PRESENTS

Kyverno Fuzzing Audit

In collaboration with the Kyverno project maintainers and The Linux Foundation

Authors

Adam Korczynski <adam@adalogics.com>
David Korczynski <david@adalogics.com>
Date: 6th September 2023

CNCF security and fuzzing audits

This report details a fuzzing audit commissioned by the CNCF and the engagement is part of the broader efforts carried out by CNCF in securing the software in the CNCF landscape. Demonstrating and ensuring the security of these software packages is vital for the CNCF ecosystem and the CNCF continues to use state of the art techniques to secure its projects as well as carrying out manual audits. Over the last handful of years, CNCF has been investing in security audits, fuzzing and software supply chain security that has helped proactively discover and fix hundreds of issues.

Fuzzing is a proven technique for finding security and reliability issues in software and the efforts so far have enabled fuzzing integration into more than twenty CNCF projects through a series of dedicated fuzzing audits. In total, more than 350 bugs have been found through fuzzing of CNCF projects. The fuzzing efforts of CNCF have focused on enabling continuous fuzzing of projects to ensure continued security analysis, which is done by way of the open source fuzzing project OSS-Fuzz[1].

CNCF continues work in this space and will further increase investment to improve security across its projects and community. The focus for future work is integrating fuzzing into more projects, enabling sustainable fuzzer maintenance, increasing maintainer involvement and enabling fuzzing to find more vulnerabilities in memory safe languages. Maintainers who are interested in getting fuzzing integrated into their projects or have questions about fuzzing are encouraged to visit the dedicated cncf-fuzzing repository https://github.com/cncf/cncf-fuzzing where questions and queries are welcome.

[1] https://github.com/google/oss-fuzz

Executive summary

In this engagement, Ada Logics worked on setting up Kyrano's fuzzing suite. At the time of this engagement, Kyverno was not integrated into OSS-Fuzz, and the goal of this fuzzing audit was to first integrate Kyverno into OSS-Fuzz and then build upon this integration and improve the fuzzing efforts in a continuous manner.

We carried out the development of the fuzzers in the CNCF-Fuzzing repository and we moved the fuzzers upstream to Kyvernos own repository at the end of the engagement. Working from the CNCF-Fuzzing repository allowed the Ada Logics team to make smaller iterations of the fuzzers throughout the audit and avoid imposing the overhead of having the Kyverno maintainers review trivial changes to the fuzzers. OSS-Fuzz was instructed to pull the fuzzers from CNCF-Fuzzing in addition to the fuzzers from Kyvernos repository.

The fuzzers found 3 code issues during the audit itself demonstrating the value that the fuzzing suite adds to Kyverno. Over time, the fuzzers will explore more of the Kyverno code base, and they may report issues that exist in the code base during the audit but are hard to find.

Results summarised
15 fuzzers developed
All fuzzers added to Kyvernos OSS-Fuzz integration
All fuzzers added to Kyvernos upstream repository
3 bugs found during the audit

Table of Contents

Project Summary

Ada Logics auditors

Name	Title	Email
Adam Korczynski	Security Engineer	Adam@adalogics.com
David Korczynski	Security Researcher	David@adalogics.com

Kyverno maintainers involved in the audit

Name	Title	Email
Chip Zoller	Kyverno Maintainer	chipzoller@gmail.com
Jim bugwadia	Kyverno Maintainer	jim@nirmata.com

Assets

Url	Branch
https://github.com/kyverno/kyverno	`main`

Kyverno fuzzing

In this section we present details on the Kyverno fuzzing set up, and in particular the overall fuzzing architecture as well as the specific fuzzers developed.

Architecture

A central component in Kyvernos fuzzing suite is the element of continuity by way of OSS-Fuzz. The Kyverno source code and the source code for the Kyverno fuzzers are the two key software packages that OSS-Fuzz uses to fuzz Kyverno. The following figure gives an overview of how OSS-Fuzz uses these two packages and what happens when an issue is found/fixed.

