EKS to the next level

Internship Report

Bachelor's degree in Electronica-ICT
field Cloud & Cybersecurity

Academic year 2022-2023

Campus Geel, Kleinhoefstraat 4, BE-2440 Geel

Wim Hembrechts

R0751084

LID VAN ASSOCIATIE KU LEUVEN

THOMAS MORE

Introduction

The internship experience is a crucial phase in a student's academic journey, providing an opportunity to bridge the gap between theoretical knowledge and practical application. It serves as a stepping stone towards professional development, allowing individuals to gain valuable insights into their chosen field of study and industry. This report presents a comprehensive overview of my internship at ACA Group.

During the internship, I was a member of the OpsKlaar team within the company ACA Group, a highly motivated group of people responsible for developing, maintaining, monitoring, and troubleshooting cloud environments for their clients. My job within the team was to improve the security of the cloud environment they use as a starting point for each new client.

My internship spanned from February 27, 2023, to May 26, 2023, followed by an evaluation process conducted by Thomas More College. I gave the school the opportunity to follow the progress of my internship. I maintained a weekly logbook that contained all my activities.

I would like to extend my deepest appreciation to my internship mentor, Jana Garcia Gonzalez, my technical internship mentor, Jan Beerden, and my internship supervisor, Luc Celis, for their invaluable guidance, expertise, and constant encouragement throughout my internship. I would also like to express my gratitude to all the members of the OpsKlaar team and to the company ACA Group for providing me with the opportunity to undertake this internship.

Table of Contents

1 INITIATION PHASE

1.1 Onboarding

At the start of my internship, I was welcomed and briefed by my internship mentor. She gave me all the things I needed for my internship (laptop, laptop bag, etc.). I also received a tour of the office. After which, my technical internship mentor gave an explanation about the technical aspect of the internship, and I received my internship assignment with all its details.

1.2 The assignment

When a new customer comes along who wants to run applications in the cloud, the OpsKlaar team starts with a standard environment. This standard environment is then later adapted to meet the specifications required by the customer. This standard environment consists of an almost empty EKS cluster in AWS that is provisioned via Terraform by means of a Jenkins job. This EKS cluster is shielded from the outside world via IAM users and roles, but no other restrictions have been implemented in this EKS cluster.

To improve the security of this standard environment, I was given the task of checking whether the EKS cluster complies with the EKS security best practices provided by AWS. Alongside the compliance check, I was also tasked with researching and implementing some additional components that also needed to comply with the EKS security best practices.

1.2.1 EKS security best practices

Security matters inside an EKS cluster. Without proper security attackers can easily gain access to the cluster. To make sure this can’t happen AWS made a guide that contains a ton of security best practices specifically for and EKS cluster. To confirm that the EKS cluster in the standard environment was adhering to these best practices I had to compare the cluster with these best practices.

1.2.2 Secret encryption in etcd with KMS

A secret is a resource used to store sensitive information or data inside K8s. These secrets are then base64 encoded and stored inside a volume named etcd. Because base64 encoding isn’t secure, the team would like to add an extra security layer to these secrets by implementing KMS encryption on the EKS cluster.

1.2.3 Replacement for Pod Security Policies

PSPs have been deprecated since K8s v1.21 and removed from K8s v1.25. Because the EKS clusters are going to be upgraded to v1.25 and beyond, a replacement for PSP needs to be implemented to ensure the security of the pods.

1.2.4 Policy enforcement tool

To further secure or restrict the EKS cluster, a tool to enforce policies needs to be implemented. This policy engine should be able to prevent a resource from being created or updated when a policy violation is detected. There should also be a way to only alert when a violation takes place without preventing the creation or update of the resource. The policies enforced by this policy engine should be fairly easy to write or adjust. By implementing a policy enforcement tool, some validation tasks can be outsourced to this tool, freeing up time for other tasks.

1.2.5 Scanning images for CVEs with Trivy

Many containers run within a K8s cluster. Each container runs a particular image that may contain vulnerabilities. The OpsKlaar people want to gain more insight into which vulnerabilities are present in each environment. This information can later be used to eliminate these vulnerabilities. To gain this insight, a CVE scanner needs to be deployed inside the EKS cluster.

1.2.6 Compliance control with Kube-bench/Kubescape

A K8s cluster can be set up in different ways. Of course, it doesn't mean that every way establishes a secure cluster. To make sure that the cluster complies with the security best practices the team would like to introduce a tools that performs cluster compliance checks. The tools presented for research are Kube-bench and Kubescape.

1.2.7 Runtime threat detection

As said before, there are many containers running within a K8s cluster. In each container, processes are running. Some of these processes might be undesirable or even dangerous. We want to know where and when these processes are taking place. A runtime threat detection tool can be used to gain this information. With this information, we can prevent these processes from taking place at a later date.

1.3 Understanding the standard setup

I first had to understand what this standard environment looked like, since I had to expand it. To gain the necessary knowledge, I was given access to the repositories of a client environment. My technical internship mentor went over the most important information in these repositories with me so that I knew how to deploy tools myself later on according to the same principles that are used by the team when deploying their tools. In retrospect, I went over these repositories by myself to make sure that I understood everything.

1.4 Gain experience and access

To avoid losing time later on in the internship, I took my time to gain experience with the tools and programs I knew I would need during my internship. It also took me some time to gain access to the test environment that was set up especially for my internship.

2 RESEARCH PHASE

To complete my internship assignment, I first had to research each component individually to gain enough knowledge to later implement it in my test environment.

Broadly speaking, the flow of this research phase looks as follows: study > test locally > troubleshoot. This process keeps repeating itself until I have enough knowledge about each component. This section contains all the research I've gathered for each component.

2.1 EKS security best practices

The EKS best practices security guide provides advice on how to protect information, systems, and assets inside an EKS cluster while making sure performance is still optimal even in production environments. The guide is separated into multiple smaller topics, each containing recommendations and best practices associated with that topic.

The topics that are handled in this guide are:

- IAM
- Pod security
- Runtime security
- Network security
- Multi-tenancy
- Detective controls
- Infrastructure security
- Data encryption and secrets management
- Regulatory compliance
- Incident response and forensics
- Image security

The recommendations and best practices listed in these topics provided more insight into how an EKS cluster can be set up safely. However, this guide only yielded a few results that still needed to be implemented. The OpsKlaar team has already done a great job because most recommendations have already been implemented. Some of those that are still missing can be enforced by a policy control tool. This is also part of my assignment, so this will be handled further in the realization part.

2.2 Secret encryption in etcd with KMS

Secrets in K8s are resources used to store valuable or sensitive data like passwords, ssh keys, credentials, etc. In K8s, they are stored in a base64-encoded format inside a volume named etcd. When using AWS EKS, this etcd volume is encrypted at disk-level, but the content of this volume isn't. According to the defense-in-depth security strategy, this isn't enough and states that we need to encrypt the content inside this volume. To accomplish this we can use a technique called envelope encryption.

Envelope encryption is a technique that involves encrypting data using a symmetric encryption key, which is then encrypted using a separate, higher-level encryption key. This two-layered approach provides an additional layer of security by keeping the symmetric encryption key protected while still allowing efficient encryption and decryption of data using the symmetric key.

AWS gives you the opportunity to use envelope encryption on secrets in an EKS cluster with the help of a KMS key. KMS is an AWS paid service through which you can manage keys. The price of using a KMS key depends on its usage.

Benefits

- Secrets in etcd are encrypted, which means an extra layer of security.
- The KMS key can be rotated for extra security.

Drawbacks

- Can't be disabled after enabling it.
- Only secrets can be encrypted, not other resources stored in etcd.
- When the KMS key is deleted, the cluster can't be recovered.

2.2.1 Key requirements

To be able to encrypt secrets in EKS via a KMS key, the key needs to comply with some requirements:

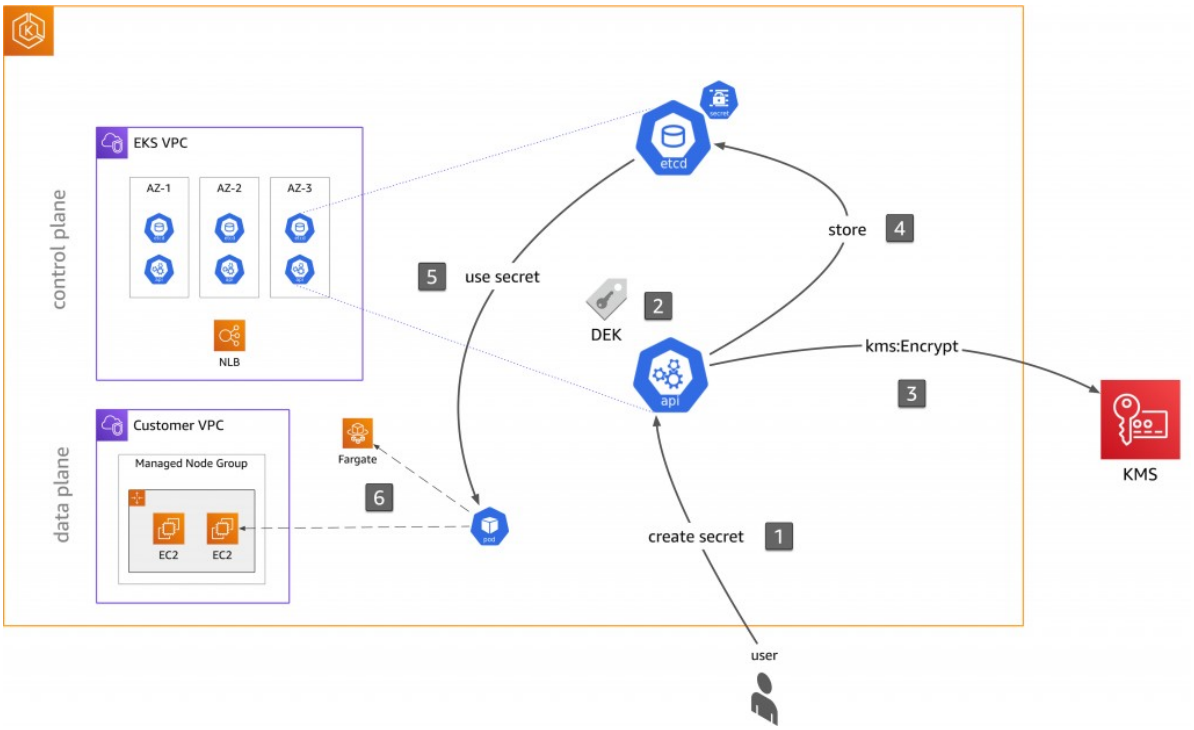
- The key must be symmetric.
- Must be able to encrypt and decrypt data.
- The key needs to be created in the same region as the EKS cluster.
- The IAM role of the control plane must be able to access and use the key.

