Validating the replacement filtering features of popular alternative admission controllers for Pod Security Policies

Research Project 2 Report

Maarten van der Slik
mslik@os3.nl
Security and Network Engineering
University of Amsterdam

Frank Wiersma
fwiersma@os3.nl
Security and Network Engineering
University of Amsterdam

July 5, 2021

***Abstract*—Kubernetes is the most common container orchestration tool, which is used to run applications of differentiating importance & confidentiality. It runs applications in "pods". Kubernetes provides the admission controller named PodSecurityPolicies to limit privileges for these pods. This is of significant importance as a default Kubernetes cluster is prone to dangerous privilege escalation attacks. This study anticipates the announcement for the deprecation of Kubernetes PodSecurityPolicies. We look at three popular alternative admission controllers, suggested by the Kubernetes security group, namely Gatekeeper, Kyverno, and k-rail. First, this study assesses the requirements for alternative admission controllers through literature study and technical experiments. Afterward, we assess the ability of other controllers to replace PSP functions. We highlight the similarities and shortcomings of alternative admission controllers relative to PodSecurityPolicies. This study concludes that the alternative admission controllers called Gatekeeper and Kyverno are, except a few aspects, worthy replacements, contrary to k-rail, which does not suffice most of the requirements.**

I. Introduction

Kubernetes is the de facto container orchestration system which is introduced in 2015 [1].

By default, the Kubernetes allows for a variety of privilege escalation routes, which allows attackers to unintentionally receive root permissions [2][3]. Therefore, the open-source project, supported by the Cloud Native Computing Foundation (CNCF), introduced a new security feature in version 1.4 (2016) called Pod Security Policies (PSP) [4]. This feature is implemented as an admission controller [5] and allows cluster administrators to enforce a security configuration baseline across a namespace and therefore limiting the number of pod capabilities that users and groups can configure.

Kubernetes admission controllers provide a way to inspect and validate and mutate requests sent to the API server. The admission control process consists of two phases: the mutating phase is executed first, followed by the validating phase [6]. See Figure 1.

The PSP admission controller specifically, acts on the creation and modification of the pod and determines if a pod should be admitted (validated) or changed (mutated) based on the requested security context and the available Pod Security Policies [7].

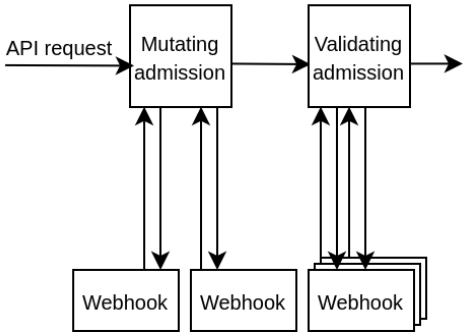

Fig. 1: kube-apiserver phases simplified, only admission controller operations are shown [6]

However, Kubernetes declared the deprecation of the feature as of version 1.21 in a blog post released on April 06, 2021, stating that the system is broadly considered to be "confusing" [3]. Compliant with the deprecation policy, Pod Security Policy will continue to be fully functional for several more releases, and will be removed in v1.25. At that point, one should already have migrated. [3] [8]

The blog post also mentions the development of a new security policy mechanism, temporarily named the "PSP Replacement Policy". (KEP-2579) The team is currently building an Alpha version of this system. However, Kubernetes states that the new system functionality will be simplistic and limited. [3] and suggests looking into three alternative projects.

These projects differ in size, popularity, programming language, and applications. We will introduce each of them, ordered by their popularity based on the GitHub repository stars.

Gatekeeper, 1852 stars [9]:
Gatekeeper is an open-source project that was accepted in 2018 by the Cloud Native Computing Foundation [10]. Gatekeeper is the Kubernetes-specific implementation of Open Policy Agent (OPA), a general-purpose policy engine [11]. Because of the relationship between Open Policy Agent with Gatekeeper, the project is often written "OPA/Gatekeeper". Gatekeeper enables request validation and, most recently, mutation `(alpha)` [12]. Another property of Gatekeeper is its reliance on a high-level declarative language called Rego, which enables writing policies as code [13]. Lastly, it offers an audit mode, contrary to its predecessor OPA [11].

