ADA LOGICS

# Kyverno 2023 Security Audit Report

In collaboration with the Kyverno project maintainers, The Linux Foundation and the Open Source Technology Improvement Fund


**Prepared by**
Adam Korczynski, Ada Logics
David Korczynski, Ada Logics



**Report version:** 1.0

**Published:** 28th November 2023

This page has intentionally been left blank

# Table of Contents

# Executive summary

In the fall of 2023, Ada Logics conducted a security audit of Kyverno in a coordinated collaboration between Ada Logics, Kyverno, OSTIF and the CNCF. The CNCF funded the work. The security audit was a holistic security audit with the following goals:

1. Assess and formalize a threat model for Kyverno, highlighting entrypoints, risks and at-risk components.
2. Review the Kyverno codebase for security vulnerabilities of any severity.
3. Review Kyvernos fuzzing suite
4. Review Kyverno's supply-chain maturity against SLSA.

To formalize the threat model, Ada Logics relied on three sources of information: 1) Kyverno's official documentation, 2) the Kyverno source tree and 3) feedback from the Kyverno maintainers. The manual review was performed against the threat model to allow the auditors to consider trust levels and threat actors while reviewing the code.

The report contains all issues found during both the threat modelling and manual code audit exercises. Five of these issues were exploitable by threat actors identified during the threat modelling, and these issues were assigned CVEs which are listed in the table below. In addition, after reviewing one of the vulnerabilities, the Kyverno team identified a similar issue in a 3rd-party dependency, Cosign, which Ada Logics disclosed to Cosign and worked with the project on a fix. This is also included below and is not referenced elsewhere in the report.

| Issue | CVE | CVE severitiy |
|---|---|---|
| Remote user can make Kyverno users consume incorrect image | CVE-2023-47630 | High |
| Denial of service from malicious signature | CVE-2023-42816 | Moderate |
| Denial of service from malicious index manifest 1 | CVE-2023-42815 | Low |
| Denial of service from malicious index manifest 2 | CVE-2023-42814 | Low |
| Denial of service from malicious manifest layer | CVE-2023-42813 | Moderate |
| Possible endless data attack from attacker-controlled registry | CVE-2023-46737 | Low |

Ada Logics disclosed these findings responsibly to Kyverno through Kyverno's public GitHub Security Advisory disclosure channels. The Kyverno security response team responded to the disclosures with fixes in a timely manner before the audit was completed.

During the fuzzing goal, Ada Logics reviewed Kyverno's fuzzing suite and added two fuzzers that target one attack surface identified during the threat modelling: Policy bypasses, i.e. where an internal attacker attempts to submit a request that bypasses a policy deployed by the Kyverno admin.

The SLSA review found that Kyverno complies at the highest level (SLSA Level 3). Kyverno builds its releases on GitHub Actions and includes verifiable provenance with releases, which makes Kyverno hardened against a series of well-known attack vectors in Kyverno's software supply-chain.

## Strategic recommendations

Kyverno maintains high security standards. The project maintains a good suite of automated testing with support of state-of-the-art SAST (static application security testing) and DAST (dynamic application security testing) security tools in the CI.

Kyverno's SLSA compliance demonstrates excessive work on supply-chain security, and during the audit, we learned that the Kyverno team works well with the community on mitigating and patching disclosed security vulnerabilities.

Kyverno maintains a high level of security maturity, and moving forward, Kyverno should maintain this level.

We recommend that Kyverno engages with maintainers and the community to continue the work that Ada Logics has done to set up the Kyverno fuzz suite in Kyvernos fuzzing security audit completed earlier in 2023, as well as the work Ada Logics did during this audit to improve the fuzzing suite.

Kyvernos fuzzers run in a continuous manner, and the project can leverage the experience and field knowledge of maintainers and the community to write fuzzers that call security-sensitive APIs considered specific attack vectors. This should be considered ongoing work, which can be done in regular sprints or as part of ongoing work.

In addition, we recommend that Kyverno encourages contributors to include fuzz tests with code contributions and/or code changes. The CNCF-Fuzzing handbook is a great source of reference for people wishing to get started with fuzzing the CNCF project landscape: https://github.com/cncf/tag-security/blob/main/security-fuzzing-handbook/handbook-fuzzing.pdf.

With regards to security hardening the Kyverno code case, we recommend validating all input from remote data sources, including remote registries. Multiple vulnerabilities found in this audit were from lack of input received after sending a request to a remote registry. If Kyverno adds similar functionality or changes existing code parts, we recommend validating the received data from calls to remote sources.

While Kubernetes tooling makes it easy to build dynamic admission controllers, security and hardening of these critical components is often overlooked. The Kyverno project's work on security and recommendations in this report can serve as a guide for implementors.

# Project Scope

The following Ada Logics auditors carried out the audit and prepared the report.

| Name | Title | Email |
|---|---|---|
| Adam Korczynski | Security Engineer, Ada Logics | Adam@adalogics.com |
| David Korczynski | Security Researcher, Ada Logics | David@adalogics.com |

The following Kyverno team members were part of the audit.

| Name | Title | Email |
|---|---|---|
| Jim Bugwadia | maintainer | jim@nirmata.com |
| Shuting Zhao | maintainer | shuting@nirmata.com |
| Charles-Edouard Brétéché | maintainer | charles.edouard@nirmata.com |
| Chip Zoller | maintainer | chipzoller@gmail.com |

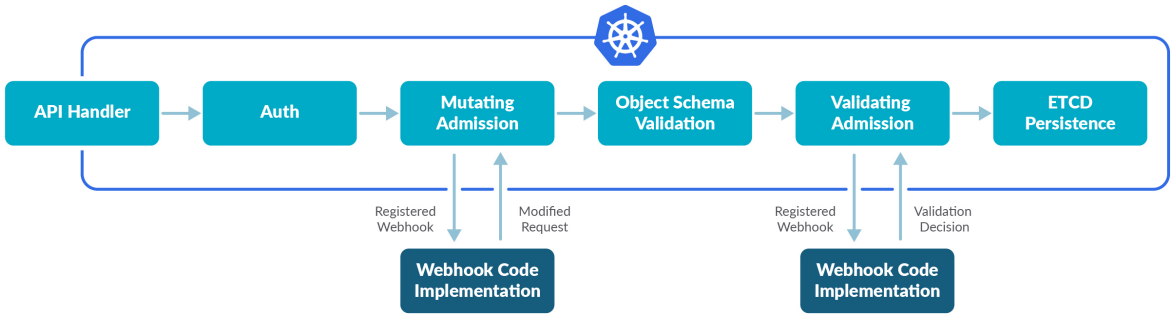
The following OSTIF members were part of the audit.

| Name | Title | Email |
|---|---|---|
| Derek Zimmer | Executive Director, OSTIF | Derek@ostif.org |
| Amir Montazery | Managing Director, OSTIF | Amir@ostif.org |

# Threat model

Kyverno is a Kubernetes admission controller, which implements a policy engine that allows cluster administrators and DevOps teams to granularly define policies that make admission decisions on workloads in the cluster. Kyverno reads the resources from incoming requests and either validates that the resource conforms to the policies that the cluster admin has deployed, or Kyverno changes the resource to conform to the deployed policies. The former is called validation and the latter is called mutation. Mutation and validation are concepts originating from the Kubernetes admission controller design; Admission control in Kubernetes is divided into two stages: Mutation runs in the first part and validation in the second.