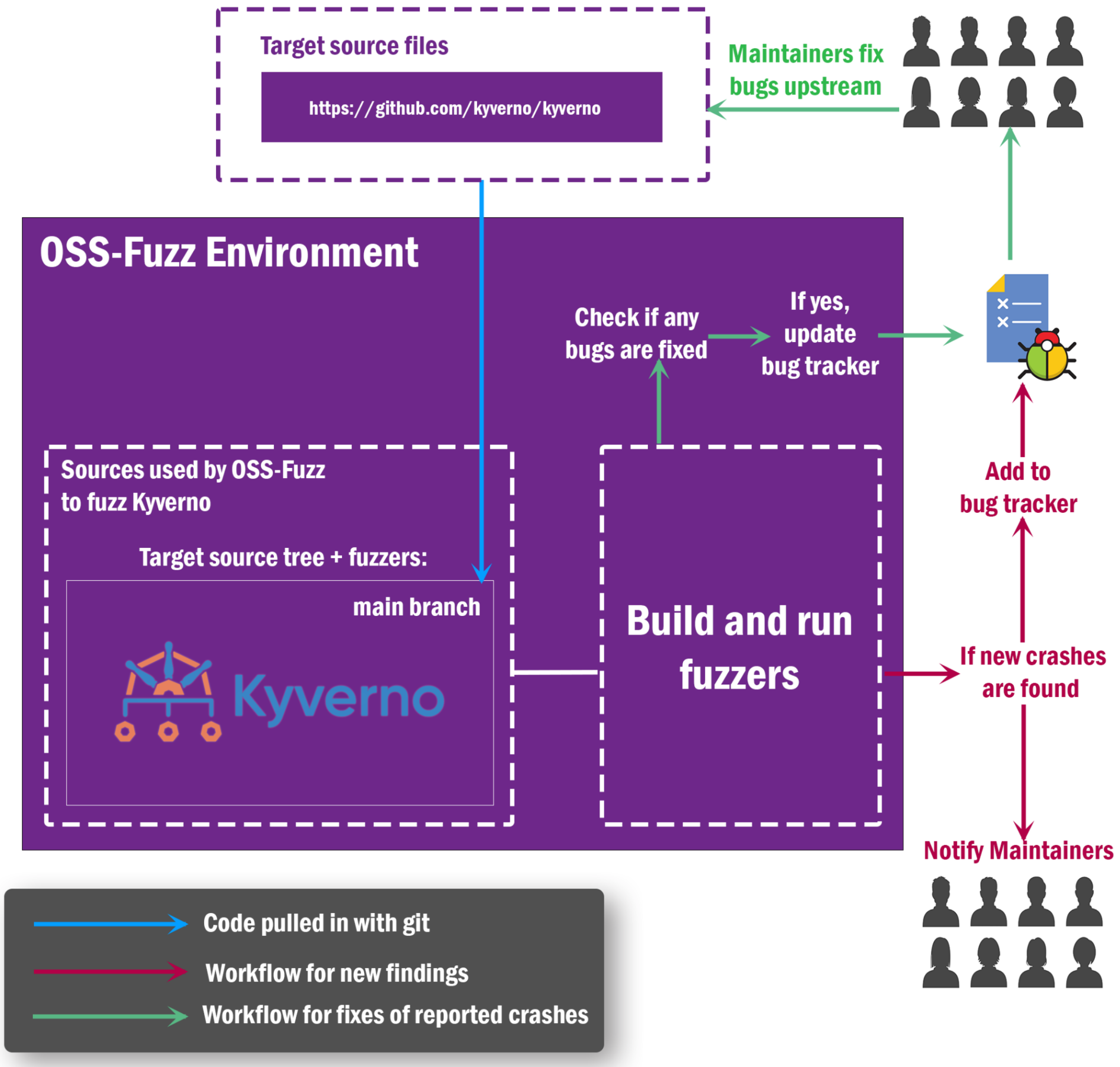
Target source files
https://github.com/kyverno/kyverno
Maintainers fix bugs upstream
OSS-Fuzz Environment
Check if any bugs are fixed
If yes, update bug tracker
Sources used by OSS-Fuzz to fuzz Kyverno
Target source tree + fuzzers:
main branch
Kyverno
Build and run fuzzers
Add to bug tracker
If new crashes are found
Notify Maintainers
Code pulled in with git
Workflow for new findings
Workflow for fixes of reported crashes