Kyverno, 1088 stars [14]:
Kyverno uses policies which are Kubernetes resources. The admission controller can validate and mutate [15]. In contrast to Gatekeeper, Kyverno does not require administrators to learn a new policy language. The policies are written in YAML. Kyverno comes with an audit mode, an important and useful feature that PSP lacked. [16]

k-rail, 380 stars [17]:
k-rail is an admission controller developed by Cruise Automation. It also supports mutation and validation. k-rail comes with a default configuration, where a lot of policies are enforced. k-rail takes a broader approach as it has policies for Kubernetes resources outside the pod context. Lastly, the existence of an audit mode is noteworthy, which logs policy violations. [18]

This study compares and validates the features of the mentioned alternative projects, suggested by the Kubernetes Security SIG (Special Interest Group). Furthermore, we find out if they are capable of imitating the PSP functions and show security directives that they lack, and present solutions to mitigate these shortcomings.

A. Research Question

The main research question we defined is:

"Are Gatekeeper, Kyverno, and k-rail able to cover all validation and mutation functionalities of the deprecated Kubernetes' Pod Security Policies?"

To formulate an answer to this question, the following sub-questions need to be answered:

1) *Which operations are executed by PSP functions to validate or mutate fields and values in the API request?*
2) *Do the alternative admission controllers offer the necessary features to replace the PSP functions?*

B. Justification

The need for the first sub-question is caused by the absence of certain details in the PSP documentation and API reference. E.g., the documentation about "MustRunAs" function rules does not describe if the pod specification (PodSpec) or the container specification is mutated, the documentation about `readOnlyRootFilesystem` does not describe if pods are mutated or only validated and no information is given about the validation and mutation of replaced annotations of beta functions that became generally available in the past or will become in the future. [5][19] To be able to imitate a PSP function completely, the exact behavior of PSP functions must be gathered.

The need for the second sub-question is caused by the lack of research, described in the next section, Section II. Although a lot of features and policy templates are available from the community-owned Gatekeeper Library [20] and Kyverno website [21], and k-rail also offers a lot of features [18], it is not clear if they cover the complete set of PSP function and replicate the exact behavior of the functions.

C. Structure

The remainder of this paper has the following structure. In section II we look at highlights of work done by others that relate to ours. In section III we define our approach to gather the PSP behavior and the corresponding requirements to replace the PSP functions. In section IV we present findings of the literature research and technical experiments. In section V we discuss our findings. Using our findings, we will conclude and present those in section VI. The last section, VII, contains suggestions for future work.

II. Related Work

Researchers have addressed container privilege escalation problems extensively.

The first study in this field that has received notable attention was conducted by Lin et al [22]. in 2018. Lin et al. underlined the importance of kernel security mechanisms such as capabilities, seccomp, SELinux, and AppArmor and their role in preventing privilege escalation, in addition to the default container isolation mechanisms (i.e., namespaces and cgroups). In 2019, Artem et al. also addressed the issue, by demonstrating the risks of the root containers in Kubernetes and mentioning security policies as mitigation solutions [2]. Artem et al. suggested which parameters in the container manifest would mitigate the issue. In 2020, Shamim et al. gathered best practices for Kubernetes security and presented them in the paper XI Commandments of Kubernetes Security [23]. One of the most common best practices is the implementation of policies and the use of admission controllers. Both network and pod security policies are covered in this paper, and Shamim et al. suggested configuring the right per-pod security context.

While these studies signify the importance of security policies and container isolation, they do not specifically discuss (the purpose of) PSP. According to Kubernetes, PSPs enable the cluster administrators to configure an upper-bound 'securityContext' for pods, hence limiting the permissions users, groups, and service accounts can give to pod configurations [5]. The necessity for PSP to prevent privilege escalation problems was underlined in 2018, when Bischoff created a framework to validate Kubernetes policies. Bischoff described the functionality, configuration possibilities, and relevance of

PSP extensively. In 2019, Megino et al. described briefly how PSP ensured security in their ATLAS computing site. Lastly, in 2020, Panagiotis described offensive and defense methods for Kubernetes, in which PSP is used to prevent vulnerable objects from being deployed.