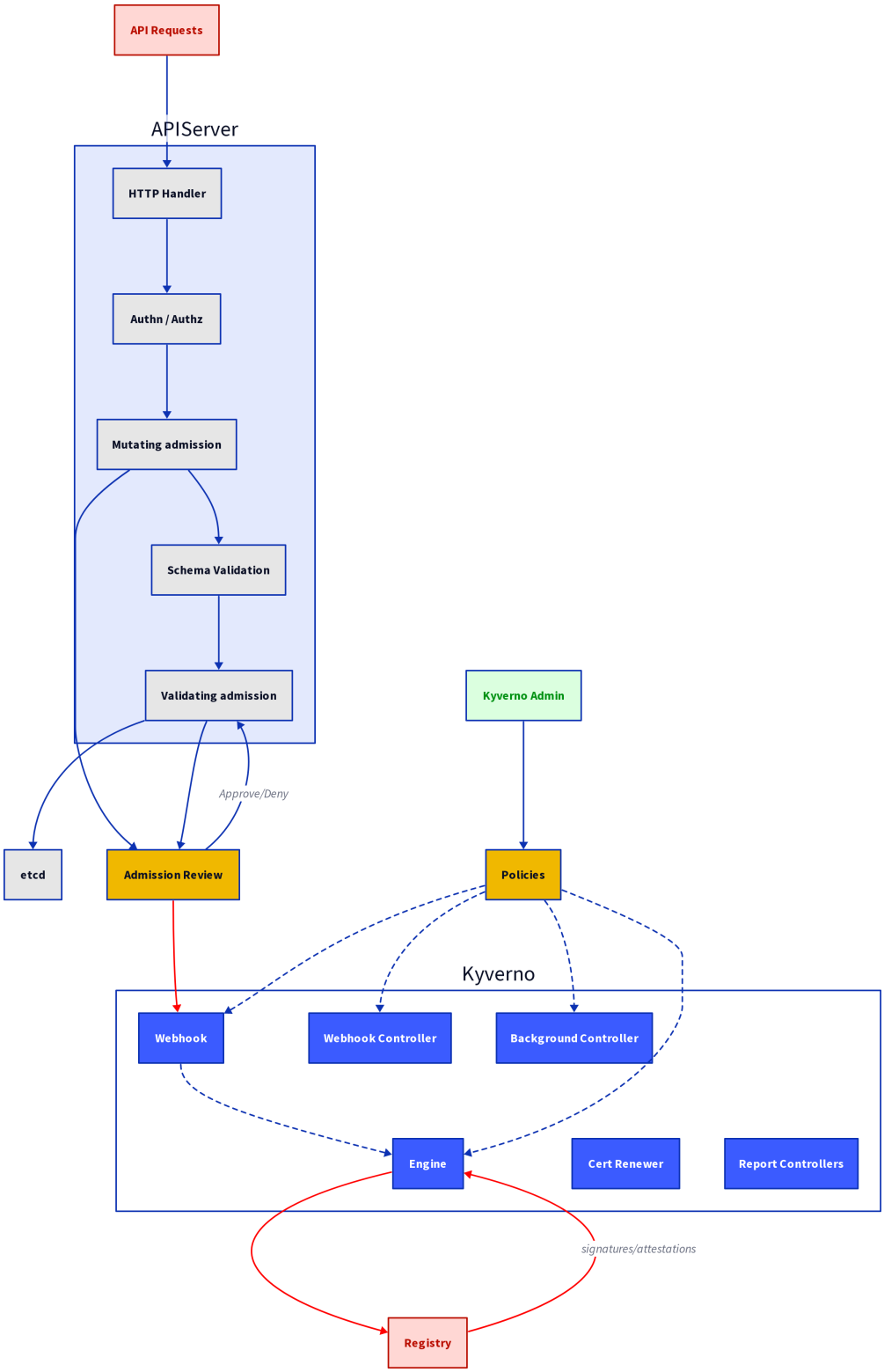
Kubernetes admission controllers are hooks into the Kubernetes request lifecycle. They run after Kubernetes has authenticated and authorized the request. The entire request lifecycle can be split into six separate steps:




The lifecycle starts with the user making a request to the APIServer, which the API Handler receives. Kubernetes then authenticates and authorizes the request, after which the `MutatingAdmissionWebhook` invokes the mutating admission control webhook. If the request fails at the mutating step, the API Server rejects the request, and the user receives an error. If the request passes the mutating admission controller, the request proceeds to the schema validation step, where the API Server validates that the resource conforms to the schema. If it does, then the request proceeds to the admission validation step, which invokes the `ValidatingAdmissionWebhook` . This is the second part of the admission control interface; Here, a validating admission controller validates the incoming request and returns a binary response: either "pass" or "fail". Also, at this stage, if the response is "fail", the request is rejected immediately. If the request passes, then the request proceeds to etcd and has been approved by the cluster.

Kyverno's policy engine allows users to create admission policies with granular requirements for the resources in the incoming requests. At a high level, users can filter and evaluate fields and values of a resource and make admission decisions from them. For example, users can consider the number of containers that a Pod resource type includes, whether a path field points to a specific location, or whether a field is true or false.

As such, Kyverno's interaction with the Kubernetes request lifecycle looks as follows:


API Requests
APIServer
HTTP Handler
Authn / Authz
Mutating admission
Schema Validation
Validating admission
Kyverno Admin
Approve/Deny
etcd
Admission Review
Policies
Kyverno
Webhook
Webhook Controller
Background Controller
Engine
Cert Renewer
Report Controllers
signatures/attestations
Registry

The entry points into Kyverno are via the mutating and validating webhooks. The requests flow through Kyverno's handler for either mutation or validation and into the Kyverno engine. The engine is responsible for processing the request against the deployed policies in the cluster and return a "pass" or "fail" response to the API Server. During the processing of requests against the deployed policies, the engine can make two types of remote calls: 1) To an image registry to fetch data about images, such as signatures attestations for verification and 2) to services of the Kyverno admins choosing. The Kyverno engine makes admission decisions based on the data returned from both types of remote services.

## Kyverno Trust Boundaries

Kyverno has two trust boundaries:

1: Between the cluster user and the request authentication and authorization. At this boundary, trust flows from low to high in the direction from the external user to the API Server. While this is a trust boundary for Kyverno, it relies on Kubernetes properly authenticating and authorizing the request; As such, Kyverno always receives authenticated and authorized requests via the admission controller webhooks.

2: Between the Kyverno engine and the external data sources and image registries. At this boundary, trust flows high to low in the direction from the engine to the remote data source and low to high from the remote data source to the Kyverno engine.

The Kyverno admin is a fully trusted user, and attacks from or against this user do not fall within Kyverno's threat model. These attacks can be policy confusion attacks where an attacker is able to swap policies on the Kyverno admin's file system after the Kyverno admin verifies them but before the Kyverno admin deploys them to the cluster. Kyverno fully trusts that the data coming from the Kyverno admin is correct and intended. Attacks on the Kyverno admin can result in compromise of the cluster; For example, if an attacker is able to steal the admin's credentials, they may be able to deploy policies of their choosing and, as a result, also deploy resources of their choosing. However, Kyverno cannot improve its security design as it follows the Kubernetes standards for deploying policies to Kyverno.

## Kyverno Attack Surface

The trust boundaries are also Kyverno's attack surface. From the API server, the data with which attackers can attack Kyverno are by way of Kubernetes object requests.

To attack Kyverno from the remote registry, an attacker must either control a registry that Kyverno fetches data from or intercept requests with a man-in-the-middle position. In either case, the registry is untrusted from the perspective of Kyverno; Kyverno is not in control of the security posture of the remote registry, and an attacker could obtain the necessary position by compromising the registry and position themselves between Kyverno and the registry. With such a position, the attacker could return malicious responses to Kyvernos requests to the registry to attempt a myriad of attacks, including policy bypass attacks or image confusion attacks, where an attacker would either consume a malicious image or make Kyverno consume malicious images in other users' workloads.

Kyverno's calls to remote data sources, including remote registries, are particularly interesting since Kyverno makes these calls after several steps of authentication and authorization have occurred in the request lifecycle. As such, the attacker does not require any privileges in the cluster to attack Kyverno when Kyverno makes requests to remote data sources. If an attacker can control responses to Kyverno from remote data sources, they can potentially return any arbitrary data to Kyverno. This combination exposes a supply-chain risk to Kyverno at runtime; An attacker can compromise an external service and attempt to escalate their position onto Kyverno, when Kyverno requests the service. Supply-chain security is an emerging area of interest with ongoing research to identify new attack vectors in the wild and proactively before malicious threat actors find these attack vectors. Supply-chain security covers a wide range of attack surface spanning from the software development life cycle (SDLC) to attacks at runtime. In this context, we are considering the runtime-specific vulnerability classes. TheUpdateFrameworks (TUF) threat model (https://theupdateframework.io/security/) identifies a series of such attacks. Kyverno inherits several of these attacks from its 3rd-party dependencies and in its engine. Below, we enumerate the supply-chain-specific attack vectors which also affect Kyverno:

**Rollback attacks:** An attacker tricks the victim system into downloading or install an older version of an artifact than is currently available. The victim system does not recognize that a newer version is available and, therefore installs the old version. In this scenario, the older version contains vulnerabilities, whereas the newer version does not. In a classic rollback attack, the attacker would need to exploit these vulnerabilities to further advance their position. In Kyverno's case, an attacker could release a newer image by tag or digest - and trick Kyverno into fetching and consuming it. As such, this has a variation compared to a classic rollback attack, since the attack could trick Kyverno into consuming an image that contains the attackers reward as well, for example a crypto miner.

**Arbitrary software installation attacks:** Similar to rollback attacks, but instead of installing a different version of a given artifact, the attacker manages to install an entirely different artifact. The attack vector in Kyverno's case is similar to the Rollback attack.

