubernetes run and monitor Pods.

Kubernetes Security Best Practices: The 4C Model

When constructing a defense-in-depth strategy, it is necessary to incorporate numerous security barriers in various areas; cloud-native security operates similarly and suggests implementing the same approach. The security techniques of Cloud Native Systems are divided into four different layers, which is referred to as “The 4C Security Model”: Cloud, Cluster, Container, Code. Addressing all these layers ensures comprehensive security coverage from development to deployment. The best practices for Kubernetes can also be classified into these four categories of the cloud-native approach.

Cloud

The cloud layer refers to the server infrastructure. Setting up a server on your preferred Cloud Service Provider (CSP) involves various services. While CSPs are primarily responsible for safeguarding such services (e.g., operating system, platform management, and network implementation), customers are still responsible for managing credentials, configuration of the infrastructure, monitoring and securing their data.

Prevent Unwanted Access to KubeAPI Server

The best practice is to limit the network layer access to Kubernetes API server. Depending on cloud provider, this is usually done by limiting the API server access to a single VPC. When users need to access the API from outside the VPC, they usually create a VPN service or SSH server in that specific VPC and using it as a gateway to Kubernetes API.

At the application level, there are three steps in the **Kubernetes API** access control process. The request is first validated, then examined for authenticity, and finally, it is subjected to admission control before it is granted access to the system. Check that the network access control and TLS connections are correctly configured before starting the authentication operation. Authentication procedures might be complicated.

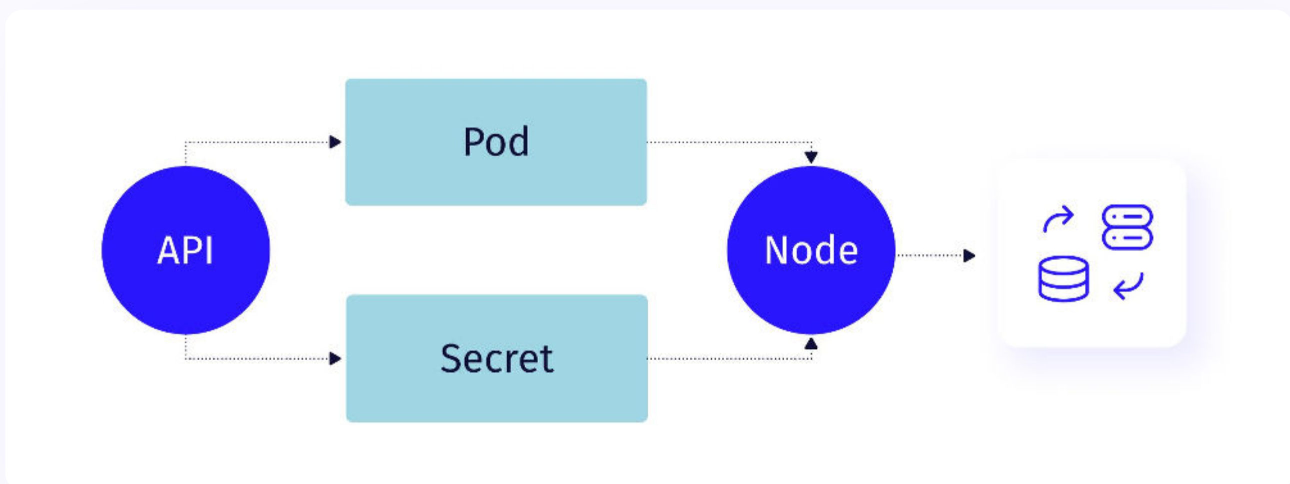
It is important that only TLS connections be used for connections to the API server, internal communication within **the Control Plane**, and communication between the Control Plane and the Kubelet. This is accomplished by supplying a TLS certificate and a TLS private key file to the API server using either command line option or configuration file. This can be called a basic security measure and nearly all Kubernetes distributions (Kind, Minikube, Rancher, OpenShift and etc) as both managed Kubernetes services (EKS, GKE and AKS) come with this best practice setup.