Figure 1.1: Kyvernos fuzzing architecture

The current OSS-Fuzz set up builds the fuzzers by cloning the upstream Kyverno Github repositories to get the latest Kyverno source code and the CNCF-Fuzzing Github repository to get the latest set of fuzzers, and then builds the fuzzers against the cloned Kyverno code. As such, the fuzzers are always run against the latest Kyverno commit.

This build cycle happens daily and OSS-Fuzz will verify if any existing bugs have been fixed. If OSS-fuzz finds that any bugs have been fixed OSS-Fuzz marks the crashes as fixed in the Monorail bug tracker and notifies maintainers.

In each fuzzing iteration, OSS-Fuzz uses its corpus accumulated from previous fuzz runs. If OSS-Fuzz detects any crashes when running the fuzzers, OSS-Fuzz performs the following actions:

1. A detailed crash report is created.
2. An issue in the Monorail bug tracker is created.
3. An email is sent to maintainers with links to the report and relevant entry in the bug tracker.

OSS-Fuzz has a 90 day disclosure policy, meaning that a bug becomes public in the bug tracker if it has not been fixed. The detailed report is never made public. The Kyverno maintainers will fix issues upstream, and OSS-Fuzz will pull the latest Kyverno main branch the next time it performs a fuzz run and verify that a given issue has been fixed.

Kyverno Fuzzers

In this section we enumerate the fuzzers that Ada Logics wrote during the fuzzing audit. A high level goal was to achieve a high level of test coverage, since Kyverno had no coverage prior to this fuzzing audit. Ada Logics analyzed the codebase initially both manually and by way of static analysis tooling to identify the optimal entrypoints to achieve the highest coverage without bloating the test suite with unnecessary fuzz tests. Our initial analysis found that the majority of the Kyverno code base is statically reachable from three API's in the `github.com/kyverno/kyverno/pkg/engine` package:

#	API
1	`github.com/kyverno/kyverno/pkg/engine.(*engine).Validate`
2	`github.com/kyverno/kyverno/pkg/engine.(*engine).VerifyAndPatchImages`
3	`github.com/kyverno/kyverno/pkg/engine.(*engine).Mutate`

Because these APIs have a lot of code to discover, we optimized the fuzzers for these entrypoints by structuring the data that the fuzzers pass to the entrypoints, such that the fuzzers reach deep into the call tree easier than with raw bytes. A substantial hurdle for the fuzzers was to generate valid policies with valid rules. We overcame this hurdle by implementing a utility function that creates policies and rules in a structured way in which the fuzzer decides how many rules to add, which specifications they should have and what the values of each specification should be.

Below we enumerate all fuzzers written during the audit.

Fuzzers

This table lists all fuzzers Ada Logics wrote during the engagement. The Name is the name of each fuzzer, and the package is both the package that the fuzzer targets as well as where it is hosted in the Kyverno source tree. The fuzzers are first listed in a table, and we then detail which Kyverno APIs they target specifically and how they generate the necessary parameters for those APIs.

#	Name	Package
1	`FuzzEvaluate`	`github.com/kyverno/kyverno/pkg/engine/variables`
2	`FuzzV2beta1PolicyValidate`	`github.com/kyverno/kyverno/api/kyverno/v2beta1`
3	`FuzzV2beta1ImageVerification`	`github.com/kyverno/kyverno/api/kyverno/v2beta1`
4	`FuzzV2beta1MatchResources`	`github.com/kyverno/kyverno/api/kyverno/v2beta1`
5	`FuzzV2beta1ClusterPolicy`	`github.com/kyverno/kyverno/api/kyverno/v2beta1`
6	`FuzzV1PolicyValidate`	`github.com/kyverno/kyverno/api/kyverno/v1`
7	`FuzzV1ImageVerification`	`github.com/kyverno/kyverno/api/kyverno/v1`
8	`FuzzV1MatchResources`	`github.com/kyverno/kyverno/api/kyverno/v1`
9	`FuzzV1ClusterPolicy`	`github.com/kyverno/kyverno/api/kyverno/v1`
10	`FuzzVerifyImageAndPatchTest`	`github.com/kyverno/kyverno/pkg/engine`
11	`FuzzEngineValidateTest`	`github.com/kyverno/kyverno/pkg/engine`
12	`FuzzMutateTest`	`github.com/kyverno/kyverno/pkg/engine`
13	`FuzzValidatePolicy`	`github.com/kyverno/kyverno/pkg/validation/p`