2.2.2 KMS cost

Each KMS key costs $1/month, and each time the key is used for an encrypt or decrypt action, this counts as a request. Each 10,000 requests made per month costs $0.03. But in the first 12 months of using the KMS key, you get 20,000 free requests each month.

To make sure the cost doesn't outweigh the benefits, I made a cost prediction based on information gathered from some client environments. I predicted that there would be roughly 2000 requests for the KMS key each month. This would translate to approximately $1.01 each month, this is negligibly small.

2.2.3 How it works



After enabling the encryption of secrets stored inside etcd, the following steps take place when creating a secret.

1. A user or application creates a secret.
2. K8s API server in control place generates Data Encryption Key (DEK) locally. The plaintext payload of the secret is then encrypted by this DEK.
3. K8s API server uses the KMS key to encrypt DEK.
4. The encrypted secret and encrypted DEK are stored inside the etcd volume.
5. When the secret needs to be used, the API server decrypts the DEK and uses it to decrypt the encrypted secret.
6. At this moment, the secret can be used.

2.3 Replacement for Pod Security Policies

Cluster administrators and operators must replace PSPs since they are scheduled for removal and are no longer being actively developed. There are two solutions that can take the place of PSPs.

2.3.1 Pod Security Standards & Pod Security Admission

Pod Security Standards (PSS) define different isolation levels for pods, and Pod Security Admission (PSA) lets you define how you want to restrict the behavior of pods in a clear, consistent manner.

2.3.1.1 Pod Security Standards

The PSS define three different policies to broadly cover the security spectrum. These policies are cumulative and range from highly permissive to highly restrictive.

These policies are defined as:

- **Privileged:** Unrestricted policy, providing the widest possible level of permissions. This policy allows for known privilege escalations.
- **Baseline:** Minimally restrictive policy that prevents known privilege escalations. Allow the default (minimally specified) pod configuration.
- **Restricted:** Heavily restricted policy, following current pod hardening best practices.

2.3.1.2 Pod Security Admission

PSA places requirements on a pod's security context and other related fields according to the three levels defined by the PSS: privileged, baseline, and restricted.

These modes are defined as:

- **Enforce:** Policy violations will cause the pod to be rejected.
- **Audit:** Policy violations will trigger the addition of an audit annotation to the event recorded in the audit log, but are otherwise allowed.
- **Warn:** Policy violations will trigger a user-facing warning but are otherwise allowed.

2.3.1.3 Implementation

PSS and PSA are configured at the K8s namespace level using labels. There are multiple options to configure the PSS and PSA. I've only explored the options that make use of manifest files, because only then will the changes be visible in the code repositories.

You can configure PSS and PSA inside the manifest file of the namespace.

Here is an example that configures PSS and PSA on the namespace demo where:

- Pods that don't comply with the baseline policy requirements are blocked.
- Pods that don't meet the restricted policy requirements trigger a user-facing warning and get an audit annotation.
- The baseline policy is set to version v1.25 of K8s.

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: demo
  labels:
    pod-security.kubernetes.io/enforce: baseline
    pod-security.kubernetes.io/enforce-version: v1.25
    pod-security.kubernetes.io/audit: restricted
    pod-security.kubernetes.io/warn: restricted
```

The PSA modes are not mutually exclusive, and different modes can be used with different levels.

You can't configure exemptions inside the namespace manifest file, this is only possible in the next option.

When configuring PSS and PSA inside the admission configuration manifest file, there are a few differences compared to the previous option. You can only set defaults for the entire cluster. But here you have the option to configure exemptions.

Here is an example that configures the same modes and levels. But in this case, the configuration is implemented on the entire cluster and not only in the demo namespace.

```yaml
apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfiguration
plugins:
  - name: PodSecurity
    configuration:
      apiVersion: pod-security.admission.config.k8s.io/v1
      kind: PodSecurityConfiguration
      defaults:
        enforce: "baseline"
        enforce-version: "v1.25"
        audit: "restricted"
        audit-version: "latest"
        warn: "restricted"
        warn-version: "latest"
      exemptions:
        usernames: []
        runtimeClasses: []
        namespaces: []
```

As you can see at the bottom of the picture, you can add exemptions. Exemption dimensions include usernames, runtimeclassnames, and namespaces.

2.3.2 Policy as Code

PAC provides guardrails for users and prevents unwanted behaviors through prescribed and automated controls. PAC uses the Kubernetes Dynamic Admission Controller to intercept K8s API server requests. The request payload is then mutated and/or validated based on policies written and stored as code.

There are several open-source PAC solutions available, but these are not part of the Kubernetes project. Some PAC solutions are:

- OPA Gatekeeper
- Kyverno
- Kubewarden
- jsPolicy

PAC will be further discussed in the next section, where I examine the first two tools.

2.4 Policy enforcement tool

As mentioned in the previous section, there are a few tools that can serve as policy enforcement tools. OPA Gatekeeper and Kyverno are the largest players when it comes to policy engines used in Kubernetes, so I thoroughly researched both separately and compared them afterwards.

2.4.1 OPA Gatekeeper

OPA Gatekeeper is the Kubernetes-specific implementation of Open Policy Agent (OPA). OPA is a general-purpose policy engine that has the ability to unify policy enforcement across the stack. OPA is a graduated project in the CNCF landscape and was originally created by Styra. It uses a purpose-built policy language named Rego that was inspired by Datalog. Gatekeeper is a validating and mutating webhook that enforces CRD-based policies and uses the OPA constraint framework to describe and enforce policies.

2.4.1.1 OPA Constraint framework

A constraint template needs to be defined before the constraint itself. The Rego code that enforces the constraint and the schema of the constraint are both described in the constraint template. The constraint template below requires that all labels in the k8srequiredlabels constraint need to be present:

```
apiVersion: templates.gatekeeper.sh/v1
kind: ConstraintTemplate
metadata:
  name: k8srequiredlabels
spec:
  crd:
    spec:
      names:
        kind: K8sRequiredLabels
      validation:
        # Schema for the `parameters` field
        openAPIV3Schema:
          type: object
          properties:
            labels:
              type: array
              items:
                type: string
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8srequiredlabels

        violation[{"msg": msg, "details": {"missing_labels": missing}}] {
          provided := {label | input.review.object.metadata.labels[label]}
          required := {label | label := input.parameters.labels[_]}
          missing := required - provided
          count(missing) > 0
          msg := sprintf("you must provide labels: %v", [missing])
        }
```

After the constraint template is defined, you can start defining the constraint itself. In the constraint, you need to define which constraint template you want Gatekeeper to enforce.

In this example, the K8sRequiredLabels constraint template (previously defined) is used to make sure the label Gatekeeper is defined on all namespaces:

```yaml
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sRequiredLabels
metadata:
  name: ns-must-have-gk
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Namespace"]
  parameters:
    labels: ["gatekeeper"]
```

The constraint will be applied to the objects defined in the match field. These are the supported matchers: kinds, scope, namespaces, excludedNamespaces, labelSelector, and name.

The purpose of a constraint is described in the parameters field. It can be referred to in the Rego source code of the constraint template as input.parameters. Values entered into the parameters field of the constraint populate input.parameters.

The action that needs to happen when a violation occurs is defined in this field. The supported values for this field are: deny, warn, and dryrun.

- **Deny:** This action rejects the resource that violates the constraint, preventing it from being created or updated.
- **Warn:** This action allows the resource creation or update to proceed but generates a warning in the terminal indicating that the resource violates the constraint.
- **Dryrun:** When this action is specified, Gatekeeper will perform a dry run evaluation of the constraint and provide a report of violations without actually blocking the resource creation or update.

2.4.1.2 Audit

OPA Gatekeeper has the option to evaluate existing resources. You can gather the results of this audit in three different ways.

There are a few Prometheus metrics available:

- **gatekeeper_audit_last_run_time:** most recent audit start timestamp
- **gatekeeper_audit_last_run_end_time:** most recent completed audit end timestamp
- **gatekeeper_violations:** total number of violations in the last audit, broken down by violation severity

Violations detected during an audit run are visible in the status field of the constraint that triggered the violation. Only violations from the latest audit run are visible.

This example shows two violations in a constraint's status field:

```yaml
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sRequiredLabels
metadata:
  name: ns-must-have-gk
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Namespace"]
  parameters:
    labels: ["gatekeeper"]
status:
  auditTimestamp: "2019-05-11T01:46:13Z"
  enforced: true
  violations:
    - enforcementAction: deny
      group: ""
      version: v1
      kind: Namespace
      message: 'you must provide labels: {"gatekeeper"}'
      name: default
    - enforcementAction: deny
      group: ""
      version: v1
      kind: Namespace
      message: 'you must provide labels: {"gatekeeper"}'
      name: gatekeeper-system
```

When violations occur, these will also be visible in the logs of the audit pod, this time in JSON format. The audit logs will only show the violations of the latest audit run, like in the previous option. But these results are more detailed than the results visible in the constraint status.

Here is an example of an audit log.