The Kubernetes API uses two HTTP ports, designated as localhost and secure port, to communicate. The localhost port does not require TLS, so requests made through this port will bypass the authentication and authorization components. Therefore, you must make sure this port is not enabled outside of the Kubernetes cluster's test configuration.

Additionally, because of the potential for security breaches, service accounts should be thoroughly audited, often. Particularly if they are associated with a **specific namespace** and are used for specific Kubernetes management duties. Each application should have a specialized service account. Thus, providing only the permissions necessary to the particular deployment. Using the default service accounts is usually avoided as it is mapped automatically to every Pod by default, which is contrary to the principle of least privilege. In order to avoid the creation of an extra attack surface, it is necessary to disable automatic mounting of default service account tokens when creating new pods if no individual service account is provided. It is very important to remove both service account tokens and private key certificate pairs from everywhere they are not needed. As of 2023, it is nearly impossible to revoke tokens and certificates in a Kubernetes cluster. If such a credential has been stolen or leaked, the only safe way is to re-install the whole cluster :(

Encryption and Firewalling

"Secrets" in Kubernetes are objects used to hold sensitive information about users, such as passwords, keys, tokens, and many other types of information. Since they are a different "Kind" than other configuration objects like "ConfigMaps", they can be treated in a special way at multiple levels thus limiting the exploitable attack surface. In addition, they provide flexibility to the pod life cycle so they can gain access to sensitive data. Secrets are namespaced objects (maximum length: 1 MB) kept in tmpfs on the nodes and password-protected.



The API server in Etcd stores Secrets as plain text by default, so it's important to enable encryption in the API server configuration. This way, if an attacker were to access the etcd data, they wouldn't be able to read it because they would also need the key used for encryption. In Kubernetes, the key is stored locally on the API server, so the data remains secure as long as the key is not compromised.

For Kubernetes to function correctly, the organization must enable firewalls and open ports. As an example, specific ports must be open on the Primary configuration. These include 6443, 2379-2380, 10250, 8472, and many others. In addition, several ports must be open on worker nodes, including ports 10250, 10255, and 8472.

Enforce TLS Between Cluster Components

Enabling TLS between the cluster components is critical for maintaining the cluster's security. Communication between APIs in a cluster is designed to be encrypted with the

help of TLS and it is the default of most Kubernetes distributions and managed Kubernetes services. Certificate production and distribution across cluster components is made possible by including appropriate certificates in the installation procedure in most installation methods. Because some components can be put on local ports, the administrator must be familiar with each component in order to be able to distinguish between trusted and untrusted communication in this situation.