**Endless data attacks:** An attacker responds to a request with an endless data stream, thereby either exhausting memory or triggering a practical infinite loop. This is an attack vector to Kyverno at any point where Kyverno sends a request to a remote data source or registry. In a situation where an attacker has compromised Kyverno's supply chain - i.e. the remote registry or manages to intercept requests as a man-in-the-middle (MITM) type of attack - they can attempt to cause denial-of-service with an endless stream of data.

**Wrong software installation:** Similar to Rollback attacks and Arbitrary Software Installation attacks, but in this case, the attack delivers a trusted file which is different than the file the victim intended to consume.

Attackers can utilize these supply-chain-specific attack vectors to carry out two high-level types of attacks: 1) Policy bypasses and 2) General attempts of compromise. For policy bypasses, an attacker is able to escalate privileges if they are able to carry out an action that the Kyverno admin has prevented by way of a Kyverno policy. For general attempts of compromise, an attacker can escalate privileges by compromising the confidentiality, integrity or availability for cases that should not be possible even if there is not a policy for it. The list of such cases is long, but as an example: Any non-cluster admin should not be able to achieve cluster admin privileges without the cluster admin granting these. Or: no user should be able to trick Kyverno into running crypto miners unless the Kyverno admin specifically allows this. Or: no user should be able to steal credentials of other users in the cluster.

## Kyverno Threat Actors

A threat actor is an individual or group that intentionally attempts to exploit vulnerabilities, deploys malicious code, or compromise or disrupt a Kyverno deployment, often for financial gain, espionage, or sabotage. A threat actor is the personification of a possible attacker of security issues. Each threat actor has a level of trust tied to them, and matching one or several threat actors with Kyvernos threat model helps identify the high-level security risk. We identify the following threat actors for Kyverno. A threat actor can assume multiple profiles from the table below; for example, a fully untrusted user can also be a contributor to a 3rd-party library used by Kyverno.

To score the level of trust, Ada Logics uses the following levels:

- `None`: The Kyverno admin has not assigned any level of privilege to the user. The Kyverno admin does not know the identity of the actor.
- `Low`: The Kyverno admin has assigned some privileges to the actor but has not assigned other privileges.
- `High`: At this level of trust, the actor has some privileges to modify the Kyverno deployment but might not have full privileges to modify everything.
- `Full`: At this level of trust, there is no doubt that the user can do anything with the Kyverno deployment, including delete the whole cluster. An actor at this level of privileges cannot escalate privileges, and features in the code that allow actors at this level of trust to compromise the deployment are not security issues.

| Threat Actor | Description | Level of trust |
|---|---|---|
| Remote users | Users that have not been granted any privileges. | None |
| Cluster Users | Users that can send requests to the API Server that will authenticate and authorize. The users' requests will reach Kyverno | Low |
| Contributors to Kyverno | Users that make code-contributions to Kyverno. | None |
| Maintainers of Kyverno | Users that have maintainer privileges at Kyvernos Github repository | High |
| Contributors to 3rd-party dependencies | Users that make code-contributions to libraries in Kyvernos dependency tree. This includes maintainers of the third-party libraries. | None |
| Container image vendors - private registries | A person or organization developing, maintaining and/or distributing container images at a private registry | High |
| Container image vendors - public registries | A person or organization developing, maintaining and/or distributing container images at a public registry | High |
| Kyverno admin | An admin with privileges to configure the Kyverno deployment and/or deploy policies to the cluster | Full |

# Fuzzing

As part of the audit, Ada Logics made new additionts to Kyvernos fuzzing suite. Prior to this security audit, Kyverno had integrated into OSS-Fuzz meaning that future improvements to fuzzing Kyverno could be added to its continuous fuzzing set up. During this audit, Ada Logics wrote new fuzzers that we added to Kyvernos OSS-Fuzz integration. The fuzzers ran during the audit and will continue to run after the audit has completed. Kyverno maintains its fuzz tests and OSS-Fuzz integration in its own source tree, and Ada Logics added the fuzzers to the package directories that they test to follow that design.

From a high level, Ada Logics made the following improvements:

**Wrote new fuzzer testing for validation bypasses**
We added a fuzzer to test for policy enforcement against a Pod resource. The fuzzer has eleven admission policies that the fuzzer tries to bypass. The fuzzer will select one of the eleven policies to test in each iteration. It will then create a random Pod resource and make its own assertion of whether the Pod should fail validation. Finally, the fuzzer invokes Kyvernos `Validate` and compares the outcome with its own assertion. The outcome should be the same as its own assertion.

The eleven policies that the fuzzer tests for are shown below. The `Policy` column contains the type of policy. All policies can be found in https://github.com/kyverno/kyverno/blob/main/pkg/enging/fuzz_test.go.

| Name | Policy |
|---|---|
| latest-image-tag-policy | Checks all containers in the Pod. If a container has `image` specified, and it is referenced with `latest`, then the container must set `imagePullPolicy` to `Always`. |
| equality-hostpath-policy | Blocks Pods with `hostPath` set to `/var/lib` |
| security-context-policy | Blocks Pods that don't set `runAsNonRoot` to `true` |
| container-name-policy | Blocks Pods with containers that are not called `nginx` |
| pod-existence-policy | Blocks Pods if not at least one container is called `nginx` |
| host-path-cannot-exist-policy | Blocks Pods specify `hostPath` in its Volumes |
| namespace-cannot-be-empty-or-default-policy | Blocks Pods that have not specified a Namespace, or if the Namespace is `default` |
| hostnetwork-and-port-not-allowed-policy | Blocks Pods that set `hostNetwork` to `true`, or if any of its containers specify `hostPort` |
| supplemental-groups-should-be-higher-than-zero-policy | Blocks Pods if `supplementalGroups` is `0` |
| supplemental-groups-should-be-between | Blocks Pods if `supplementalGroups` is `0` or higher than `100001` |
| should-have-more-memory-than-first-container | Blocks the Pod if the first container sets its request memory limit lower than any of the other containers, and if any containers memory limit is not `2048Mi` |

**Wrote new fuzzer testing for PSS bypasses**

As part of the audit, Ada Logics added a fuzzer for testing Kyvernos PSS enforcement. The fuzzer takes the same approach as the fuzzer testing for validation bypasses: The fuzzer has a set of policies and uses the test case to generate pseudo-random objects that attempt to break Kyvernos PSS verification. Kyvernos Pod Security Admission has had cases of bypasses in the past (CVE-2023-33191), and fuzzing can help test for edge cases that could bypass PSS.

#### Wrote new fuzzer for Kyvernos use of go-jmespath

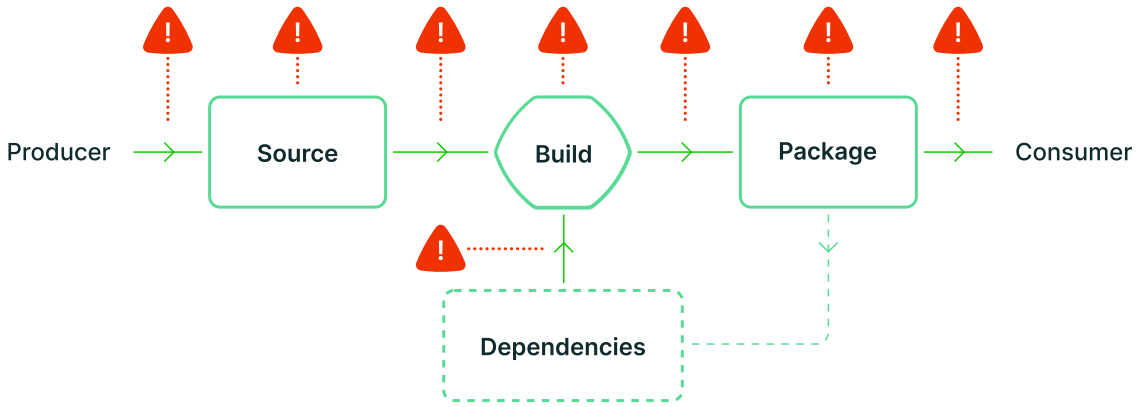
Kyverno maintains its own fork of go-jmespath and uses this across its code. Ada Logics wrote a fuzzer for the `github.com/kyverno/kyverno/pkg/engine/context.(*context).HasChanged()` which calls into Kyvernos go-jmespath library which performs complex processing over a string. Targetting an API that Kyverno invokes deep in the call tree allows for a level of over-approximation; They are likely to trigger issues with data that Kyverno will sanitize further up in the execution tree. At the same time, if code changes one place and removes sanitization that code further down the tree depends on, bugs could become available to untrusted input, and the code may not have sufficient e2e testing that catches this. As such, the fuzzers provide value in finding bugs that may not be cause for concern at this point but may be in the future.