```
{
  "level": "info",
  "ts": 1632889070.3075402,
  "logger": "controller",
  "msg": "container <kube-scheduler> has no resource limits",
  "process": "audit",
  "audit_id": "2021-09-29T04:17:47Z",
  "event_type": "violation_audited",
  "constraint_group": "constraints.gatekeeper.sh",
  "constraint_api_version": "v1beta1",
  "constraint_kind": "K8sContainerLimits",
  "constraint_name": "container-must-have-limits",
  "constraint_namespace": "",
  "constraint_action": "deny",
  "constraint_annotations": {
    "test-annotation-1": "annotation_1"
  },
  "resource_group": "",
  "resource_api_version": "v1",
  "resource_kind": "Pod",
  "resource_namespace": "kube-system",
  "resource_name": "kube-scheduler-kind-control-plane",
  "resource_labels": {
    "env": "prod",
    "my-app-system": "true"
  }
}
```


2.4.1.3 Policy library

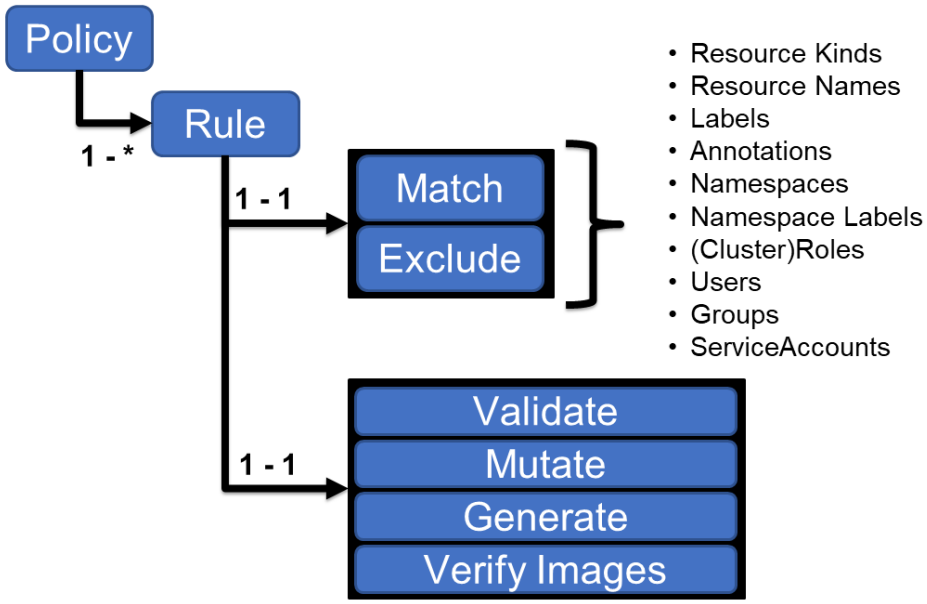
OPA Gatekeeper has a community-owned policy library that contains constraints and constraint templates you can use. The library contains both validation and mutation policies. These policies can be implemented using kustomize or kubectl. The OPA website contains a manual on how you can add policies to this library yourself.

2.4.2 Kyverno

Kyverno is a K8s-native policy engine that focuses on policy enforcement and management within K8s clusters, originally created by Nirmata. It allows you to define and enforce policies for Kubernetes resources using simple and intuitive rules expressed in YAML or JSON format. This allows the use of familiar tools such as kubectl, git, and kustomize to manage policies. Kyverno policies can validate, mutate, generate, and cleanup K8s resources, plus ensure OCI image supply chain security. Kyverno has a CLI version that can be used to test policies and validate resources as part of a CI/CD pipeline.

2.4.2.1 Policies and rules

Policies in Kyverno are written as Kubernetes resources.



There are two options when defining policies: cluster-wide policies (kind: ClusterPolicy) and namespaced policies (kind: Policy). The cluster-wide policies are implemented on the entire K8s cluster at once, while the namespaced policies are only implemented in a specific namespace.

Each policy is a collection of rules. A rule must contain a match field, may contain an exclude field, and must contain one of the four valid rule types. These are the four types of rules that can be used with Kyverno:

- **Generate:** Used to generate new resources when a resource is created or updated. It can be useful to create supporting resources.
- **Mutate:** Used to modify matching resources in a given way.
- **Validate:** Used to validate resources to check their compliance.
- **VerifyImages:** This can be used to verify container image signatures. Currently a beta feature. According to Kyverno, it is not ready for production use because there may be breaking changes.

Only any or all may be present in either the match or exclude clauses because they both have the same structure. Any will act as a logical OR operation, and all will act as a logical AND operation. Under the any or all clause, you can specify the following resource filters:

- **resources:** select resources by names, namespaces, kinds, label selectors, annotations, and namespace selectors
- **Subjects:** select users, user groups, and service accounts
- **roles:** select namespaced roles
- **clusterRoles:** select cluster-wide roles

When using resource as a resource filter, it is mandatory to use the kind attribute. You can, however, use wildcards (* and ?) in the kind attribute. By using a combination of these resource filters, you can be more selective as to which resources you wish to process.

This example matches all pods with the label app set to critical, excluding those created by the clusterRole cluster-admin or by the user John:

```
spec:
  rules:
    - name: match-critical-except-given-rbac
      match:
        any:
        - resources:
            kind:
            - Pod
            selector:
              matchLabels:
                app: critical
      exclude:
        any:
        - clusterRoles:
          - cluster-admin
        - subjects:
          - kind: User
            name: John
```

As mentioned before, there are four types of rules that can be used with Kyverno, each with a different use case.

The verify images rule is still a beta feature for the time being, which the maintainers of Kyverno themselves say is not yet ready for production environments and may be subject to major changes.

If the generate and mutate rules are used, new resources will be created automatically and resources will be changed. Within ACA, this is not desirable, as they want the code base to fully match the environment.

For these reasons, I mainly focused on the validation rules, as they will be used the most within ACA.

The validate rules are used to validate new or existing resources. A typical validation rule specifies the mandatory properties, that must be present when creating or updating a given resource. When the given resource doesn't comply with the mandatory properties, a violation occurs. The action that occurs when a violation is detected depends on the value defined in the validation failure action. There are two options:

- **Audit:** Each policy violation is logged in a policy report or cluster policy report, but resource creation or update is permitted.

- **Enforce:** When a violation occurs, the resource creation or update is blocked.

There is an option to override the validation failure action of a policy or cluster policy for each individual namespace.

When using the validation rules, you have a few options. :

- **Pattern:** To pass the validation rule, the matched resources must include all fields and expressions defined in the pattern. For example, making sure the number of replicas set in a deployment is greater than or equal to two.

- **Deny:** This can be used to deny a request when the conditions written as expressions are met. For example, block the deletion of specific resources.

- **Foreach:** Allows looping over multiple elements. For example, looping over all containers in a pod.

- **Manifest validation:** Makes it possible to check the signatures of YAML manifest files signed with Sigstore. For example, check if a YAML manifest file has changed by verifying the signature.

- **Pod security:** Simplifies the process of validating if PSS are configured. For example, validating that pods in a specific namespace comply with the baseline level of PSS.

Like OPA Gatekeeper, Kyverno also has the ability to validate existing resources. This can be achieved by using the background attribute, which is enabled by default.

Existing resources that are validated by a background scan aren't deleted when a violation occurs, even if the validation failure action is set to enforce. Updating these resources is possible as long as they continue to violate the policy. However, when the violation is eventually fixed, new updates triggering a violation will be blocked.

Here is an example cluster policy that makes sure that the label app is present in all pods in the K8s cluster.

```yaml
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: require-labels
  annotations:
    policies.kyverno.io/title: Require Labels
    policies.kyverno.io/category: Best Practices
    policies.kyverno.io/minversion: 1.6.0
    policies.kyverno.io/severity: medium
    policies.kyverno.io/subject: Pod, Label
    policies.kyverno.io/description: >-
      Define and use labels that identify semantic attributes of your application or Deployment.
      A common set of labels allows tools to work collaboratively,
      describing objects in a common manner that
      all tools can understand. The recommended labels describe applications in a way that can be
      queried. This policy validates that the label `app` is specified with some value.
spec:
  validationFailureAction: Audit
  background: true
  rules:
    - name: check-for-labels
      match:
        any:
          - resources:
              kinds:
                - Pod
      validate:
        message: "The label `app` is required."
        pattern:
          metadata:
            labels:
              app: "?*"
```

2.4.2.2 Policy reports

As stated previously, you can set the validation failure action to audit. When a violation occurs against a policy with this setting, the resources that triggered the violation are blocked from being created or updated, instead, it triggers Kyverno to generate a policy and cluster policy report. Background scans that detect violations also trigger the generation of policy and cluster policy reports.

Each policy that triggers a violation generates a new policy and a cluster policy report. A policy report for a specific policy contains all violations that occurred within the namespace in which the report is generated. A cluster policy report contains violations against a specific policy for the entire cluster.

When a resource is deleted, the violation of that resource is removed from the reports. This way, the reports only contain current policy violations.

Below you find an example of a cluster policy report for the cluster policy named require-ns-labels. In the report, you can see every resource that committed a violation against the specific cluster policy. In this case, only the namespace kube-node-lease committed a violation because the label team was not defined.