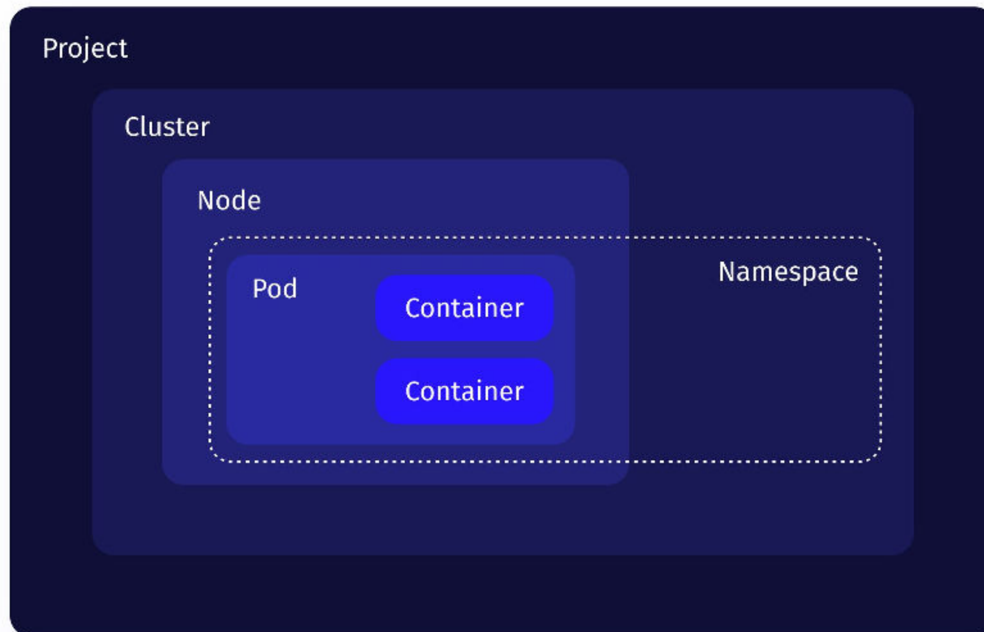
Protect Nodes

Nodes are responsible for the operations of your containerized apps. As such, it is essential to provide nodes with a secure environment. For example, a single node serving as both for control plane and application Podsfor the cluster can be used in a Kubernetes test environment. In a real world production environment, specific applications require separation. Most of the clusters employ different nodes for control plane components and worker nodes.

It can operate in either a virtual or a physical environment, depending on the organization's needs or regulatory compliance requirements, respectively.

Isolation in the Cluster with Proper Configurations

In a Kubernetes cluster, multiple users, clients, and applications share the cluster's resources. To ensure the protection of resources, it becomes necessary to enforce security measures. There are two options for achieving this security: soft multi-tenancy and hard multi-tenancy. Through the use of host volumes or directories, tenants can access shared data or gain administrative privileges. Soft multi-tenancy operates under the assumption that tenants are trustworthy, while hard multi-tenancy assumes that tenants may act maliciously. Therefore, a zero-trust policy is followed to maintain security.



Kubernetes offers various layers of isolation, including containers, pods, namespaces, nodes, and clusters. By utilizing a collection of nodes, pods can be run with a unique identity and access the necessary resources. Network rules also provide isolation within Pods. Network policy within Pods determines which Pods can communicate with each other. To comply with a hard-multitenant environment, the isolation of worker nodes and namespaces and preventing communication between namespaces are crucial.

API servers should not be accessible from the public internet or too many sources because doing so increases the attack surface and the likelihood of a compromise. However, with the help of rule-based access control (RBAC) rules, rotating the keys provided to users, and restricting access from IP subnets, it is possible to limit the risk.

In a Kubernetes cluster, kubelet is a daemon that runs on each node and is always in the background, processing requests. It is maintained by an INIT system or a service manager, and a Kubernetes administrator must configure it before it can be used. The setup details of all Kubelets within a cluster must be identical to each other.

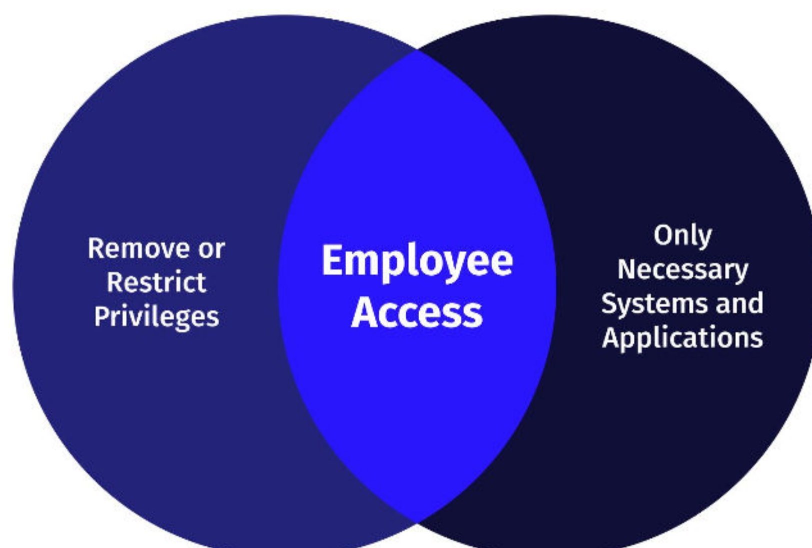
Cluster

Since Kubernetes is the most widely used container orchestration tool, we will concentrate our attention on it while discussing cluster security in general. Therefore, all security recommendations are limited to safeguarding the cluster itself in this section.