# SLSA review

ADA Logics carried out a SLSA review of Kyvenro. SLSA (https://github.com/slsa.dev) is a framework for assessing the security practices of a given software project with a focus on mitigating supply-chain risk. SLSA emphasises tamper resistance of artifacts as well as ephemerality of the build and release cycle.

SLSA mitigates a series of attack vectors in the software development life cycle (SDLC), all of which have seen real-world examples of successful attacks against open-source and proprietary software.

Below, we see a diagram made by the SLSA illustrating the attack surface of the SDLC.




Each of the red markers demonstrates different areas of possible compromise that could allow attackers to tamper with the artifact that the consumer invokes at the end of the SDLC.

SLSA splits its assessment criteria into 4, increasingly demanding levels. At a high level, the higher the level of compliance, the higher tamper-resistance the project ensures its consumers.

Kyverno builds its releases via the official SLSA-Github-Generator project: https://github.com/slsa-framework/slsa-github-generator. This project implements a reusable workflow which builds the artifacts and creates provenance for the artifacts. Users of the SLSA-Github-Generator reusable workflows can choose to release binaries and provenance via the reusable workflow. Below we see a snippet of how Kyverno uses the reusable workflow to generate provenance for the `kyverno-init` artifact:

https://github.com/kyverno/kyverno/blob/b391694e67dc0e63040d16b6002fc39e3faf40f9/.github/workflows/release.yaml#L151-L164

```
151   generate-kyverno-init-provenance:
152     needs: release-images
153     permissions:
154       id-token: write   # To sign the provenance.
155       packages: write   # To upload assets to release.
156       actions: read     # To read the workflow path.
157     # NOTE: The container generator workflow is not officially released as GA.
158     uses: slsa-framework/slsa-github-
    generator/.github/workflows/generator_container_slsa3.yml@v1.9.0
159     with:
160       image: ghcr.io/${{ github.repository_owner }}/kyvernopre
161       digest: "${{ needs.release-images.outputs.kyverno-init-digest }}"
162       registry-username: ${{ github.actor }}
163     secrets:
164       registry-password: ${{ secrets.GITHUB_TOKEN }}
```

Kyverno uses v1.9.0, which is the latest release.

This usage represents the optimal way for building and release, according to the SLSA framework, and Kyverno mitigates the risks identified by SLSA to the highest degree. As such, Kyverno complies with SLSA level 3, which is the highest level.

# Issues found

Ada Logics found 10 issues during the audit. The list includes all issues found by way of manual auditing and fuzzing. Ada Logics uses a scoring system that considers impact and ease of exploitation. This is different from the CVSS scoring system, and there may be discrepancies between the severity assigned by Ada Logics and the severity resulting from a CVSS calculation.

| # | Title | Status | Severity |
|---|---|---|---|
| 1 | Denial of service from malicious index manifest 1 | Fixed | Low |
| 2 | Denial of service from malicious index manifest 2 | Fixed | Low |
| 3 | Denial of service from malicious manifest layer | Fixed | Moderate |
| 4 | Denial of service from malicious signature | Fixed | Moderate |
| 5 | Index out of range panic in third-party dependency | Fixed | Low |
| 6 | Missing closing of HTTP response body | Fixed | Low |
| 7 | Missing size check when making call to remote services. | Fixed | Low |
| 8 | Possible endless data attack from attacker-controlled registry | Fixed | Low |
| 9 | Remote user can make Kyverno users consume incorrect image | Fixed | High |
| 10 | Type Confusion in Jmespath Execution | Fixed | Low |

# Denial of service from malicious index manifest 1

| | |
|---|---|
| **Severity:** | Low |
| **Status:** | fixed |
| **Id:** | ADA-KYV-2023BHFA |
| **Component:** | Notary Verifier |

Kyverno's Notary client implementation is susceptible to a Denial-of-Service vulnerability. The vulnerable method is `ListSignatures`, which fetches referrers on an image descriptor from a remote registry, gets the index manifest of the referrers, adds the descriptors of `ArtifactTypeNotation` and finally invokes `fn()` on the descriptors. An attacker who controls the remote registry can return malicious referrers containing a very high amount of manifests that would cause the loop to continue for an amount of time controlled by the attacker. Below, on line 43, the attacker can control the malicious referrers, and the loop on lines 54-57 continues for an amount of time controlled by an attacker.

https://github.com/kyverno/kyverno/blob/9361100f1761eec0e3cad6d389dd06d802b382ef/pkg/notary/repository.go#L42-L61

```
42 func (c *repositoryClient) ListSignatures(ctx context.Context, desc
   ocispec.Descriptor, fn func(signatureManifests []ocispec.Descriptor) error) error {
43         referrers, err :=
   remote.Referrers(c.ref.Context().Digest(desc.Digest.String()), c.remoteOpts...)
44         if err != nil {
45                 return err
46         }
47
48         referrersDescs, err := referrers.IndexManifest()
49         if err != nil {
50                 return err
51         }
52
53         descList := []ocispec.Descriptor{}
54         for _, d := range referrersDescs.Manifests {
55                 if d.ArtifactType == notationregistry.ArtifactTypeNotation {
56                         descList = append(descList, v1ToOciSpecDescriptor(d))
57                 }
58         }
59
60         return fn(descList)
61 }
```

An attacker can exploit this in different ways; in every attack scenario, they would need to control the response from the registry to Kyverno, when Kyverno sends a request to a remote registry. From the perspective of Kyverno's threat model, the registry is considered untrusted since it is not always controlled by the Kyverno user. There are several ways the attacker could control the registry. One is to compromise the server on which the registry runs. At this level, the attacker would compromise the machine on which the registry is running, which could be enabled by misconfiguration or lack of patching security vulnerabilities. The registry could also be deployed without proper and best-practices configuration. In this case, the attacker would look for misconfiguration in the way the registry was deployment, i.e. weak authorization, hidden admin pages, HTTP traffic where it should be HTTPS and other mistakes. An attacker could also seek to compromise the registry at the code level and add vulnerabilities to the code. If the registry is open-source, the attacker can make malicious pull requests containing code that obfuscates vulnerabilities. If the code for the registry is closed-source, the attacker can attempt to steal developer credentials and impersonate an employee to make pull requests to the registry. Highlighting these different levels of compromise illustrates the attack vector of the remote registry. Kyverno has no control over the configuration of the server, the registry or the security practices surrounding developer credentials, and Kyverno must guard itself against such insecurities from the registry.

In the event that an attacker is able to control the response to Kyverno from the registry, they will return referrers with a high number of manifests, which causes Kyverno to go into a loop without completing validation. This could prevent Kyverno from validating requests made by any users on the cluster, and users on the cluster would not be able to deploy new resources to the cluster. The outcome of this attack would be to prevent other users from using the cluster for legitimate purposes.

A variation on this attack is if the attacker has permissions to make requests to the Kubernetes APIServer; the attacker can release an image containing a high number of referrer manifests and then make an admission request with a reference to that image. As a result, the attacker could trigger the vulnerability and deny Kyverno from validating requests from other users.

This vulnerability requires interaction from Kyverno for an attacker to exploit it. It is not possible to send a request to Kyverno with the malicious payload; the vulnerability can only be exploited with a malicious response once Kyverno makes a request to the registry.