```
---
apiVersion: wgpolicyk8s.io/v1alpha2
kind: ClusterPolicyReport
metadata:
  creationTimestamp: "2022-10-18T11:55:20Z"
  generation: 1
  labels:
    app.kubernetes.io/managed-by: kyverno
  name: cpol-require-ns-labels
  resourceVersion: "950"
  uid: 6dde3d0d-d2e8-48d9-8b56-47b3c5e7a3b3
results:
  - category: Best Practices
    message: 'validation error: The label `team` is required. rule check-for-ns-labels
    failed at path /metadata/labels/team/'
    policy: require-ns-labels
    resources:
      - apiVersion: v1
        kind: Namespace
        name: kube-node-lease
        uid: 06e5056f-76a3-461a-8d45-2793b8bd5bbc
    result: fail
    rule: check-for-ns-labels
    scored: true
    severity: medium
    source: kyverno
    timestamp:
      nanos: 0
      seconds: 1666094105
```

2.4.2.3 Policy library

Just like OPA Gatekeeper, Kyverno also has a policy library that is maintained by the community. The vast library contains policies of every type, but mostly validation policies. Almost all policies in this library have the validation failure action set to audit by default. This way, you can test the policies without causing damage to your K8s cluster.

2.4.3 OPA Gatekeeper vs Kyverno

Gatekeeper is the K8s specific implementation of OPA, which is a general-purpose policy engine. Kyverno is also a policy engine, but this one was designed to be used in K8s from the start. Both tools have a lot of features, some of which are available in both tools. But Kyverno has more features and capabilities.

Now that I've researched both tools, it's time for a comparison between the two to see which one is the better fit.

Features	Gatekeeper	Kyverno
Validation	Yes	Yes
Mutation	Yes	Yes
Generation	No	Yes
Image verification	No (only via extension)	Yes
Image registry lookup	No (only via extension)	Yes
Extensions	Yes	No
Metrics exposed	Yes	Yes
OpenAPI validation schema (kubectl explain)	No (not available with all CRDs)	Yes
…	…	…

Extra info	Gatekeeper	Kyverno
CNCF status	Graduated	Incubation
Github (stars, forks, commits)	3100, 671, 1299	3900, 576, 5899
Sample policies in library	46	265
Birth (age)	July 2017 (5 years, 10 months)	May 2019 (4 years)

Advantages

Gatekeeper

- Capable of more complex policies
- Maturer

Kyverno

- Kubernetes-style policy framework
- More sample policies are available
- More features and capabilities
- OCI image supply chain security

Disadvantages

Gatekeeper

- Requires learning a new program language to write policies
- OPA constraint framework

Kyverno

- Younger, but fast-growing community
- Highly complex policies aren’t possible

Verdict

As you can see in the features table, Kyverno has more features than Gatekeeper. And when we add the pros and cons, the gap between the two tools becomes even clearer.

Kyverno might be younger than Gatekeeper, but it’s far easier to use because of its Kubernetes-style policy framework, and the larger policy library comes in handy during the implementation.

2.5 Scanning images for CVEs with Trivy

Trivy is an open-source security scanner designed for containerized environments, made by Aqua Security. It can be used to scan for vulnerabilities, IaC misconfigurations, Kubernetes security risks, etc. Trivy is mainly used as a CLI tool, but there is also a K8s operator that can automatically perform the same scans.

2.5.1 Trivy CLI

With the CLI implementation of Trivy, you can scan container images for known vulnerabilities by referencing vulnerability databases and various security advisories. It checks for vulnerabilities in packages, application dependencies, and embedded libraries within images. Trivy CLI can scan multiple locations: local images, remote registries, and K8s clusters. Because it is a CLI tool, you have to manually start each scan, but you could also create a script that periodically takes scans.

When scanning the latest Nginx alpine image for vulnerabilities without any other options, the command looks like this: "trivy image -scanners vuln nginx:1.23.4-alpine".

The output of the command above is shown in the picture below. As you can see, there are a few vulnerabilities in this image. The output shows the severity of the vulnerabilities and the version in which they are fixed.

nginx:1.23.4-alpine (alpine 3.17.3)

Total: 13 (UNKNOWN: 0, LOW: 2, MEDIUM: 10, HIGH: 1, CRITICAL: 0)

Library	Vulnerability	Severity	Installed Version	Fixed Version	Title
curl	CVE-2023-28319	MEDIUM	7.88.1-r1	8.1.0-r0	use after free in SSH sha256 fingerprint check https://avd.aquasec.com/nvd/cve-2023-28319
	CVE-2023-28321				IDN wildcard match may lead to Improper Cerificate Validation https://avd.aquasec.com/nvd/cve-2023-28321
	CVE-2023-28322				more POST-after-PUT confusion https://avd.aquasec.com/nvd/cve-2023-28322
	CVE-2023-28320	LOW			siglongjmp race condition may lead to crash https://avd.aquasec.com/nvd/cve-2023-28320
libcrypto3	CVE-2023-1255	MEDIUM	3.0.8-r3	3.0.8-r4	Input buffer over-read in AES-XTS implementation on 64 bit ARM https://avd.aquasec.com/nvd/cve-2023-1255
libcurl	CVE-2023-28319		7.88.1-r1	8.1.0-r0	use after free in SSH sha256 fingerprint check https://avd.aquasec.com/nvd/cve-2023-28319
	CVE-2023-28321				IDN wildcard match may lead to Improper Cerificate Validation https://avd.aquasec.com/nvd/cve-2023-28321
	CVE-2023-28322				more POST-after-PUT confusion https://avd.aquasec.com/nvd/cve-2023-28322
	CVE-2023-28320	LOW			siglongjmp race condition may lead to crash https://avd.aquasec.com/nvd/cve-2023-28320
libssl3	CVE-2023-1255	MEDIUM	3.0.8-r3	3.0.8-r4	Input buffer over-read in AES-XTS implementation on 64 bit ARM https://avd.aquasec.com/nvd/cve-2023-1255
libwebp	CVE-2023-1999	HIGH	1.2.4-r1	1.2.4-r2	Double-free in libwebp https://avd.aquasec.com/nvd/cve-2023-1999
libxml2	CVE-2023-28484	MEDIUM	2.10.3-r1	2.10.4-r0	NULL dereference in xmlSchemaFixupComplexType https://avd.aquasec.com/nvd/cve-2023-28484
	CVE-2023-29469				Hashing of empty dict strings isn't deterministic https://avd.aquasec.com/nvd/cve-2023-29469

2.5.2 Trivy operator

To integrate vulnerability scanning into K8s environments, the Trivy operator, was created. It's an extension of the Trivy CLI. With the Trivy operator you don't have to worry about starting your scans because the operator takes care of this for you. The Trivy operator will automatically start a scan when a K8s resource is created or updated. Trivy will then create CRDs that contain the scan results. When it comes to vulnerability results, each replicaSet, daemonSet, job, statefulSet, and pod will get its own vulnerability report.

Here is an example of a vulnerability report for a replicaSet in the demo namespace:

```
apiVersion: aquasecurity.github.io/v1alpha1
kind: VulnerabilityReport
metadata:
  annotations:
    trivy-operator.aquasecurity.github.io/report-ttl: 24h0m0s
  creationTimestamp: "2023-05-24T18:21:25Z"
  generation: 2
  labels:
    resource-spec-hash: 5d485fbfb4
    trivy-operator.container.name: nginx
    trivy-operator.resource.kind: Pod
    trivy-operator.resource.name: nginx
    trivy-operator.resource.namespace: demo
  name: pod-nginx-nginx
  namespace: demo
  ownerReferences:
  - apiVersion: v1
    blockOwnerDeletion: false
    controller: true
    kind: Pod
    name: nginx
    uid: 07cef3f4-71a2-45ea-8498-1c91018d60f1
  resourceVersion: "26001383"
  uid: 619d4804-477d-46e2-a6c9-dabd23d1d7fb

report:
  artifact:
    repository: library/nginx
    tag: 1.23.4-alpine
  registry:
    server: index.docker.io
  scanner:
    name: Trivy
    vendor: Aqua Security
    version: 0.39.0
  summary:
    criticalCount: 0
    highCount: 1
    lowCount: 2
    mediumCount: 10
    noneCount: 0
    unknownCount: 0
  updateTimestamp: "2023-05-24T18:21:48Z"
  vulnerabilities:
  - fixedVersion: 8.1.0-r0
    installedVersion: 7.88.1-r1
    links: []
    primaryLink: https://avd.aquasec.com/nvd/cve-2023-28319
    resource: curl
    score: 5.9
    severity: MEDIUM
    target: ""
    title: use after free in SSH sha256 fingerprint check
    vulnerabilityID: CVE-2023-28319
```

In this part of the vulnerability report, you can find information about the resource that was scanned.

The part that actually contains the vulnerabilities that were found can be found here. You can find the image that was scanned, a summary of the CVEs that were found, and a section with details about each CVE.

2.5.3 Trivy CLI vs Trivy operator

There are a few differences between the CLI and operator versions of Trivy. Here is a table to better visualize the differences.

Advantages

Trivy CLI

- Can block containers from launching when a CVE is detected.
- Output directly in the terminal or as an output file.

Trivy operator

- Automated scans
- Can detect changes in in-cluster resources
- Runs inside the cluster,

Disadvantages

Trivy CLI

- Manual scans
- Runs outside the cluster and is a security risk.

Trivy operator

- Can’t block containers from launching when a CVE is detected.

Verdict

When looking at the pros and cons of both versions of Trivy, it is clear that the operator is the best choice. This way, we can automate our scans. We can also scan already existing resources, this will come in handy when implementing Trivy in a K8s cluster that is already populated with resources. The only major downside to using the operator version of Trivy is that we can’t block containers from launching when they contain CVEs. But the benefits outweigh the drawbacks.

2.6 Compliance control with Kube-bench/Kubescape

2.6.1 Kube-bench

Kube-bench is an open-source security auditing tool designed to check the security configuration of K8s clusters. It scans the cluster's configuration and compares it against industry best practices and security benchmarks, such as the CIS Kubernetes Benchmark. Tests in Kube-bench are configured in YAML files.

Because tests or checks in Kube-bench are configured in YAML files and these files are loaded at the start of the container, it is possible to add some custom checks. By adding some additional checks in a YAML file and changing the configuration of the Kube-bench job to include custom checks, they can be imported.

2.6.1.1 Running Kube-bench

Kube-bench can be run in a few different ways.

- In terminal: As a CLI tool in terminal. You may need root or sudo rights to have access to all the configuration files.
- Inside a container: Avoid installing Kube-bench on the host by running it inside a container. It needs a host PID namespace and mounting of /etc and /var.
- Job inside a K8s cluster: The maintainers defined a job that can be run inside a K8s cluster to test the compliance of the cluster.
- Inside EKS, AKS, and GKE clusters: Apart from a job for a default K8s cluster, there are also jobs made that can be used to run inside managed K8s clusters such as EKS, AKS, and GKE.

2.6.1.2 Scan results

After running the scan job made for the EKS cluster, the result is visible in the logs of the pod that executed the scan. In the picture below, you can see the results of the scan job. First is a list of all checks that were performed by Kube-bench and the status (INFO, PASS, WARN, or FAIL) of the scan. After this list, you get more detail on each check that resulted in a status of WARN or FAIL. The exact reason for the status, however, is not explained, nor is the resource on which the check failed.