		`olicy`
14	`FuzzAnchorParseTest`	`github.com/kyverno/kyverno/pkg/engine/anchor`
15	`FuzzEngineResponse`	`github.com/kyverno/kyverno/pkg/engine/api`

FuzzEvaluate
Tests Kyvernos condition evaluation routine, `github.com/kyverno/kyverno/pkg/engine/variables.Evaluate`. The fuzzer creates a condition type with a random key, a known operator and a random value and then passes the condition onto the evaluation routine.

FuzzV2beta1PolicyValidate
Tests Kyvernos validation API for v2beta1 policies. The fuzzer creates a random policy and invokes its `Validate()` method.

FuzzV2beta1ImageVerification
Tests Kyvernos validation API for the v2beta1 `ImageVerification` type. The fuzzer creates a random `ImageVerification` struct and invokes its `Validate()` method.

FuzzV2beta1MatchResources
Tests Kyvernos validation APIs for the v2beta1 MatchResources type. The fuzzer creates a random `MatchResources` struct and invokes its `ValidateResourceWithNoUserInfo()` and `Validate()` methods.

FuzzV2beta1ClusterPolicy
This fuzzer tests multiple methods of the v2beta1 `ClusterPolicy` type. The fuzzer creates a random `ClusterPolicy` and invokes the following methods:
- `HasAutoGenAnnotation()`
- `HasMutateOrValidateOrGenerate()`
- `HasMutate()`
- `HasValidate()`
- `HasGenerate()`
- `HasVerifyImages()`
- `AdmissionProcessingEnabled()`
- `BackgroundProcessingEnabled()`
- `Validate(nil)`

FuzzV1PolicyValidate
Tests Kyvernos validation API for v1 policies. The fuzzer creates a random policy and invokes its `Validate()` method.

FuzzV1ImageVerification
Tests Kyvernos validation API for the v1 `ImageVerification` type. The fuzzer creates a random `ImageVerification` type and invokes its `Validate()` method.

FuzzV1MatchResources
Tests Kyvernos validation APIs for the v1 `MatchResources` type. The fuzzer creates a random `MatchResources` struct and invokes its `ValidateResourceWithNoUserInfo()` and `Validate()` methods.

FuzzV1ClusterPolicy
This fuzzer tests multiple methods of the v1 `ClusterPolicy` type. The fuzzer creates a random `ClusterPolicy` and invokes the following methods:

- `HasAutoGenAnnotation()`
- `HasMutateOrValidateOrGenerate()`
- `HasMutate()`
- `HasValidate()`
- `HasGenerate()`
- `HasVerifyImages()`
- `AdmissionProcessingEnabled()`
- `BackgroundProcessingEnabled()`
- `Validate(nil)`

FuzzVerifyImageAndPatchTest
Tests `github.com/kyverno/kyverno/pkg/engine.(*engine).VerifyAndPatchImages` with a randomized policy, resource and old resource. The fuzzer first creates a context which contains the Kyverno policy and the two Kubernetes resources. It then passes the context onto `github.com/kyverno/kyverno/pkg/engine.(*engine).VerifyAndPatchImages`.

FuzzEngineValidateTest
Tests `github.com/kyverno/kyverno/pkg/engine.(*engine).Validate` with a randomized policy and resource. The fuzzer first creates a policy and then a policycontext which contains the Kubernetes resource. It then passes both onto `github.com/kyverno/kyverno/pkg/engine.(*engine).Validate`.