While these studies emphasize the significance of preventing privilege escalation issues and point out the mitigation solution PSP, studies on potential replacement options for PSP are missing. In a weblog Zoller compared two admission controllers with PSP in general, although an unintended bias should be taken into consideration as the author is an contributor of the Kyverno project. Moreover, Zoller did not compare the possibilities of Gatekeeper and Kyverno with PSP. To conclude, a comparison in detail is missing. Because PSP is deprecated and the (potentialy limited [3]) successor has not been released yet, this particularly neglected area should be covered to ensure cluster administrators can replace PSP in the near future. Therefore, we compare, test and evaluate the alternative options.

III. Methodology

In this section, we outline our research method that we perform to answer the research questions, as described in section I-A.

A. Phase 1 - Requirement assessment for PSP replacement functions

During the first phase of our research, we assess the requirements for potential replacement controllers that we derive from an in-depth investigation of the actual operations performed by the PSP functions. To gather these requirements, we first acquire the information available from the Kubernetes API reference [19] (version 1.20) and the PSP documentation[5]. With this information, we describe the operations that are performed to validate and/or mutate fields in an API request.

However, as we described in section I: the documentation is incomplete some areas. In that case, we perform technical experiments with PSP, as is described in the subsection III-C of this section.

B. Phase 2 - Assessment of replacement capabilities

Subsequently, we assess if the alternative admission controllers can meet the requirements to replace PSP functions. As mentioned in section I, we research the most popular admission controllers on GitHub, which are also suggested by the Kubernetes Security SIG [3]. The approach differs per admission controller type. While Gatekeeper and Kyverno policies are based on templates, k-rail offers a fixed set of policies.

Therefore, we first study the template configuration capabilities for Gatekeeper and Kyverno, based on literature research. We also analyze the behavior of templates that the Gatekeeper provides in their Gatekeeper Library [20] and Kyverno provides in their Policies webpage [21]. Based on the PSP function list [5] and behavioral requirements we listed during the first phase of our research, we analyze the available sample templates and conclude if they might be capable of replacing a PSP feature. If that is not the case, we find out if the policy templates can be created with the available configuration logic.

Afterward, we verify the sample or created templates by performing technical experiments, which are described in more detail in subsection III-C of this section.

The offered sample templates can be retrieved from the following locations:

- **Gatekeeper:**
  - Sample validating templates: https://github.com/open-policy-agent/gatekeeper-library/tree/master/src/pod-security-policy
  - Sample mutating templates: https://github.com/open-policy-agent/gatekeeper-library/tree/master/library/experimental/mutation/pod-security-policy
- **Kyverno:**
  - https://kyverno.io/policies/pod-security/
- **k-rail:**
  - https://github.com/cruise-automation/k-rail/tree/master/policies/pod

Secondly, we study the k-rail features and their behavior, based on literature research. When k-rail offers a feature that might meet the requirements we listed during the first phase of our research, we verify the template by performing technical experiments, which are described in more detail in the following subsection.

C. Technical experiments

In the technical experiments, many variations of pod objects are tested to indicate and note the behavior of the feature admission controllers (PSP and k-rail) or a sample template or created template (Gatekeeper and Kyverno).

When we analyze a validation function or function rule, all allowable and disallowable pod objects are tested:

- Forbidden values
- Allowed values
- Corresponding API fields values absent
- When values are validated on multiple locations (described in API reference), all possible variations of forbidden or allowed pod objects are tested
- When a field can contain a large number of possible values, example values or edge values are tested

When we analyze a mutation function or function rule, all allowable and disallowable pod objects are tested:

- Corresponding API fields contain values
- Corresponding API fields values absent
- When values can be mutated on multiple locations (described in API reference), all possible variations of pod objects are tested

When features are both validating and mutating, we perform both the validating and mutating tests, as we just described. Futhermore, pod specifications in Deployments, Statefulsets and Daemonsets are also verified. After all PSP functions and

replacement candidates are verified, the results are presented in a table.