```
[INFO] 3 Worker Node Security Configuration
[INFO] 3.1 Worker Node Configuration Files
[PASS] 3.1.1 Ensure that the kubeconfig file permissions are set to 644 or more restrictive (Manual)
[PASS] 3.1.2 Ensure that the kubelet kubeconfig file ownership is set to root:root (Manual)
[PASS] 3.1.3 Ensure that the kubelet configuration file has permissions set to 644 or more restrictive
(Manual)
[PASS] 3.1.4 Ensure that the kubelet configuration file ownership is set to root:root (Manual)
[INFO] 3.2 Kubelet
[PASS] 3.2.1 Ensure that the --anonymous-auth argument is set to false (Automated)
[PASS] 3.2.2 Ensure that the --authorization-mode argument is not set to AlwaysAllow (Automated)
[PASS] 3.2.3 Ensure that the --client-ca-file argument is set as appropriate (Manual)
[PASS] 3.2.4 Ensure that the --read-only-port argument is set to 0 (Manual)
[PASS] 3.2.5 Ensure that the --streaming-connection-idle-timeout argument is not set to 0 (Manual)
[PASS] 3.2.6 Ensure that the --protect-kernel-defaults argument is set to true (Automated)
[PASS] 3.2.7 Ensure that the --make-iptables-util-chains argument is set to true (Automated)
[PASS] 3.2.8 Ensure that the --hostname-override argument is not set (Manual)
[WARN] 3.2.9 Ensure that the --eventRecordQPS argument is set to 0 or a level which ensures appropriate
 event capture (Automated)
[PASS] 3.2.10 Ensure that the --rotate-certificates argument is not set to false (Manual)
[PASS] 3.2.11 Ensure that the RotateKubeletServerCertificate argument is set to true (Manual)
[INFO] 3.3 Container Optimized OS
[WARN] 3.3.1 Prefer using Container-Optimized OS when possible (Manual)

== Remediations node ==
3.2.9 If using a Kubelet config file, edit the file to set eventRecordQPS: to an appropriate level.
If using command line arguments, edit the kubelet service file
/etc/systemd/system/kubelet.service.d/10-kubeadm.conf on each worker node and
set the below parameter in KUBELET_SYSTEM_PODS_ARGS variable.
Based on your system, restart the kubelet service. For example:
systemctl daemon-reload
systemctl restart kubelet.service

3.3.1 audit test did not run: No tests defined

== Summary node ==
14 checks PASS
0 checks FAIL
2 checks WARN
0 checks INFO

== Summary total ==
14 checks PASS
0 checks FAIL
2 checks WARN
0 checks INFO
```

2.6.2 Kubescape

Kubescape is an open-source tool designed to assess the security posture of K8s clusters and applications deployed within them. Kubescape includes risk analysis, security compliance, and misconfiguration scans. Kubescape uses multiple frameworks to detect misconfigurations. The included frameworks are: NSA-CISA, MITRE ATT&CK, and CIS Kubernetes Benchmark. Below that which is visible, the tool uses OPA to verify K8s objects. Checks performed by Kubescape are loaded from an online database. It is possible to add extra checks to this database, but not easily.

2.6.2.1 Running Kubescape

Kubescape can be used in two ways: as a CLI in a terminal or CI/CD pipeline, or as an operator inside a K8s cluster. In both cases, you have the option to integrate the ARMO portal, the maker of Kubescape. ARMOsec is a platform that provides security solutions for cloud-native and containerized environments. On this platform, you can visualize your scan results and, when using the operator, even start scans from within the platform. However, there is an option to disable the integration with this platform for people who don't want to expose their K8s clusters to another cloud platform.

2.6.2.2 Scan results

Depending on the implementation of Kubescape, the scan results are either visible on the ARMOsec platform or directly inside the terminal. Here is an example summary of the scan results performed by the CLI implementation of Kubescape:

SEVERITY	CONTROL NAME	FAILED RESOURCES	ALL RESOURCES	% RISK-SCORE
Critical	Disable anonymous access to Kubelet service	0	0	Action Required ****
Critical	Enforce Kubelet client TLS authentication	0	0	Action Required ****
High	Forbidden Container Registries	0	20	Action Required *
High	Resources memory limit and request	0	20	Action Required *
High	Resource limits	13	20	62%
High	Applications credentials in configuration files	0	47	Action Required *
High	List Kubernetes secrets	8	114	7%
High	Host PID/IPC privileges	1	20	4%
High	Writable hostPath mount	1	20	4%
High	Insecure capabilities	1	20	4%
High	HostPath mount	2	20	8%
High	Resources CPU limit and request	0	20	Action Required *
High	Workloads with Critical vulnerabilities exposed to...	0	0	Action Required **
High	Workloads with RCE vulnerabilities exposed to exte...	0	0	Action Required **
High	RBAC enabled	0	0	Action Required ***
Medium	Data Destruction	4	114	4%
Medium	Non-root containers	10	20	50%
Medium	Allow privilege escalation	9	21	44%
Medium	Ingress and Egress blocked	14	31	46%
Medium	Delete Kubernetes events	1	114	1%
Medium	Automatic mapping of service account	29	78	38%
Medium	CoreDNS poisoning	1	114	1%
Medium	Malicious admission controller (mutating)	5	5	100%
Medium	Access container service account	15	63	24%
Medium	Cluster internal networking	4	11	36%
Medium	Linux hardening	10	20	50%
Medium	Configured liveness probe	6	20	24%
Medium	Secret/ETCD encryption enabled	0	0	Action Required ***
Medium	Audit logs enabled	0	0	Action Required ***
Medium	Containers mounting Docker socket	1	20	4%
Medium	Images from allowed registry	0	20	Action Required *
Medium	Workloads with excessive amount of vulnerabilities	0	0	Action Required **
Medium	CVE-2022-0492-cgroups-container-escape	10	20	50%
Low	Immutable container filesystem	10	20	50%
Low	Configured readiness probe	6	20	24%
Low	Malicious admission controller (validating)	5	5	100%
Low	Network mapping	4	11	36%
Low	PSP enabled	0	0	Action Required ***
Low	Naked PODs	1	23	4%
Low	Image pull policy on latest tag	1	20	4%
Low	Label usage for resources	13	20	62%
Low	K8s common labels usage	3	20	12%
	RESOURCE SUMMARY	58	282	10.18%

In this summary above, a lot of information is visible:

- The severity
- The description of the check
- The number of resources that failed the check
- The total number of resources that were checked
- The risk score, if available. When no risk score is available, it states that action is required. Further down in the results is a legend explaining why extra action is required.

The summary is printed at the end. Before the summary, there are detailed scan results for each K8s resource in the cluster. Here is an example of the scan results of a serviceaccount inside the ingress-nginx namespace. The link that is present can give even more information about the result.

```
################################################################################
ApiVersion:
Kind: ServiceAccount
Name: ingress-nginx
Namespace: ingress-nginx

Controls: 10 (Failed: 2, action required: 0)