FuzzMutateTest
Tests `github.com/kyverno/kyverno/pkg/engine.(*engine).Mutate` with a randomized policy and resource. The fuzzer first creates a policy and a resource. It then creates a policycontext which contains the Kubernetes resource. It then passes both onto `github.com/kyverno/kyverno/pkg/engine.(*engine).Mutate`.

FuzzValidatePolicy
Tests `github.com/kyverno/kyverno/pkg/validation/policy.Validate` with a randomized Kyverno `ClusterPolicy`.

FuzzAnchorParseTest
Tests the parsing routine for anchors in the `github.com/kyverno/kyverno/pkg/engine/anchor` package. The parser is based on a regular expression and the fuzzer tests this regex'es safety.

FuzzEngineResponse
Tests the engine response type and whether a malicious resource can cause disruption when creating an engine response or when the created response invokes any of its methods.

Issues found during audit

The fuzzers found 3 issues during the audit all of which Ada Logics fixed in Kyverno. The three issues had their root cause in the Kyverno code base.

Issue 1: Type confusion in policy validation routine

OSS-Fuzz bug tracker:	https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=60714
Mitigation:	Fixed in https://github.com/kyverno/kyverno/pull/7857
ID:	ADA-KYV-FUZZ-1

A fuzzer was able to pass a policy to `github.com/kyverno/kyverno/pkg/validation/policy.Validate` that triggered a type confusion on the lines below:

https://github.com/kyverno/kyverno/blob/dfceb4bf821feea15c2d86b2f76f2a4448fc1582/pkg/validation/policy/validate.go#L498

```
func cleanup(policy kyvernov1.PolicyInterface) kyvernov1.PolicyInterface {
	ann := policy.GetAnnotations()
	if ann != nil {
		ann["kubectl.kubernetes.io/last-applied-configuration"] = ""
		policy.SetAnnotations(ann)
	}
	if policy.GetNamespace() == "" {
		pol := policy.(*kyvernov1.ClusterPolicy)
		pol.Status.Autogen.Rules = nil
		return pol
	} else {
		pol := policy.(*kyvernov1.Policy)
		pol.Status.Autogen.Rules = nil
		return pol
	}
}
```

The issue was fixed by adding a type check before casting. When triaging this issue, we found another similar issue which we also fixed in the same PR.

Issue 2: Assignment to nil-map in policy validation

OSS-Fuzz bug tracker:	https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=60759
Mitigation:	Fixed in https://github.com/kyverno/kyverno/pull/7874
ID:	ADA-KYV-FUZZ-2

The policy validation fuzzer found an assignment to a nil-map when Kyverno validates the namespace on the highlighted line below:

https://github.com/kyverno/kyverno/blob/7647a1632dd6af71ff35d001edaff88e874a0708/pkg/validation/policy/validate.go#L1280

```
func validateNamespaces(s *kyvernov1.Spec, path *field.Path) error {
	action := map[string]sets.Set[string]{
		"enforce":  sets.New[string](),
		"audit":    sets.New[string](),
		"enforceW": sets.New[string](),
		"auditW":   sets.New[string](),
	}

	for i, vfa := range s.ValidationFailureActionOverrides {
		patternList, nsList := wildcard.SeperateWildcards(vfa.Namespaces)

		if vfa.Action.Audit() {
			if action["enforce"].HasAny(nsList...) {
				return fmt.Errorf("conflicting namespaces found in path:
%s: %s", path.Index(i).Child("namespaces").String(),

strings.Join(sets.List(action["enforce"].Intersection(sets.New(nsList...))), ", "))
			}
			action["auditW"].Insert(patternList...)
		} else if vfa.Action.Enforce() {
			if action["audit"].HasAny(nsList...) {
				return fmt.Errorf("conflicting namespaces found in path:
%s: %s", path.Index(i).Child("namespaces").String(),

strings.Join(sets.List(action["audit"].Intersection(sets.New(nsList...))), ", "))
			}
			action["enforceW"].Insert(patternList...)
		}
		action[strings.ToLower(string(vfa.Action))].Insert(nsList...)
```

At this point in the execution, the action map did dot have an action corresponding to `strings.ToLower(string(vfa.Action))` which resulted in the line effectively being `nil.Insert(nsList...)`. The fix was to validate the `vfa.Action` further up in the function body.