Using Kubernetes Secrets for Application Credentials

A Secret is an object in Kubernetes that holds sensitive information like passwords or tokens for authentication. It's important to know where and how these sensitive data are stored and accessed. Although a pod cannot directly access the secrets, it's still important to keep the secrets separate from other resources. This is especially important for applications that are accessible to the public and handle multiple processes, as they may be more vulnerable to security threats. Instead of passing secrets as environment variables, it's recommended to mount them as read-only volumes in containers for increased security.

Apply Least Privilege on Access



https://www.horangi.com/blog/least-privilege-policy

Most Kubernetes components require authorization. Thus, you need to be logged in for your request to access the cluster's resources to be authorized. **Role-based Access Control (RBAC)** determines whether an entity can call the Kubernetes API to perform a specific action on a specific resource.

RBAC authorization makes use of the rbac.authorization.k8s.io API group to make authorization decisions, which allows you to change policies on the fly. Activating RBAC requires an authorization flag set to a comma-separated list that contains RBAC and then restarting the API server.

```
kube-apiserver –authorization-mode=Example,RBAC –other-options –more-options
```

Consider the following illustration. As a role in the "default" namespace, it only provides the pods for reading purposes.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
namespace: default
name: read-user
rules:
– apiGroups: [“”]
resources: [“pods”]
verbs: [“get”, “watch”, “list”]
```

Use Admission Controllers and Network Policy

Admission controllers are plugins for the Kubernetes container orchestration system that govern and enforce how the cluster is used. This type of process might be regarded as a gatekeeper, intercepting (authenticated) API requests and altering them or even denying them entirely.

The best practice for this was based on the built-in PodSecurityPolicy admission controller. But, alas, those were deprecated in Kubernetes 1.21 and removed completely in Kubernetes 1.25. These were replaced with Pod Security Standards that still require a tool to enforce them. Two of these tools are OPA/Gatekeeper and Kyverno based admission controllers.

Admission controllers improve security by requiring a suitable security baseline across an entire namespace or cluster of computers. They can be used to prevent containers from operating as root or ensure that the container's root filesystem is permanently mounted read-only, among other things. Also available is a webhook controller, which allows for retrieving images exclusively from particular registries known to the organization while limiting access to unfamiliar image registries. Webhook controllers can also reject deployments that do not comply with security standards.

Container

Container runtime engines are required in order for the container to run in a cluster environment. Docker is, by far, the most common container runtime environment (CRE) for use with Kubernetes. In addition, it already contains a variety of images that developers may use to configure everything from Linux servers to Nginx servers.

Use Verified Images with Proper Tags

It is critical to scan images and only use those images permitted by the organization's policy because, due to the usage of unknown and vulnerable images, organizations are more vulnerable to the risk of being compromised. In addition, because much code is obtained from open-source projects, it is good to scan images before rolling them. Finally, the container registry is the central repository for all **container images**.

If a public registry is required, it can be used; however, it is usually recommended to use private registries as the container registry where possible. These private repositories can only be used to store images that have been approved. Thus, the risk of using

vulnerable images is reduced, and the harm posed to organizations due to supply chain attacks is also reduced.

You can also install the image scanning process at the CI; image scanning is a process in which a container image's contents and build process are inspected to find vulnerabilities, CVEs, or poor practices. Because it is incorporated into the CI/CD pipeline, it can prevent vulnerabilities from ever reaching a registry. In addition, it will protect the registry from any vulnerabilities introduced by third-party images utilized in the pipeline.

The image you scan for vulnerabilities may not have the same dependencies or libraries as the deployed image in your Kubernetes cluster, in which case you'll need to rescan it. The use of tags like "latest" or "staging" can also make it challenging to determine if the scan results are still accurate, as the image may have changed.

Using changeable tags can lead to different versions of the same image being used in containers, causing security issues and making troubleshooting more difficult. To avoid these issues, it's recommended to use immutable tags like "nginx:deploy-082022" whenever possible, so you know exactly which image was deployed and when.

Limit Application Privileges