+----------+--------------------------------+-----------------------------------+-----------------------------------------+
| SEVERITY |          CONTROL NAME          |                DOCS               |          ASSISTANT REMEDIATION          |
+----------+--------------------------------+-----------------------------------+-----------------------------------------+
| High     | List Kubernetes secrets        | https://hub.armosec.io/docs/c-0015 | relatedObjects[1].rules[1].resources[2] |
|          |                                |                                   | relatedObjects[1].rules[1].verbs[0]     |
|          |                                |                                   | relatedObjects[1].rules[1].verbs[1]     |
|          |                                |                                   | relatedObjects[1].rules[1].verbs[2]     |
|          |                                |                                   | relatedObjects[1].rules[1].apiGroups[0] |
|          |                                |                                   | relatedObjects[0].subjects[0]           |
|          |                                |                                   | relatedObjects[0].roleRef.name          |
+----------+--------------------------------+-----------------------------------+-----------------------------------------+
| Medium   | Access container service       | https://hub.armosec.io/docs/c-0053 |                                         |
|          | account                        |                                   |                                         |
+----------+--------------------------------+-----------------------------------+-----------------------------------------+
```

2.6.3 Kube-bench vs Kubescape

Kube-bench and Kubescape can both perform compliance checks on a running EKS cluster. But they differ in the way they can be used and which frameworks are used to check the clusters compliance.

Advantages

Kube-bench

- It is easy to add extra checks by adding checks in YAML files

Kubescape

- K8s operator can automate compliance scans
- Includes more frameworks
- Very detailed scan results

Disadvantages

Kube-bench

- Less detailed scan results
- Jobs need to be run manually
- Fewer frameworks are included

Kubescape

- Harder to add custom checks

Verdict

The tool that comes out on top is Kubescape, which has far more benefits than Kube-bench while having fewer drawbacks. However, Kube-bench can come in handy when setting up a K8s cluster as it can perform very quick scans. For implementation inside a running cluster, Kubescape is better because of its automated scans.

2.7 Runtime threat detection

When it comes to runtime threat detection tools that are used in K8s, there's really only one option, and that's Falco. No other tool can perform this task the way Falco does.

2.7.1 Falco

Falco is an open-source runtime security tool designed specifically for containerized environments, including K8s. It uses a rules-based engine and kernel-level instrumentation to monitor and detect abnormal or suspicious behavior within containers in real-time.

Falco can detect and alert to any behavior that involves making Linux system calls. These system calls are intercepted on the host by a driver. These system calls are then analyzed and evaluated against rules in the library. When a suspicious or unwanted event is detected, it is alerted to the outputs that are configured.

2.7.1.1 Interception system calls

Falco works by intercepting Linux system calls from the host system at runtime. These system calls can't be intercepted by default, a driver is needed in order to consume this data. Currently, Falco supports three different drivers:

- A kernel module
- An eBPF probe
- A ptrace(2) userspace program

The kernel module approach is the most common and quickest way to run Falco. But because we need to load a piece of code directly inside the Linux kernel, we need administrative privileges on the host system.

When accessing and modifying the Linux kernel is not desirable, the best approach is the eBPF probe. This attaches some custom code to specific kernel functions, such as system calls. This can all happen without the need for administrative privileges on the host system.

When both previous options are not possible, the third option becomes the most viable. A userspace program can attach to a target process, monitor its execution, and intercept system calls, but it introduces additional overhead and context switching between the program and the traced program.

Verdict

The OpsKlaar team would like to make as few changes as possible to the host system that runs the actual K8s cluster via EKS on AWS. When considering this and the available options, the eBPF probe is the most viable option. This allows us to use Falco without any overhead and doesn't require administrative privileges on the host system.

2.7.1.2 Rules

As mentioned, Falco uses a rules-based engine to analyze the intercepted system calls for abnormal or suspicious behavior. A Falco rule file is written in YAML and can contain three types of elements:

- **Rules:** These define the conditions under which an alert should be triggered.
- **Macros:** Rule conditions that can be used inside other macros or in rules.
- **Lists:** Collection of items. Can be used inside other lists or in macros.

Besides these three elements, a Falco rule file may also contain two other elements that are related to version control: required_engine_version and required_plugin_versions.

The most important part of a Falco rule file is obviously the rule itself. A rule consists of four parts: description, condition, output, and priority.

The description describes what the function of the rule is. The condition is an expression evaluated against system calls to see whether an alert should be sent. The output is sent together with the alert to the output selected. And the priority can give an indication of how serious the event actually is. Here is a list of all supported priorities:

- EMERGENCY
- ALERT
- CRITICAL
- ERROR
- WARNING
- NOTICE
- INFORMATIONAL
- DEBUG

When deploying Falco inside a K8s environment, Falco comes equipped with a default library of rules. The library contains 78 rules, of which most (57) are enabled by default.

2.7.2 Falcosidekick

Falco has a sidekick that connects Falco to your ecosystem via a simple daemon. The main reason for using the sidekick is being able to choose different or multiple outputs. The sidekick takes in the suspicious events from Falco and forwards them to the configured outputs. You only need one sidekick for your entire ecosystem because the sidekick can handle multiple Falco instances at once.

Here is a list of some outputs that can be configured in the sidekick:

- Slack
- Teams
- Datadog
- Prometheus
- AlertManager
- Opsgenie
- Elasticsearch
- Grafana
- AWS S3
- GCP Storage
- AWS Lambda
- Kubeless
- GCP Cloud Run
- OpenFaaS
- Azure Event Hubs
- SMTP
- Falcosidekick WebUI
- …

3 REALIZATION PHASE

Prior to the actual realization and implementation, I determined the order in which I would implement the tools. I have determined this order based on the amount of work it would take to implement each tool and how much each tool actually contributes to the environment and the team.

I used the following table to create the implementation order:

Tool	Effort	Time	Profit	Profit score	Order
Secret encryption in etcd	- Create KMS key - Enable encryption - Test combination with sealed secrets	**6h**	- Extra security layer with low cost	**5**	**2**
PSS & PSA	- Few lines of code	**4h**	- Pod security with little control	**3**	**7**
Kyverno	- Deploy Kyverno - Add policies	**4d**	- Pod security with a lot of control - Policy control	**9**	**1**
Trivy	- Deploy operator - Test each scanner	**2d**	- CVE scanning - Config scanning - Detect exposed secrets	**7**	**3**
Kube-bench	- Automate cronjob	**6h**	- Periodic compliance checks	**4**	**6**
Kubescape	- Deploy operator -	**6h**	- Automated compliance checks	**4**	**5**
Falco	- Research Bottlerocket support - Deploy Falco - Add rules - Deploy Falcosidekick	**4d**	- Threat detection on container level	**7**	**4**

The implementation order looked like this:

1. Kyverno
2. Secret encryption etcd
3. Trivy
4. Falco
5. Kubescape
6. Kube-bench
7. PSS & PSA

I ended up not implementing the last two tools. Some tools took more time than originally expected. Because of this, I didn't have enough time left for the last two tools.

3.1 Helm charts

When deploying tools in a K8s cluster, there are a few options. One of the more popular options is using Helm. Helm is a package manager for K8s that simplifies the deployment, management, and versioning of containerized applications and their dependencies through the use of charts. When deploying an application using Helm charts, most steps are the same across applications.

Here is an example of how to use Helm charts to deploy an application:

Steps

1. Add the helm repo if you haven't done so already and update your Helm repo's

```
helm repo add <REPO_NAME> <CHART_URL>
helm repo update
```

   - `<REPO_NAME>` : The name you want to give to this repository
   - `<CHART_URL>` : The location url where the chart can be found
2. Check which Helm chart version provides the desired release for the application

```
helm search repo <REPO_NAME> --versions
```

   Replace `<REPO_NAME>` : The name you want to give to this repository
3. Fetch the current values.yaml

```
helm show values <REPO_NAME>/<APPLICATION_NAME> --version <CHART_VERSION>
> values.yaml.temp
```

   - `<REPO_NAME>` : The name of this repo chosen in step 1
   - `<APPLICATION_NAME` : Name of the application you want to deploy
   - `<CHART_VERSION>` : The Helm chart version chosen in step 2
4. Compare `values.yaml.temp` with the `values.yaml` file in this repo and update it accordingly
5. Run the following command, but replace some parameters:

```
helm --version <CHART_VERSION> --kube-version=<KUBE_VERSION> template
"<APPLICATION_NAME>" <REPO_NAME>/<APPLICATION_NAME> --include-crds -f values.yaml
```

   - `<REPO_NAME>` : The name of this repo chosen in step 1
   - `<APPLICATION_NAME` : Name of the application you want to deploy
   - `<CHART_VERSION>` : The Helm chart version chosen in step 2
   - `<KUBE_VERSION>` : The Kubernetes version of the K8s cluster
6. Merge the resulting resources with the ones in git

3.2 Bash deploy script

After I used Helm to generate a manifest file with the resources needed to create the application, I separated the resources into a file for each resource kind. This provided a better overview of all the resources that needed to be created for this application.

When it was time to deploy the application inside the EKS cluster, I created a deploy script using Bash, like the ones used by the OpsKlaar team to deploy most of their applications and tools. The main reason for this script is to be able to change variables between test and production environments. But the script can be useful when testing applications.

This way, I could make some changes to the values.yaml file and generate a new manifest file with Helm templating. After passing the changes into their corresponding resource files, I could then use the deploy script again to apply the changes made in the values.yaml file.

I created a deploy script for every application that I configured with the help of Helm charts.

- Kyverno
- Trivy
- Falco & Falcosidekick

3.3 Kyverno

The only supported option to deploy Kyverno in a production environment, was via Helm charts. Because this tool would later be implemented in a production environment I also chose the Helm chart option.

3.3.1 Helm

These are the final changes made to the values.yaml file:

- Changing the namespace Kyverno would be deployed in because Kyverno needs its own namespace.
- Setting the image pull policy to always makes sure the image used is always the latest iteration of that version.
- Disabling the cleanup controller. The cleanup function in Kyverno is still an alpha feature, therefore, this feature may go through breaking changes. Because of this, the features isn't yet ready for production environments.

3.3.2 Policies

In my research, I mentioned that Kyverno has a large library filled with policies. I went through this list of policies and split them into four groups:

- **Must-have:** policies that must be used
- **Should-have:** policies that should be implemented in the near future
- **Could-have:** policies of less importance that could be implemented in the far future
- **Won't-have:** policies that have no use in our environment

The must-have and should-have groups mostly contain policies from the best practices and pod security categories, as these are the ones that contribute the most to the security in the EKS cluster.

3.3.3 Best practices

Kyverno has some security best practices documented for people using the tool.

Kyverno pods default configuration complies with the security best practices of K8s and the restricted profile of the PSS.

It's also recommended to create network policies to restrict the traffic in the Kyverno namespace. The following communication should be allowed while all the rest should be restricted:

- Ingress to port 9443 from K8s API server
- Ingress to port 9443 from the host
- Ingress to port 8000 if you want to collect metrics
- Egress to the API server if you want to use the API Call feature

Furthermore, Kyverno also recommends using the PSS and best practices policy sets at a minimum.

3.4 Secret encryption in etcd with KMS

3.4.1 Sealed Secrets

The OpsKlaar team uses a tool called Sealed Secrets to securely store and distribute secrets in a GitOps workflow in K8s.

When using sealed secrets inside a K8s cluster, a CLI tool called Kubeseal is used to encrypt secrets into sealed secrets. These sealed secrets can then be safely stored inside Git repositories. When deploying these sealed secrets, the sealed secrets controller decrypts them into secrets that can be used by K8s. When a sealed secret is deleted from the cluster, the corresponding secret is also deleted.

3.4.2 Enabling secret encryption

To enable the encryption of secrets stored in etcd, I had to make some changes to the EKS cluster and create a KMS key. Because ACA uses Terraform to provision its environments, I had to make changes to these files.

To create a KMS key during the pipeline, I created a Terraform file. The KMS key created adheres to the requirements set by AWS to enable secret encryption.