# Denial of service from malicious index manifest 2

| | |
|---|---|
| **Severity:** | Low |
| **Status:** | fixed |
| **Id:** | ADA-KYV-2023IBQE |
| **Component:** | Notary Verifier |

Kyverno's Notary client implementation is susceptible to a Denial-of-Service vulnerability. The vulnerable method is `FetchAttestations`, which fetches referrers from a remote registry, gets the index manifest of the referrers and loops through the manifests. An attacker who controls the remote registry can return malicious referrers containing a very high amount of manifests that would cause the loop to continue for an amount of time controlled by the attacker. Below, on line 149, the attacker would return the malicious referrers, and the loop on line 169-202 would continue for an amount of time controlled by the attacker, depending on the amount of manifests.

https://github.com/kyverno/kyverno/blob/9361100f1761eec0e3cad6d389dd06d802b382ef/pkg/notary/notary.go#L130-L205

```
130 func (v *notaryVerifier) FetchAttestations(ctx context.Context, opts images.Options)
    (*images.Response, error) {
131         v.log.V(2).Info("fetching attestations", "reference", opts.ImageRef, "opts",
    opts)
132
133         ref, err := name.ParseReference(opts.ImageRef)
134         if err != nil {
135                 return nil, errors.Wrapf(err, "failed to parse image reference: %s",
    opts.ImageRef)
136         }
137         authenticator, err := getAuthenticator(ctx, opts.ImageRef, opts.Client)
138         if err != nil {
139                 return nil, errors.Wrapf(err, "failed to parse authenticator: %s",
    opts.ImageRef)
140         }
141
142         remoteOpts, err := getRemoteOpts(*authenticator)
143         if err != nil {
144                 return nil, err
145         }
146
147         v.log.V(4).Info("client setup done", "repo", ref)
148
149         repoDesc, err := gcrremote.Head(ref, remoteOpts...)
150         if err != nil {
151                 return nil, err
152         }
153         v.log.V(4).Info("fetched repository", "repoDesc", repoDesc)
154
155         referrers, err :=
    gcrremote.Referrers(ref.Context().Digest(repoDesc.Digest.String()), remoteOpts...)
156         if err != nil {
157                 return nil, err
158         }
159
160         referrersDescs, err := referrers.IndexManifest()
161         if err != nil {
162                 return nil, err
163         }
164
165         v.log.V(4).Info("fetched referrers", "referrers", referrersDescs)
166
167         var statements []map[string]interface{}
168
169         for _, referrer := range referrersDescs.Manifests {
170                 match, _, err := matchArtifactType(referrer, opts.Type)
171                 if err != nil {
172                         return nil, err
173                 }
174
175                 if !match {
176                         v.log.V(6).Info("type doesn't match, continue", "expected",
    opts.Type, "received", referrer.ArtifactType)
```

```
177                         continue
178                 }
179
180                 targetDesc, err := verifyAttestators(ctx, v, ref, opts, referrer)
181                 if err != nil {
182                         msg := err.Error()
183                         v.log.V(4).Info(msg, "failed to verify referrer %s",
    targetDesc.Digest.String())
184                         return nil, err
185                 }
186
187                 v.log.V(4).Info("extracting statements", "desc", referrer, "repo",
    ref)
188                 statements, err = extractStatements(ctx, ref, referrer, remoteOpts)
189                 if err != nil {
190                         msg := err.Error()
191                         v.log.V(4).Info("failed to extract statements %s", "err",
    msg)
192                         return nil, err
193                 }
194
195                 v.log.V(4).Info("verified attestators", "digest",
    targetDesc.Digest.String())
196
197                 if len(statements) == 0 {
198                         return nil, fmt.Errorf("failed to fetch attestations")
199                 }
200                 v.log.V(6).Info("sending response")
201                 return &images.Response{Digest: repoDesc.Digest.String(),
    Statements: statements}, nil
202         }
203
204         return nil, fmt.Errorf("failed to fetch attestations %s", err)
205 }
```

An attacker can exploit this in different ways; in every attack scenario, they would need to control the response from the registry to Kyverno when Kyverno makes a request to a remote registry. From the perspective of Kyverno's threat model, the registry is considered untrusted since it is not always controlled by the Kyverno user. There are several ways the attacker could control the registry. One is to compromise the server on which the registry runs. At this level, the attacker would compromise the machine on which the registry is running, which could be enabled by misconfiguration or lack of patching security vulnerabilities. The registry could also be deployed without proper and best-practices configuration. In this case, the attacker would look for misconfiguration in the deployment of the registry, i.e. weak authorization, hidden admin pages, HTTP traffic where it should be HTTPS and other mistakes. An attacker could also seek to compromise the registry at the code level and add vulnerabilities to the code. If the registry is open-source, the attacker can make malicious pull requests containing code that obfuscates vulnerabilities. If the code for the registry is closed-source, the attacker can attempt to steal developer credentials and impersonate an employee to make pull requests to the registry. Highlighting these different levels of compromise illustrates the attack vector of the remote registry. Kyverno has no control over the configuration of the server, the registry or the security practices surrounding developer credentials, and Kyverno must guard itself against such insecurities from the registry.

In the event that an attacker is able to control the response to Kyverno from the registry, they will return referrers with a high number of manifests, which cause Kyverno to go into a loop without completing validation. This could prevent Kyverno from validating requests made by any users on the cluster, and users on the cluster would not be able to deploy new resources to the cluster. The outcome of this attack would be to prevent other users from using the cluster for legitimate purposes.

A variation on this attack is if the attacker has permissions to make requests to the Kubernetes APIServer; the attacker can release an image containing a high number of referrer manifests and then make an admission request with a reference to that image. As a result, the attacker could trigger the vulnerability and deny Kyverno from validating requests from other users.

This vulnerability requires interaction from Kyverno for an attacker to exploit it. It is not possible to send a request to Kyverno with the malicious payload; the vulnerability can only be exploited with a malicious response once Kyverno makes a request to the registry.

### Advisory details

GHSA: GHSA-9g37-h7p2-2c6r CVE: CVE-2023-42814

# Denial of service from malicious manifest layer

| | |
|---|---|
| **Severity:** | Moderate |
| **Status:** | fixed |
| **Id:** | ADA-KYV-2023NADD |
| **Component:** | Notary Verifier |

Kyvernos Notary client implementation is susceptible to a Denial-of-Service vulnerability. The issue exists in `extractStatement` API, which fetches a signature layer from a remote registry and reads it entirely into memory. If an attacker has control over the remote registry and can return a descriptor that has a manifest that has a layer that has a very large predicate, they can crash Kyverno by exhausting memory of the machine when Kyverno reads the memory, resulting in denial of service for other users. The attacker will return a malicious descriptor on line 290 below, and on line 326, Kyverno reads the predicate entirely into memory:

https://github.com/kyverno/kyverno/blob/9361100f1761eec0e3cad6d389dd06d802b382ef/pkg/notary/notary.go#L283-L341

```
283 func extractStatement(ctx context.Context, repoRef name.Reference, desc
    v1.Descriptor, remoteOpts []gcrremote.Option) (map[string]interface{}, error) {
284         refStr := repoRef.Context().RegistryStr() + "/" +
    repoRef.Context().RepositoryStr() + "@" + desc.Digest.String()
285         ref, err := name.ParseReference(refStr)
286         if err != nil {
287                 return nil, errors.Wrapf(err, "failed to parse image reference: %s",
    refStr)
288         }
289
290         remoteDesc, err := gcrremote.Get(ref, remoteOpts...)
291         if err != nil {
292                 return nil, fmt.Errorf("error in fetching manifest: %w", err)
293         }
294         manifestBytes, err := remoteDesc.RawManifest()
295         if err != nil {
296                 return nil, fmt.Errorf("error in fetching statement: %w", err)
297         }
298         var manifest ocispec.Manifest
299         if err := json.Unmarshal(manifestBytes, &manifest); err != nil {
300                 return nil, err
301         }
302
303         if len(manifest.Layers) == 0 {
304                 return nil, fmt.Errorf("no predicate found: %+v", manifest)
305         }
306         if len(manifest.Layers) > 1 {
307                 return nil, fmt.Errorf("multiple layers in predicate not supported:
    %+v", manifest)
308         }
309         predicateDesc := manifest.Layers[0]
310
311         layer, err :=
    gcrremote.Layer(ref.Context().Digest(predicateDesc.Digest.String()), remoteOpts...)
312         if err != nil {
313                 return nil, err
314         }
315         ioPredicate, err := layer.Uncompressed()
316         if err != nil {
317                 return nil, err
318         }
319         predicateBytes := new(bytes.Buffer)
320         _, err = predicateBytes.ReadFrom(ioPredicate)
321         if err != nil {
322                 return nil, err
323         }
324
325         predicate := make(map[string]interface{})
326         if err := json.Unmarshal(predicateBytes.Bytes(), &predicate); err != nil {
327                 return nil, err
328         }
329         data := make(map[string]interface{})
330         if err := json.Unmarshal(manifestBytes, &data); err != nil {
331                 return nil, err
```

```
332         }
333
334         if data["type"] == nil {
335                 data["type"] = desc.ArtifactType
336         }
337         if data["predicate"] == nil {
338                 data["predicate"] = predicate
339         }
340         return data, nil
341 }
```

The vulnerability exists in Kyverno's Notary verifier, which has a method - `FetchAttestations` - that downloads a predicate to verify an image against. Kyverno uses the Notary verifier as part of the Kyverno imageVerifier. The Kyverno imageVerifier first verifies the attestors and then the attestions of an image. When the imageVerifier proceeds to verify the attestations, it downloads an attestation for each entry of the attestors. It is the job of the Kyverno Notary verifier to download the attestations; specifically, the `FetchAttestations` method of the Notary verifier handles the downloading. `FetchAttestations` first downloads a list of manifest digests called "Referrers". For each manifest digest, the Kyverno Notary verifier verifies the digest against the attestators and then downloads the manifest from a remote registry. The downloaded manifest should have a single layer, which is the predicate and attestation that Kyverno will verify against the image in the requested workload in the cluster. The predicate layer is compressed, and when Kyverno decompresses it, it reads in entirely into memory without checking for - or enforcing an upper limit to the size of the layer. As such, if the layer is sufficiently large, it could drain memory of the host machine and cause a resource-exhaustion denial-of-service of Kyverno.