Running a container as root can lead to privilege escalation, which is a critical security concern. This is because a root user inside a container has the same level of privileges as a root user on a host system, allowing them to execute commands with elevated permissions. To avoid this security issue, it's recommended to avoid running containers as root.

Examine tasks such as installing software packages, launching services, creating users, and other similar activities. For application developers, this presents several challenges. Additionally, while running apps on a Virtual Machine, you should avoid running them as the root user to avoid security risks.

This is also true in the case of containers. If someone gains access to the container, the root user will access and execute anything on the underlying host as if they were logged in as the root user. The risk of this includes:

- **Accessing** filesystem mounts
- **Gaining access** to username/password combinations that have been configured on the host for connecting to other services (yet another reason to use a secrets manager solution)
- **Installing** malicious software on the host
- **Gaining access** to other cloud resources

Code

Although we may be running various applications in the container, the code layer is sometimes called application security. It is also the layer over which businesses have the greatest control. Because the code of your apps and their related databases is the most crucial element of the system and is typically open to the internet, attackers will concentrate their efforts on this section of the system if all other components are securely protected.

Scan For Vulnerabilities

The majority of application software primarily relies on open-source packages, libraries, and other third-party components. As a result a vulnerability in any one of these dependencies can undoubtedly affect the complete functionality. Consequently, the likelihood that at least some of the programs that you are currently using contain vulnerabilities is significant.

Attackers frequently target deployments by exploiting known vulnerabilities in widely used dependency code; therefore, we must place some mechanism to verify those

dependencies regularly. Scanners may be included in the lifecycle of a container image deployed into Kubernetes, which can help decrease the possible attack surface and prevent attackers from stealing data or interfering with your deployment. These scanners will scan the dependencies and notify a vulnerability or update to the container, securing the container on the code level.

What is Kubernetes Security?

Kubernetes operates mostly in a cloud environment that includes the cloud, clusters, containers, and codes. Kubernetes security is primarily concerned with the combination of your code, clusters, and the container. Thus, security is built around all these features, including cluster security, **container security**, application security, and access control in a cloud environment.

It is necessary to continuously scan the entire system to identify misconfigurations and vulnerabilities as soon as possible to keep it secure. Additionally, pod container security and the Kubernetes API are important aspects of Kubernetes security. These APIs are essential for making Kubernetes scalable and flexible, and they include a significant amount of information.

There may be challenges in performing security practices on Kubernetes. For example, if a system has multiple different points that can be attacked via unauthorized entry or extraction of data, each point must be secured in order to secure the application. This large attack surface is a result of having many containers that could be exploited if not pre-emptively secured, taking up a lot of resources and time before or during deployment.

The Importance of Kubernetes Security

Since Kubernetes security is a broad, complex, and critical topic, it deserves special attention. The complexity of Kubernetes security is due to its dynamic and immutable nature and open source usage, which requires constant attention and resources to keep secure. Organizations that use containers and Kubernetes in their production environments must take Kubernetes security very seriously, as should all other aspects of their IT infrastructure.

If one compromises on a Kubernetes installation, it can lead to a large number of nodes that may be running compromised microservices. Thus, it poses a threat to the organization and may have an impact on its day-to-day operations. For example, there are two well-known types of CVEs, of which one is **CVE-2022-0185** (integer underflow). Discovered by William Liu and Jamie Hill-Daniel in 2022, this CVE shows how an attacker can exploit an unbound write function to modify kernel memory and access any other processes running on that node.

Implementing Kubernetes Security Best Practices

Security Updates for the Environment

When containers are running with a variety of open-source software and other software, it

is possible that a security vulnerability is discovered too late. It is, therefore, critical to scan images and keep up with software updates to see if any vulnerabilities have been discovered and patched. Using Kubernetes' rolling update capability, it is possible to gradually upgrade a running application by updating it to the most recent version available on the platform.

Security Context