```
resource "aws_kms_key" "eks_encryption" {
  description             = "KMS key for eks encryption"
  is_enabled              = true
  key_usage               = "ENCRYPT_DECRYPT"
  customer_master_key_spec = "SYMMETRIC_DEFAULT"
  enable_key_rotation     = true
  deletion_window_in_days  = 30
}
```

Next, I had to make sure that the right IAM users and roles could administer the key and that the IAM role of the control plane could use the key for encrypt and decrypt actions. For this, I created an IAM policy document. In the first statement, I made sure a key administrator was defined.

```
data "aws_iam_policy_document" "eks_encryption" {
  statement {
    sid       = "Allow access for key Administrators"
    effect    = "Allow"
    actions   = [
      "kms:Create*",
      "kms:Describe*",
      "kms:Enable*",
      "kms:List*",
      "kms:Put*",
      "kms:Update*",
      "kms:Revoke*",
      "kms:Disable*",
      "kms:Get*",
      "kms:Delete*",
      "kms:TagResource",
      "kms:UntagResource",
      "kms:ScheduleKeyDeletion",
      "kms:CancelKeyDeletion"
    ]
    resources = [
      "*"
    ]
    principals {
      identifiers = [
        "<IAM ROLES>"
      ]
      type        = "AWS"
    }
  }
```

The next statement made sure the IAM role of the control plane could access and use the KMS key.

```
  statement {
    sid       = "Allow use of the key"
    effect    = "Allow"
    actions   = [
      "kms:Encrypt",
      "kms:Decrypt",
      "kms:ReEncrypt*",
      "kms:GenerateDataKey*",
      "kms:DescribeKey"
    ]
    resources = [
      aws_kms_key.eks_encryption.arn
    ]
    principals {
      identifiers = [
        "<IAM ROLES>"
      ]
      type        = "AWS"
    }
    condition {
      test     = "StringEquals"
      values   = [
        "kms:EncryptionContext:aws:eks:cluster-name"
      ]
      variable = "${var.project}-${var.environment}"
    }
  }
}
```

This IAM policy needs to be applied to the KMS key we created a few steps ago.

```
resource "aws_kms_key_policy" "eks_encryption_eks_encryption" {
  key_id = aws_kms_key.eks_encryption.id
  policy = data.aws_iam_policy_document.eks_encryption.json
}
```

The last resource I needed to change was the EKS cluster. Encryption needs to be enabled on the EKS cluster. The cluster needs to know which KMS key to use for the encryption and what it needs to encrypt. Currently, only secret encryption is supported.

```
enabled_cluster_log_types = [
  "api",
  "audit",
  "authenticator",
  "controllerManager",
  "scheduler"
]
```

When running the modified Terraform code, our EKS cluster should be configured to encrypt secrets inside etcd.

3.4.3 Strange behavior

In my research phase, I made a cost estimate based on the number of requests for the KMS key. After enabling the secret encryption, I noticed some weird behavior on the KMS key. There was a lot more action on the KMS key than originally expected. At first, I thought that this must be because of the Sealed Secrets controller, but I couldn't find any indication that this was the case. After a while, I stopped searching for a solution because I had already spent too much time searching for this problem and I still had other tools to implement.

3.5 Trivy operator

In the research phase, I determined that I would use the operator version of Trivy. The best way to deploy the operator is via Helm charts. The steps are mostly the same as I those used with Kyverno.

3.5.1 Helm

This time I did modify a few more settings inside the values.yaml.

- I again changed the namespace in which the Trivy operator would be deployed because this is one of the best practices.
- Setting the image pull policy to always makes sure the image used is always the latest iteration of that version.
- Disabled the config audit scanner. The task this scanner performs is already performed by Kyverno.
- Disabled the RBAC assessment scanner. Kyverno can restrict how roles and cluster roles can be configured. It wouldn't be useful if Trivy would scan these resources after they've already been validated by Kyverno.
- Disabled the infra assessment scanner because this scanner doesn't include any meaningful checks for managed K8s clusters, such as an EKS cluster.
- Disabled the compliance scanner because it doesn't provide us with much detail when a compliance check fails. Kubescape does this task better.
- I enabled the metrics for the vulnerability scanner. Currently, this one does not have much benefit because the metrics are not yet collected by any tool, but this will be the case in the future.

3.6 Falco & Falcosidekick

The implementation of Falco contains two parts: the Falco instance itself and a Falcosidekick that will be responsible for sending the alerts to the chosen output.

3.6.1 Falco instance

As stated in my research about Falco, Falco needs a driver to be able to intercept the Linux system calls from the host system. There are a few driver options for Falco, but the one I decided to use is the eBPF probe, so I didn't have to change any parts of the host file system. Falco provides a list of operating systems where you can find out if there is a driver specifically for your OS and the kernel version used by this OS. ACA uses Bottlerocket as the host OS on each node. The specific version of Bottlerocket used is included in this list, so there is on less thing to worry about.

3.6.1.1 Helm

The recommended way to install Falco in a K8s environment is through the use of Helm charts. Again, the same steps listed in the Helm section are used.

Here is a list of settings I modified in the values.yaml file:

- Set the image pull policy to always for all container images to make sure the latest iteration of the image is used when deploying a new Falco instance.
- Changing the namespace Falco will be deployed in its own namespace.
- Modifying the driver to use the eBPF probe the container initializing the eBPF probe by default has all privileges. This container doesn't need all these privileges, so I enabled the setting to enable the least privileged mode.
- Falco has the option to include custom rules. To be able to add custom rules later, I created an example file with an example rule. By doing this, Falco makes everything ready if somebody wants to add custom rules later on.
- With the default configuration, Falco doesn't alert to any output. I enabled the JSON output and also enabled and configured the HTTP output. This HTTP output would later be used to send events to the Falcosidekick for alerting.

3.6.1.2 Rules

Falco comes built with standard rules included in its deployment. I left the default included rules enabled to get more insight into unwanted or dangerous processes. All these rules can be disabled at once or one by one.

To disable all rules at once, you need to change the values.yaml file.

```
rules_file:
  - /etc/falco/falco_rules.yaml
  - /etc/falco/falco_rules.local.yaml
  - /etc/falco/rules.d
```

The first line under the rules_file element must be deleted. Then generate a new manifest fill, pass the changes through to each resource file, and run the deploy script.

To disable each rule individually, you need to add a few lines to the configmap named falco-rules. This configmap is generated by the Helm template command.

In this picture, you can see an example of how to disable a default-enabled rule or how to enable a rule that is disabled by default. You only need to know the name of the rule you wish to enable or disable and change the file.

```
default-rules.yaml: |-
  ######################################################################
  ##                        Default Rules                             ##
  ######################################################################
  # Example how to disable default enabled rules
  # - rule: Terminal shell in container
  #   enabled: false
  #
  # Example how to enable default disabled rules
  # - rule: Disallowed SSH Connection
  #   enabled: true
  #
```

After the configmap is modified, you need to run the deploy script again because Falco only loads its rules when initializing the container and currently can’t load new rules when already running.

3.6.2 Falcosidekick

3.6.2.1 Helm

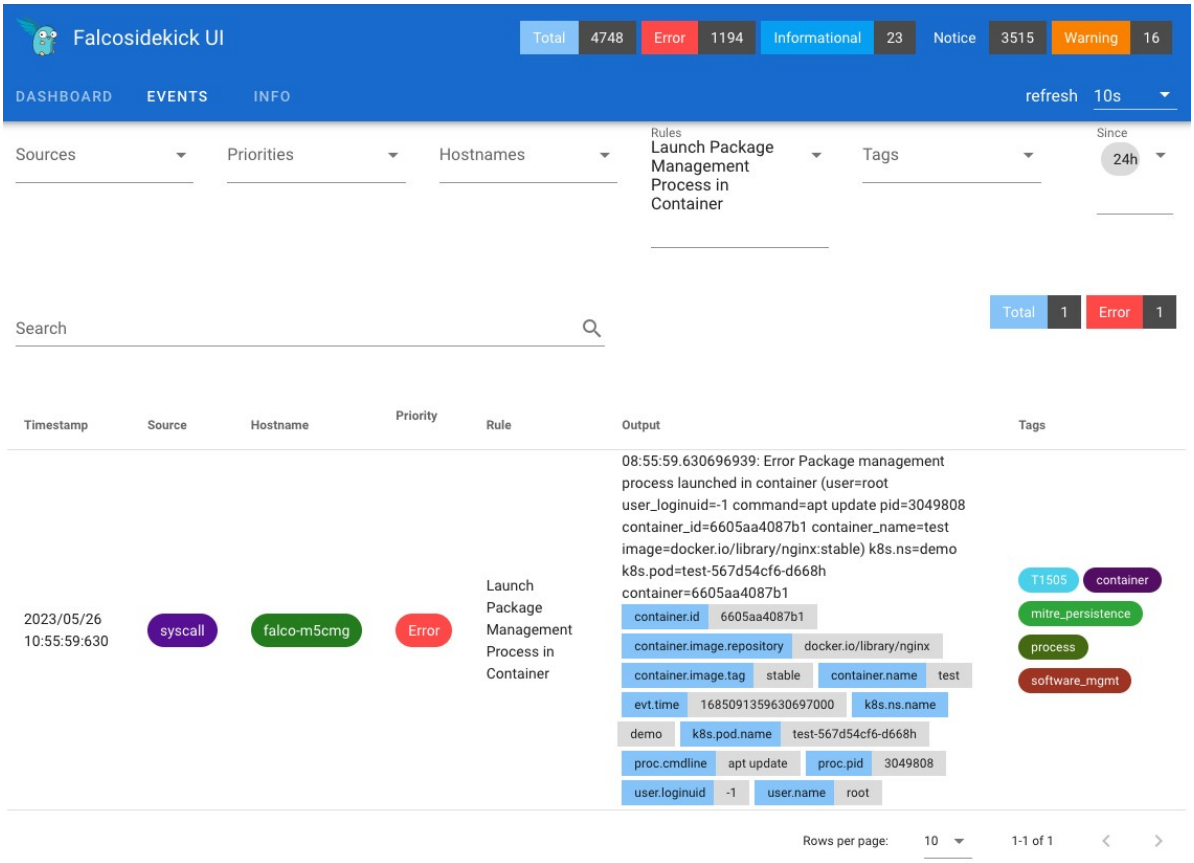
The best way to deploy the Falcosidekick is with the help of Helm charts. The steps are the same as before, except the settings in the values.yaml change.

- Set the image pull policy to always get the latest iteration of the images used for the Falcosidekick.
- Enable the web interface to make it possible to get information when events are triggered by Falco.
- The web interface needs an ingress resource to make it possible to access it via a load balancer.

3.6.2.2 Web interface

After the installation of the Falcosidekick with an integrated web interface, Falco forwards events to the Falcosidekick, and those events are visible on the web interface.

On the web interface, you can find a dashboard with graphs about the events detected by all the Falco instances connected to the sidekick. The events tab is more useful, here you get a list of unwanted or dangerous processes triggered by a rule.



3.7 Kubescape

For running compliance control checks inside a K8s cluster, Kubescape came out on top. With Kubescape, the operator version is preferred over the CLI version because it can perform automatic scans. The only available option to deploy the Kubescape operator is through Helm charts.

3.7.1 Helm

When trying to deploy the Kubescape operator via Helm, I again followed the same steps as mentioned in the Helm section.

When trying to implement Kubescape with its default configuration, I got some errors because the default configuration wasn't complete. Two settings need to be changed. You need to add an account and cluster name in the values.yaml. But because ACA doesn't want to expose its clusters to the Kubescape cloud platform, I searched for a setting to disable this default function.

By changing submit to false in the values.yaml, the account didn't need to be filled in. The cluster name field was still required, so I defined the cluster name and tried again to implement Kubescape. Everything went well, and I thought all the resources needed got deployed.

I tested every option in Kubescape to see if everything was working fine and as expected. Spoiler, this was not the case, I found out there was a deployment missing. The deployment that was responsible for launching the operator pods was missing and wasn't included in the files generated by the Helm template command.

When going through the Helm charts of Kubescape that are available on Github, I found the reason for the missing deployment. In every template directly related to the operator, deployment, network policy, and service, there was one line causing problems.