To launch this attack, an attacker needs to first be in a position that allows them to control the response from the registry.

**Advisory Details**
GHSA: GHSA-wc3x-5rfv-hh5v CVE: CVE-2023-42813

# Denial of service from malicious signature

| | |
|---|---|
| **Severity:** | Moderate |
| **Status:** | fixed |
| **Id:** | ADA-KYV-2023MKDS |
| **Component:** | Notary Verifier |

The Kyverno Notary verifier is susceptible to a Denial-of-Service (DoS) attack from a malicious signature.

The `FetchSignatureBlob` method fetches a signature layer from a remote registry and reads it entirely into memory. If an attacker has control over the remote registry and can return a descriptor that has a manifest that has a very large signatureBlobLayer, they can crash Kyverno by exhausting memory of the machine when Kyverno reads the memory, resulting in denial of service for other users. The attacker will return a malicious descriptor on line 69 below, and on line 104, Kyverno reads the layer entirely into memory:

https://github.com/kyverno/kyverno/blob/9361100f1761eec0e3cad6d389dd06d802b382ef/pkg/notary/repository.go#L63-L105

```
63 func (c *repositoryClient) FetchSignatureBlob(ctx context.Context, desc
   ocispec.Descriptor) ([]byte, ocispec.Descriptor, error) {
64         manifestRef, err := name.ParseReference(c.getReferenceFromDescriptor(desc))
65         if err != nil {
66                 return nil, ocispec.Descriptor{}, err
67         }
68
69         remoteDesc, err := remote.Get(manifestRef, c.remoteOpts...)
70         if err != nil {
71                 return nil, ocispec.Descriptor{}, err
72         }
73         manifestBytes, err := remoteDesc.RawManifest()
74         if err != nil {
75                 return nil, ocispec.Descriptor{}, err
76         }
77
78         var manifest ocispec.Manifest
79         if err := json.Unmarshal(manifestBytes, &manifest); err != nil {
80                 return nil, ocispec.Descriptor{}, err
81         }
82         manifestDesc := manifest.Layers[0]
83
84         signatureBlobRef, err :=
   name.ParseReference(c.getReferenceFromDescriptor(manifestDesc))
85         if err != nil {
86                 return nil, ocispec.Descriptor{}, err
87         }
88
89         signatureBlobLayer, err :=
   remote.Layer(signatureBlobRef.Context().Digest(signatureBlobRef.Identifier()),
   c.remoteOpts...)
90         if err != nil {
91                 return nil, ocispec.Descriptor{}, err
92         }
93
94         io, err := signatureBlobLayer.Uncompressed()
95         if err != nil {
96                 return nil, ocispec.Descriptor{}, err
97         }
98         SigBlobBuf := new(bytes.Buffer)
99
100         _, err = SigBlobBuf.ReadFrom(io)
101         if err != nil {
102                 return nil, ocispec.Descriptor{}, err
103         }
104         return SigBlobBuf.Bytes(), manifestDesc, nil
105 }
```

### Advisory

GHSA: GHSA-4mp4-46gq-hv3r CVE: CVE-2023-42816

# Index out of range panic in third-party dependency

| | |
|---|---|
| **Severity:** | Low |
| **Status:** | fixed |
| **Id:** | ADA-KYV-2023ASDV |
| **Component:** | go-jmespath |

One of the fuzzers written during this security audit detected an index-out-of-range `panic` in a third-party dependency to Kyverno: go-jmespath.

The `panic` was found by fuzzing Kyverno and not the third-party dependency directly, and as such, it was triggerable through Kyverno. That being said, the fuzzer over-approximates its input test cases, and the issue was unlikely to be triggerable in a production deployment of Kyverno.

The crash was found by OSS-Fuzz, after Ada Logics added the fuzzer to Kyverno's continuous fuzzing integration. Below, we include the stack trace of the crash produced by OSS-Fuzz:

```
        panic: runtime error: index out of range [2] with length 2 [recovered]
  panic: runtime error: index out of range [2] with length 2
goroutine 17 [running, locked to thread]:
main.catchPanics()
  ./main.1065168555.go:49 +0x27c
panic({0x4311000?, 0x10c000b03440?})
  runtime/panic.go:914 +0x21f
github.com/kyverno/go-jmespath.(*Lexer).consumeUnquotedIdentifier(0x10c000bf9020)
  github.com/kyverno/go-jmespath@v0.4.1-0.20230705123211-d067dc3d6613/lexer.go:389 +0x237
github.com/kyverno/go-jmespath.(*Lexer).tokenize(0x10c000e0d020, {0x10c000bb058c?,
0x57c411?})
  github.com/kyverno/go-jmespath@v0.4.1-0.20230705123211-d067dc3d6613/lexer.go:153 +0x125
github.com/kyverno/go-jmespath.(*Parser).Parse(0x10c000e0d0a0, {0x10c000bb058c?, 0x0?})
  github.com/kyverno/go-jmespath@v0.4.1-0.20230705123211-d067dc3d6613/parser.go:129 +0xc6
github.com/kyverno/go-jmespath.Compile({0x10c000bb058c?, 0x0?})
  github.com/kyverno/go-jmespath@v0.4.1-0.20230705123211-d067dc3d6613/api.go:16 +0x65
github.com/kyverno/kyverno/pkg/engine/jmespath.newJMESPath({0x44fcb00, 0x59884e0},
{0x10c000bb058c?, 0x59884e0?})
  github.com/kyverno/kyverno/pkg/engine/jmespath/new.go:9 +0x5a
github.com/kyverno/kyverno/pkg/engine/jmespath.implementation.Query(...)
  github.com/kyverno/kyverno/pkg/engine/jmespath/interface.go:25
```

# Missing closing of HTTP response body

| | |
|---|---|
| **Severity:** | Low |
| **Status:** | fixed |
| **Id:** | ADA-KYV-2023QKNE |
| **Component:** | apiCall Package |

Kyvernos `apiCall` package allows users to make calls to remote services to fetch data and use it to make admission decisions. When `apiCall` makes a request to a remote service, it checks the status code and proceeds into this branch, if the status is not `200`:

https://github.com/kyverno/kyverno/blob/4046315dac616d2bff6dab54a123d8aec79e558a/pkg/engine/apicall/apiCall.go#L131-L138

```
131         if resp.StatusCode < 200 || resp.StatusCode >= 300 {
132                 b, err := io.ReadAll(resp.Body)
133                 if err == nil {
134                         return nil, fmt.Errorf("HTTP %s: %s", resp.Status,
    string(b))
135                 }
136
137                 return nil, fmt.Errorf("HTTP %s", resp.Status)
138         }
```

In this branch, Kyverno is missing a call to close the response body. According to the Golang `net/http` documentation[https://pkg.go.dev/net/http](https://pkg.go.dev/net/http), users should close the body if the request succeeds:

> The caller must close the response body when finished with it:

```
resp, err := http.Get("http://example.com/")
if err != nil {
    // handle error
}
defer resp.Body.Close()
body, err := io.ReadAll(resp.Body)
// ...
```

# Missing size check when making call to remote services.

| | |
|---|---|
| **Severity:** | Low |
| **Status:** | fixed |
| **Id:** | ADA-KYV-2023AKLS |
| **Component:** | apiCall Package |

Kyvernos `apiCall` package allows users to make calls to remote services to fetch data and use it to make admission decisions. When `apiCall` makes a request to a remote service, it reads the response body entirely into memory. If an attacker is able to return a large response to Kyverno or the user misconfigures their remote service to return a response with a large body, Kyverno could exhaust memory of the machine and crash. The result of this would be a denial of service of Kyverno for other users of the cluster.