Each pod and container has its own security context, which defines all of the privileges and access control settings that can be used by them. It is essentially a definition of the security that has been given to the containers, such as Discretionary Access Control (DAC), which requires permission to access an item based on the user id. It's possible that there are other program profiles in use that limit the functionality of the various programs. So, the security context refers to the manner in which we apply security to the containers and pods in general.

Resource Management

There are several types of resources available to every pod, including CPU and memory, which are required to execute the containers. So, to declare a pod, we must first specify how many resources we want to allocate to every container. A resource request for the container in a pod is specified, and the Kube-Scheduler utilizes this information to determine which node the pod should be placed on in order to best utilize the resources. Following its scheduling, the container will not be able to consume any resources in excess of those that have been allotted to it by the system. You must first observe how requests are processed before assigning resources to the containers. Only after you have completed the observation can you provide the resources that are required for the containers.

Shift Left Approach

The default configurations of Kubernetes are typically the least secure. Essentially, Kubernetes does not immediately apply network policies to a pod when it is created. Containers in the pods are immutable which means that they are not patched. Instead, they are simply destroyed and re-deployed when they become unavailable.

Shift left security involves conducting security testing early in the container development process, allowing for earlier identification and resolution of vulnerabilities. By catching security issues early in the software development lifecycle (SDLC), it helps reduce the time spent fixing them later at the production stage. This results in a shorter development cycle, improved overall quality, and faster progression to deployment phases.

Kubernetes Security Frameworks: An Overview

There are a variety of security frameworks that are available for the Kubernetes container orchestration system. A number of different organizations, such as MITRE, CIS, NIST, and others, have made the frameworks available. All of these frameworks have their own set of pros and cons, which allows the company to readily access them and apply anything they desire.

The CIS benchmark provides different methods and configuration management by which we can harden the security and configuration of the Kubernetes cluster. In accordance with the name, MITRE ATT&CK provides

different options or scenarios by which an attacker can attack the environment and take advantage of them. It also defines different methods by which we can patch the vulnerabilities or configuration issues that are required to protect the entire environment. You can use the PCI-DSS for the Kubernetes environment to implement the compliance-related configurations that are necessary for the environment – this is commonly used for fintech purposes. There are numerous frameworks that you can utilize to strengthen your Kubernetes security posture.

Conclusion

Cloud-native containers running in the Kubernetes environment are now considered part of the modern infrastructure. Microservices are now being used on a large scale in every enterprise, making it imperative that all security best practices be adhered to at all times. Because microservice operations handle a variety of data types, including personally identifiable information (PII) and other sensitive information, it is necessary to adhere to all of the security requirements to create a secure environment.

About ARMO

ARMO's mission is to build an end-to-end Kubernetes security platform, powered by open source. A platform which covers all Kubernetes security issues without adding to engineers' burden. ARMO focuses solely on open source based CI/CD & Kubernetes security, allowing organizations to be fully compliant and secure from code to production. Our solutions make security simple and frictionless for DevOps and are embraced by security.

Kubescape

ARMO is the creator of the one of the most complete and fastest growing open-source security platforms for Kubernetes - Kubescape. Kubescape lets users address the whole organization's CI/CD & Kubernetes security and compliance needs with one easy-to-use platform. It gives DevOps, I&O managers and cloud security professionals a common ground and a full picture. As a result, your organization can maintain security control, compliance and visibility, while keeping code delivery fast and agile.

ARMO Platform

ARMO Platform is the enterprise solution based on Kubescape. It's a multi-cloud Kubernetes and CI/CD security single pane of glass. Features include: risk analysis, security compliance, misconfiguration and image vulnerability scanning, RBAC visualization.

It simplifies, makes transparent and reduces the complexity of Kubernetes security.

Provides security pros visibility, transparency and control to assess risks.

DevOps-friendly self-service with full risk profile for security governance.

ARMO

Join the discussion
on **Slack**

Get involved
on **Github**

Follow us
on **Twitter**

ARMO PLATFORM

Sign up for
ARMO Platform