```
{{- if and .Values.operator.enabled .Values.kubescape.submit }}
```

This was the first line in each of these three template files. This line states that the two settings need to be enabled or set to true, otherwise, these templates would not be used when trying to implement the Kubescape operator via Helm charts.

In the values.yaml file, I enabled the operator, but because I didn't want to expose my EKS cluster to the Kubescape cloud platform, I disabled the submit option. With the current versions of the Helm charts for Kubescape, it wasn't possible for us to deploy Kubescape the way we wanted.

3.7.2 Solution

The operator version of Kubescape isn't an option in our environment unless we change the Helm charts ourselves. Because I had a limited timeframe for my internship, it wasn't possible to achieve this. I made a suggestion to the OpsKlaar team to include the Kubescape CLI in an audit that would be performed periodically.

4 NEXT STEPS

My internship period is coming to an end. I succeeded in implementing all required components into my EKS test environment. Before these tools and components can be used in production environments, the team needs to take some extra steps.

4.1 Kyverno

Kyverno is up and running inside my test environment, and I separated all useful policies into groups. Some groups are more important than others.

Before the team can fully use Kyverno, they need to determine which rules they actually want to use. Some rules may require some tweaking to exclude some resources that would otherwise be blocked by Kyverno when trying to update or create them.

I also suggest that they find a solution to link the policy reports to their monitoring system to make alerting about policy violations easier.

4.2 Secret encryption in etcd with KMS

The encryption is enabled on my EKS cluster. However, there is still some weird and strange behavior happening on the KMS key.

Before the team can enable the encryption of secrets in etcd, they need to do some extra research to be able to make a more representative cost prediction. Because the encryption of secrets can't be disabled, we can't really test this in a production environment. So I suggest they use my test environment or create a new one to further test and research this weird behavior.

4.3 Trivy operator

Trivy is fully implemented in my test environment. The unused or less useful scanners are disabled, and only the ones that are actually needed are enabled and work perfectly.

Like with Kyverno, I suggest linking Trivy to Datadog so that the team receives alerts of vulnerabilities and exposed secrets present in the environment. Because right now they still need to go through the reports to see this information.

4.4 Falco & Falcosidekick

In my environment, I succeeded in installing Falco and the Falcosidekick. Falco can detect unwanted or dangerous processes on the container level and send these events to the sidekick. I used the web interface of the sidekick to visualize the events triggered by Falco.

The team uses Datadog as a monitoring tool for all the applications running inside each environment. I advise the team to disable the webUI of the sidekick and configure the forward function to Datadog. This way, all events are available in one location, giving a better overview of what's happening in all environments.

5 CONCLUSION

I am satisfied with the work I delivered during my internship, and I've learned a lot about how everything works in the cloud industry.

I achieved everything that was listed in my assignment: I introduced a tool for policy enforcement and pod security, a tool for scanning CVEs in images is present in my test environment, Falco can detect unwanted or dangerous processes on the container level, and secrets stored inside etcd are encrypted via a KMS key.

Because every environment is different, some extra changes or research may be needed to make each tool fully production-ready.

6 TERMINOLOGY

6.1 Internship framing

Concept	Description	Content
Internship supervisor (Thomas More)	The lecturer from the education institution which guides the internship student during the internship period	Luc Celis
Internship coordinator (Thomas More)	The administrative assistant from the educational institution which provides the organizational and legal aspects of the internship	Christine Smeets
Internship mentor	The employee from the internship company who is the formal contractual point of contact for the school for all legal aspects of the internship	Jana Garcia Gonzalez
Technical internship mentor	The employee from the internship company who bears ultimate responsibility for the technical content of the internship.	Jan Beerden
Internship student	The student who carries out the internship (= intern)	Wim Hembrechts
Internship company	The company offering the internship to the educational institution and internship student (= internship company)	ACA Group
Internship location	The physical location where internship student performs his tasks	Herkenrodesingel 8b 2.01, 3500 Hasselt
Period	The period within which the internship will take place in accordance with the provisions of the internship contract	27/02/2023 to 26/5/2023
Internship contract	The legally binding document between the educational institution, the internship company and the intern.	

6.2 Technical concepts

Concept	Description
Cloud	Cloud refers to a network of remote servers that provide on-demand access to a variety of computing resources and services, enabling scalable storage, computation, and data management over the internet
AWS	AWS (Amazon Web Services) is a cloud computing platform by Amazon that provides a wide range of on-demand services, including storage, computing power, networking, and databases, accessible over the internet
KMS	KMS (Key Management Service) is a managed service by AWS that enables the creation, management, and protection of cryptographic keys used for data encryption and decryption within the AWS ecosystem
IAM	IAM (Identity and Access Management) is a service in AWS that enables users to manage access and permissions for various AWS resources, allowing granular control over user authentication, authorization, and resource-level permissions.
Container	A container is a lightweight, self-contained executable software package that includes all the necessary code, runtime, system libraries, and settings to run an application
Kubernetes (K8s)	Kubernetes is an open-source container orchestration platform that automates and scales the management of containerized applications
EKS	EKS (Elastic Kubernetes Service) is AWS' managed K8s services that simplifies the deployment and management of K8s clusters
Etcd	Etcd is a distributed key-value store that provides a reliable and highly available way to store and retrieve data, often used as a crucial component for service discovery and coordination in distributed systems like K8s
Pod	A Pod in K8s is the smallest and simplest unit of deployment, representing a single instance of a running process or application, encapsulating one or more containers and shared resources
Secret	A Secret in K8s is an object that securely stores sensitive data, such as passwords or API keys, and allows for secure consumption of this information by authorized applications and services
RBAC	RBAC (Role-Based Access Control) is a security mechanism used in K8s that regulates access to resources in a system based on the roles and permissions assigned to individual users or groups, ensuring control over resource authorization
CRD	CRD (Custom Resource Definition) it is an extension mechanism in K8s that allows users to define and create their own custom resources with associated API endpoints
PSP	PSP (Pod Security Policy) is a security mechanism in K8s that defines a set of rules and restrictions to govern the security context of pods, ensuring adherence to specific security policies and mitigating potential risks
IaC	IaC (Infrastructure as Code) is an approach where infrastructure provisioning and management of tasks are automated through machine-readable code, allowing for consistent, scalable, and repeatable deployment and configuration of resources
Terraform	Terraform is an open-source IaC tool that enables the provisioning and management of cloud resources from various providers through declarative configuration files
CI/CD	CI/CD (Continuous Integration/Continuous Deployment) is a software development practice that involves automating the integration, testing, and deployment of code

	changes to enable frequent and reliable software releases
Git	Git is a distributed version control system that allows multiple developers to collaborate on a codebase by tracking changes, facilitating branch management, and enabling efficient code merging
CNCF	CNCF (Cloud Native Computing Foundation) it is a vendor-neutral organization that promotes the adoption and advancement of cloud-native technologies and best practices in the software development industry
JSON	JSON (JavaScript Object Notation) is a lightweight data interchange format that uses human-readable text to transmit and represent structured data objects consisting of key-value pairs
YAML	YAML (YAML Ain’t Markup Language) is a human-readable data serialization format that is commonly used for configuration files, representing data structures using indentation and simple syntax
Bash	Bash (Bourne Again Shell) is a popular command-line interpreter and scripting language used in Unix-like operating systems for executing commands and automating tasks