On the below lines, Kyverno reads the response entirely into memory on lines 132 and 141:

https://github.com/kyverno/kyverno/blob/4046315dac616d2bff6dab54a123d8aec79e558a/pkg/engine/apicall/apiCall.go#L111-L148

```
111 func (a *apiCall) executeServiceCall(ctx context.Context, apiCall
    *kyvernov1.APICall) ([]byte, error) {
112         if apiCall.Service == nil {
113                 return nil, fmt.Errorf("missing service for APICall %s",
    a.entry.Name)
114         }
115
116         client, err := a.buildHTTPClient(apiCall.Service)
117         if err != nil {
118                 return nil, err
119         }
120
121         req, err := a.buildHTTPRequest(ctx, apiCall)
122         if err != nil {
123                 return nil, fmt.Errorf("failed to build HTTP request for APICall %s:
    %w", a.entry.Name, err)
124         }
125
126         resp, err := client.Do(req)
127         if err != nil {
128                 return nil, fmt.Errorf("failed to execute HTTP request for APICall
    %s: %w", a.entry.Name, err)
129         }
130
131         if resp.StatusCode < 200 || resp.StatusCode >= 300 {
132                 b, err := io.ReadAll(resp.Body)
133                 if err == nil {
134                         return nil, fmt.Errorf("HTTP %s: %s", resp.Status,
    string(b))
135                 }
136
137                 return nil, fmt.Errorf("HTTP %s", resp.Status)
138         }
139
140         defer resp.Body.Close()
141         body, err := io.ReadAll(resp.Body)
142         if err != nil {
143                 return nil, fmt.Errorf("failed to read data from APICall %s: %w",
    a.entry.Name, err)
144         }
145
146         a.logger.Info("executed service APICall", "name", a.entry.Name, "len",
    len(body))
147         return body, nil
148 }
```

The `apiCall` package allows Kyverno users to make HTTP requests to remote services and use the data from the response to make decisions on admission requests. In case an attacker establishes control over the remote service and is able to return a response of their choice, they can crash Kyverno by draining resources of the host machine. To do this, they would need to intercept the request from Kyverno and return a malicious response. A successful attack would prevent Kyverno from validating workloads from other users. The attacker does not require privileges in the cluster, i.e. they do not need to be able to make requests to the cluster to crash Kyverno. As such, the attacker is remote and untrusted from the perspective of the cluster; however, the attacker must escalate privileges outside of the cluster, or they must trick the Kyverno admin to make requests to a malicious service to carry out this attack successfully.

#### Mitigation

Kyverno should check the size of the response body before reading it into memory.

# Possible endless data attack from attacker-controlled registry

| | |
|---|---|
| **Severity:** | Low |
| **Status:** | fixed |
| **Id:** | ADA-KYV-2023KANF |
| **Component:** | Cosign Verifier |

This is a vulnerability in a 3rd-party dependency of Kyverno, Cosign, which was identified in collaboration between the Ada Logics auditors and the Kyverno team. Kyverno has two options for verifying images: Notary and Cosign. We (the Kyverno team and Ada Logics) identified a vulnerability in Cosign that could allow an attacker to carry out an endless data attack and thereby launch a denial of service against the user of Cosign. The attack would result in Kyverno running for a time controlled by an attacker, which would result in Kyverno spending excessive time on a single request and preventing other users and requests from completing their intended tasks. "Endless data attacks" in a supply-chain specific vulnerability class where an attacker sends an endless stream of data to the victim, which causes the victim's systems to be stuck on a single task.

An attacker would need to either control the registry that Cosign communicates with or intercept traffic between Cosign and the registry and return the malicious response.

The issue existed in Cosigns `FetchAttestations`, which pulls the attestations of a signed entity from a remote registry and reads each of them. Cosign did not enforce an upper limit to the number of attestations it would process, and an attacker could return a high number of attestations to launch the endless-data attack. The attestations did not need to be correct or unique to act as a malicious payload.

The vulnerable API looks as such:

https://github.com/sigstore/cosign/blob/0044432284428 50fb28f248fd59765afad99b6df/pkg/cosign/fetch.go#L135-L196

```
135 func FetchAttestations(se oci.SignedEntity, predicateType string)
    ([]AttestationPayload, error) {
136         atts, err := se.Attestations()
137         if err != nil {
138                 return nil, fmt.Errorf("remote image: %w", err)
139         }
140         l, err := atts.Get()
141         if err != nil {
142                 return nil, fmt.Errorf("fetching attestations: %w", err)
143         }
144         if len(l) == 0 {
145                 return nil, errors.New("found no attestations")
146         }
147
148         attestations := make([]AttestationPayload, 0, len(l))
149         var attMu sync.Mutex
150
151         var g errgroup.Group
152         g.SetLimit(runtime.NumCPU())
153
154         for _, att := range l {
155                 att := att
156                 g.Go(func() error {
157                         rawPayload, err := att.Payload()
158                         if err != nil {
159                                 return fmt.Errorf("fetching payload: %w", err)
160                         }
161                         var payload AttestationPayload
162                         if err := json.Unmarshal(rawPayload, &payload); err != nil {
163                                 return fmt.Errorf("unmarshaling payload: %w", err)
164                         }
```

```
165
166                         if predicateType != "" {
167                                 var decodedPayload []byte
168                                 decodedPayload, err =
    base64.StdEncoding.DecodeString(payload.PayLoad)
169                                 if err != nil {
170                                         return fmt.Errorf("decoding payload: %w",
    err)
171                                 }
172                                 var statement in_toto.Statement
173                                 if err := json.Unmarshal(decodedPayload,
    &statement); err != nil {
174                                         return fmt.Errorf("unmarshaling statement:
    %w", err)
175                                 }
176                                 if statement.PredicateType != predicateType {
177                                         return nil
178                                 }
179                         }
180
181                         attMu.Lock()
182                         defer attMu.Unlock()
183                         attestations = append(attestations, payload)
184                         return nil
185                 })
186         }
187         if err := g.Wait(); err != nil {
188                 return nil, err
189         }
190
191         if len(attestations) == 0 && predicateType != "" {
192                 return nil, fmt.Errorf("no attestations with predicate type '%s'
    found", predicateType)
193         }
194
195         return attestations, nil
196 }
```

### Advisory Details

GHSA: GHSA-vfp6-jrw2-99g9 CVE: CVE-2023-46737

# Remote user can make Kyverno users consume incorrect image

| | |
|---|---|
| **Severity:** | High |
| **Status:** | fixed |
| **Id:** | ADA-KYV-2023LNAF |
| **Component:** | Engine Registry Client |

Kyverno is susceptible to a supply-chain vulnerability, which could allow an attacker to carry out a Wrong Software Installation attack.

The root cause of this issue lies in how Kyverno generates the `imageData` from images referenced by digests. The `imageData` map data structure allows Kyverno users to make granular admission decisions against image manifests and stores data about an image such as reference, registry and digest, and other data which images are usually referenced with. To obtain the image data, Kyverno will resolve the image from the reference provided by the Kyverno admin user who will reference an image by tag or digest.

When the image loader fetches the descriptor on this line:

https://github.com/kyverno/kyverno/blob/9361100f1761eec0e3cad6d389dd06d802b382ef/pkg/engine/context/loaders/imagedata.go#L111

```
111   desc, err := client.ForRef(context.Background(), ref)
```

… which in turn gets the descriptor from `FetchImageDescriptor` from this line:

https://github.com/kyverno/kyverno/blob/9361100f1761eec0e3cad6d389dd06d802b382ef/pkg/engine/adapters/rclient.go#L24

```
24   desc, err := a.Client.FetchImageDescriptor(ctx, ref)
```

… Kyverno does not perform validation of the fetched descriptor on these lines:

https://github.com/kyverno/kyverno/blob/9361100f1761eec0e3cad6d389dd06d802b382ef/pkg/registryclient/client.go#L180-L190

```
180 func (c *client) FetchImageDescriptor(ctx context.Context, imageRef string)
    (*gcrremote.Descriptor, error) {
181   parsedRef, err := name.ParseReference(imageRef)
182   if err != nil {
183     return nil, fmt.Errorf("failed to parse image reference: %s, error: %v",
    imageRef, err)
184   }
185   desc, err := gcrremote.Get(parsedRef, gcrremote.WithAuthFromKeychain(c.keychain),
    gcrremote.WithContext(ctx))
186   if err != nil {
187     return nil, fmt.Errorf("failed to fetch image reference: %s, error: %v",
    imageRef, err)
188   }
189   return desc, nil
190 }
```

As such, Kyverno trusts the registry to return the requested image descriptor; however, the registry can be compromised and can be used to deliver a different descriptor than the one requested.

The problematic call is on this line:

https://github.com/kyverno/kyverno/blob/9361100f1761eec0e3cad6d389dd06d802b382ef/pkg/registryclient/client.go#L185

```
185   desc, err := gcrremote.Get(parsedRef, gcrremote.WithAuthFromKeychain(c.keychain),
    gcrremote.WithContext(ctx))
```

On this line, Kyverno makes a call to a remote registry. If this registry is compromised, or the attacker manages to intercept communication and control the response to Kyverno, they can return a descriptor that the Kyverno user did not intend to consume.