Issue 3: Missing fallback results in a nil-dererence panic

OSS-Fuzz bug tracker:	https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=62032
Mitigation:	Fixed in https://github.com/kyverno/kyverno/pull/8271
ID:	ADA-KYV-FUZZ-3

In this part of the PSS handler, `getSpec` could return `nil, nil, nil` which resulted in a Go nil-dereference panic on the highlighted lines:

https://github.com/kyverno/kyverno/blob/34bfb57c084bc7364f9a7a0cbce3626ea67e090d/pkg/engine/handlers/validation/validate_pss.go#L26-L46

```
func (h validatePssHandler) Process(
	ctx context.Context,
	logger logr.Logger,
	policyContext engineapi.PolicyContext,
	resource unstructured.Unstructured,
	rule kyvernov1.Rule,
	_ engineapi.EngineContextLoader,
) (unstructured.Unstructured, []engineapi.RuleResponse) {
	// Marshal pod metadata and spec
	podSecurity := rule.Validation.PodSecurity
	if resource.Object == nil {
		resource = policyContext.OldResource()
	}
	podSpec, metadata, err := getSpec(resource)
	if err != nil {
		return resource, handlers.WithError(rule, engineapi.Validation, "Error
while getting new resource", err)
	}
	pod := &corev1.Pod{
		Spec:       *podSpec,
		ObjectMeta: *metadata,
	}
```

The reason was that `github.com/kyverno/kyverno/pkg/engine/handlers/validation.getSpec` checked the resource type but was missing a fallback in case it was not a DaemonSet, Deployment, Job, StatefulSet, ReplicaSet, ReplicationController, CronJob or a Pod. The below code snippet shows the fix highlighted with green:

https://github.com/kyverno/kyverno/blob/27566eb3fa45ee397e3e2b57af5938660146b4e0/pkg/engine/handlers/validation/validate_pss.go#L69-L118

```
func getSpec(resource unstructured.Unstructured) (podSpec *corev1.PodSpec, metadata
*metav1.ObjectMeta, err error) {
	kind := resource.GetKind()
```

```go
	if kind == "DaemonSet" || kind == "Deployment" || kind == "Job" || kind ==
"StatefulSet" || kind == "ReplicaSet" || kind == "ReplicationController" {
		var deployment appsv1.Deployment

		resourceBytes, err := resource.MarshalJSON()
		if err != nil {
			return nil, nil, err
		}
		err = json.Unmarshal(resourceBytes, &deployment)
		if err != nil {
			return nil, nil, err
		}
		podSpec = &deployment.Spec.Template.Spec
		metadata = &deployment.Spec.Template.ObjectMeta
		return podSpec, metadata, nil
	} else if kind == "CronJob" {
		var cronJob batchv1.CronJob

		resourceBytes, err := resource.MarshalJSON()
		if err != nil {
			return nil, nil, err
		}
		err = json.Unmarshal(resourceBytes, &cronJob)
		if err != nil {
			return nil, nil, err
		}
		podSpec = &cronJob.Spec.JobTemplate.Spec.Template.Spec
		metadata = &cronJob.Spec.JobTemplate.ObjectMeta
	} else if kind == "Pod" {
		var pod corev1.Pod

		resourceBytes, err := resource.MarshalJSON()
		if err != nil {
			return nil, nil, err
		}
		err = json.Unmarshal(resourceBytes, &pod)
		if err != nil {
			return nil, nil, err
		}
		podSpec = &pod.Spec
		metadata = &pod.ObjectMeta
		return podSpec, metadata, nil
	} else {
		return nil, nil, fmt.Errorf("Could not find correct resource type")
	}

	if err != nil {
		return nil, nil, err
	}
	return podSpec, metadata, err
}
```