The attacker has multiple ways to exploit this vulnerability. Consider this policy:

```
1  apiVersion: kyverno.io/v1
2  kind: ClusterPolicy
3  metadata:
4    name: resolve-image-to-digest
5    annotations:
6      policies.kyverno.io/title: Resolve Image to Digest
7      policies.kyverno.io/category: Other
8      policies.kyverno.io/severity: medium
9      kyverno.io/kyverno-version: 1.6.0
10      policies.kyverno.io/minversion: 1.6.0
11      kyverno.io/kubernetes-version: "1.23"
12      policies.kyverno.io/subject: Pod
13      policies.kyverno.io/description: >-
14        Image tags are mutable and the change of an image can result in the same tag.
15        This policy resolves the image digest of each image in a container and replaces
16        the image with the fully resolved reference which includes the digest rather
    than tag.
17  spec:
18    background: false
19    rules:
20    - name: resolve-to-digest
21      match:
22        any:
23        - resources:
24            kinds:
25            - Pod
26      preconditions:
27        all:
28        - key: "{{request.operation || 'BACKGROUND'}}"
29          operator: NotEquals
30          value: DELETE
31      mutate:
32        foreach:
33        - list: "request.object.spec.containers"
34          context:
35            - name: resolvedRef
36              imageRegistry:
37                reference: "{{ element.image }}"
38                jmesPath: "resolvedImage"
39          patchStrategicMerge:
40            spec:
41              containers:
42              - name: "{{ element.name }}"
43                image: "{{ resolvedRef }}"
```

This policy mutates admission requests to have the reference that the registry returned. As such, a malicious registry can control the accepted values of admission requests.

In this case, the user is able to control the reference to the `resolvedImage` by returning a malicious digest from the call to the remote registry.

The workflow for exploiting this issue is as follows:

1. The Kyverno admin configures their deployment to allow certain admission requests based on the data in the `imageData` data structure.
2. When the Kyverno admin deploys the rule to the cluster, Kyverno resolves the image reference from a remote registry and fetches the additional data and the digest.
3. An attacker intercepts the request and returns malicious data from the registry, including a digest that is different from the one that the Kyverno admin user provided in the rule.
4. Kyverno stores the data from the registry as well as other data parts from the image reference in the rule.

5. At this point, there are two attack vectors: Either the attacker makes their own request with a digest that is different from the digest from the initial rule, but that matches the digest in the `imageData` map. This request gets accepted. Alternatively, if the Kyverno use case allows it, a non-attacker makes an admission request to Kyverno with a digest that differs from the rule or the `imageData` map, and Kyverno mutates (rewrites) the digest to be the attacker-controlled digest.

The impact is limited to a small set of Kyverno users that fulfil all the following requirements:

1. The Kyverno user must reference images by digest.
2. The Kyverno user must make admission decisions based on the digest from the imageData and not the digest that the user referenced initially.

If the Kyverno admin initially deploys policies that reference images by tags, then there is no potential for increased privileges. #2 is important since the essence of this issue is that the returned digest and the digest initially used may differ. As such, when the Kyverno image verifier makes admission requests based on the digest returned from the registry, it can make decisions on a different digest than the one the Kyverno user expects to be making admission decisions against.

The attacker can make users consume the same image as the one from the initial rule but of a different digest. This is not a vulnerability in itself; For example, consuming the previous version of a MySQL image is not necessarily insecure. The risk arises when the attacker knows of an exploitable issue in different versions of images than the Kyverno user intends to consume. Alternatively, there can be scenarios where the attacker can release a new version of the image to the remote registry with the same name. If the attacker can achieve this level of control, they can achieve the highest level of impact for the Kyverno user; however, this requires that the attacker obtain the highest level of privilege in the registry.

### Advisory details

GHSA: GHSA-3hfq-cx9j-923w CVE: CVE-2023-47630

# Type Confusion in Jmespath Execution

| | |
|---|---|
| **Severity:** | Low |
| **Status:** | fixed |
| **Id:** | ADA-KYV-2023ATTB |
| **Component:** | go-jmespath |

Kyvernos use of `go-jmespath` in the image extraction utility is susceptible to a type confusion. On the following lines, Kyverno instantiates a new Jmespath `Interface`, creates a `Query` and searches the query. The type confusion was a result of casting a `nil`-value into a string.

https://github.com/kyverno/kyverno/blob/ae1fa9b2600302e3ba5f90d3312467f647a9e441/pkg/utils/api/image.go#L107-L112

```
107         jp := jmespath.New(cfg)
108          q, err := jp.Query(jmesPath)
109          if err != nil {
110                  return fmt.Errorf("invalid jmespath %s: %v", jmesPath, err)
111          }
112          result, err := q.Search(value)
```

With a well-crafted string, `github.com/kyverno/go-jmespath.(*treeInterpreter).Execute()` will trigger a type confusion.

STACKTRACE

```
        panic: interface conversion: interface {} is nil, not string [recovered]
  panic: interface conversion: interface {} is nil, not string
goroutine 17 [running, locked to thread]:
main.catchPanics()
  ./main.1065168555.go:49 +0x27c
panic({0x40e5200?, 0x10c000ab8690?})
  runtime/panic.go:914 +0x21f
github.com/kyverno/go-jmespath.(*treeInterpreter).Execute(0x10c000e5ca80, {0x4, {0x0,
0x0}, {0x10c000a31800, 0x9, 0x10}}, {0x3f6ea40, 0x6519ca0})
  github.com/kyverno/go-jmespath@v0.4.1-0.20230705123211-d067dc3d6613/interpreter.go:93
+0x4585
github.com/kyverno/go-jmespath.(*treeInterpreter).Execute(0x10c000e5ca80, {0x4,
{0x3f6ea40, 0x10c00107f370}, {0x10c0009f5e00, 0x7, 0x8}}, {0x3f6ea40, 0x6519ca0})
  github.com/kyverno/go-jmespath@v0.4.1-0.20230705123211-d067dc3d6613/interpreter.go:85
+0x43d0
github.com/kyverno/go-jmespath.(*JMESPath).Search(0x0?, {0x3f6ea40?, 0x6519ca0?})
  github.com/kyverno/go-jmespath@v0.4.1-0.20230705123211-d067dc3d6613/api.go:37 +0x8f
```

This part of Kyverno is invoked as part of the image verification by `imageVerifier.Verify`. Below, `HasChanged()` invokes the faulty execution path:

https://github.com/kyverno/kyverno/blob/ae1fa9b2600302e3ba5f90d3312467f647a9e441/pkg/engine/internal/imageverifier.go#L211-L240

```
211 func (iv *ImageVerifier) Verify(
212         ctx context.Context,
213         imageVerify kyvernov1.ImageVerification,
214         matchedImageInfos []apiutils.ImageInfo,
215         cfg config.Configuration,
216 ) ([]jsonpatch.JsonPatchOperation, []*engineapi.RuleResponse) {
217         var responses []*engineapi.RuleResponse
218         var patches []jsonpatch.JsonPatchOperation
219
220         // for backward compatibility
221         imageVerify = *imageVerify.Convert()
222
223         for _, imageInfo := range matchedImageInfos {
224                 image := imageInfo.String()
225
226                 if HasImageVerifiedAnnotationChanged(iv.policyContext, iv.logger) {
```

```
227                         msg := kyverno.AnnotationImageVerify + " annotation cannot
    be changed"
228                         iv.logger.Info("image verification error", "reason", msg)
229                         responses = append(responses,
    engineapi.RuleFail(iv.rule.Name, engineapi.ImageVerify, msg))
230                         continue
231                 }
232
233                 pointer := jsonpointer.ParsePath(imageInfo.Pointer).JMESPath()
234                 changed, err := iv.policyContext.JSONContext().HasChanged(pointer)
235                 if err == nil && !changed {
236                         iv.logger.V(4).Info("no change in image, skipping check",
    "image", image)
237                         iv.ivm.Add(image, true)
238                         continue
239                 }
240
```

This issue was found by way of fuzzing Kyvernos invocation of `go-jmespath`. The fuzzer tests `(*context).HasChanged` with a pseudo-random jmespath string, where the context has been initialized with pseudo-random strings for its objects. This is an over-approximation of the `HasChanged` API; Kyverno sanitizes the context objects before they reach this part of the image verification workflow. The `panic` resulting from this type confusion is recoverable and would not crash the Kyverno engine, nor would it have an effect on other users' ability to use the Kyverno engine; it would prevent the user sending the admission request from completing validation.