D. Experimental set-up

We perform the experiments on a experimental environment, based on virtual machines with Ubuntu 18.04 LTS, Docker and minikube running Kubernetes 1.20.2. Each setup will contain its own admission controller: PodSecurityPolicies, Gatekeeper 3.4.0, Kyverno 1.3.6 and k-rail 3.4.1. See Figure 2.

Fig. 2: Experimental environment

IV. Results

In this section, we present the results of our research. The section is divided into two subsections, based on the two research phases of the research.

The ordering of the PSP functions is based on the table in the PSP documentation page [5].

A. Phase 1 - Requirement assessment

During the first research phase, we gathered the requirements that needs to be met to replicate PSP functions. For each PSP function, a description is given about the configuration options and operations that are performed. Functions can validate one or multiple fields, mutate one or multiple fields.

1) privileged:
Configuration: `privileged` boolean value can be specified to allow the pod to run privileged containers
Validation: when enabled, the pod can contain an enabled `spec.containers[].securityContext.privileged` boolean value

2) hostPID:
Configuration: `hostPID` boolean value can be specified to determine if the pod can share the host' PID namespace
Validation: when enabled, the pod can contain an enabled `spec.hostPID` boolean value

3) hostIPC:
Configuration: `hostIPC` boolean value can be specified to determine if the pod can use the host's IPC namespace
Validation: when enabled, the pod can contain an enabled `spec.hostIPC` boolean value

4) hostNetwork:
Configuration: `hostNetwork` boolean value can be specified to determine if the pod can use the host's network namespace
Validation: when enabled, the pod can contain an enabled `spec.hostNetwork` boolean value

5) hostPorts:
Configuration: `hostPort` array with multiple integer ranges can be configured to determine which the ports the container can expose to the underlying host
Validation: when enabled, the pod is allowed to contain a `spec.containers[].ports.hostPort` integer value within the ranges that are specified in the policy

6) volumes:
Configuration: `volumes` string array can be used to determine which volume plugins can be used
Validation: when configured, the pod can contain field in the `spec.volumes[]` array that are specified in the policy

7) allowedHostPaths:
Configuration: `allowedHostPaths` array can be used to specify which path the host volume can contain. The value in the policy can also be the prefix of the complete path, as long as the path starts with a trailing slash after the prefix. Also, the policy must be able to ensure that connected volumeMounts are set to readOnly.
Validation: when configured, the `spec.volumes[].hostPath` volume will be allowed if the `spec.volumes[].hostPath.path` string value is matches an array item within the policy and connected volumeMounts are set to readOnly (`spec.containers[].VolumeMounts[].readOnly`) if this is configured in the corresponding policy.

8) allowedFlexVolumes:
Configuration: the `allowedFlexVolumes` array contains the `allowedFlexVolumes[].driver` field in which a string can be specified to configure which FlexVolume driver are allowed in FlexVolume volumes. Multiple "driver" items can occur in the array.
Validation: when configured, the `spec.volumes[].flexVolume.driver` string value is validated

9) fsGroup MustRunAs:
Configuration: `fsGroup` contains the `fsGroup.ranges` array, in which multiple ranges can be specified using `min` and `max` integers to allow multiple ranges of FSGroups. When the `rule` string option is set to MustRunAs, the policy must contain at least one FSGroup range.
Mutation: The minimum value of the first range is applied as value if no value is given in `spec.securityContext.fsgroup`
Validation: MustRunAs will validate if the `spec.securityContext.fsgroup` value is within one of

the specified policy ranges.

10) fsGroup MayRunAs:
Configuration: `fsGroup` contains the `fsGroup.ranges` array, in which multiple ranges can be specified using `min` and `max` integers to allow multiple ranges of FSGroups. When the `rule` string option is set to MayRunAs, the policy must contain at least one FSGroup range.
Validation: MayRunAs will only validate `spec.securityContext.fsgroup` if a value is specified. MayRunAs will validate if the specified `spec.securityContext.fsgroup` value is within one of the specified policy ranges.

11) fsGroup RunAsAny:
Configuration: `fsGroup` contains, inter alia, the `rule` string option. When the RunAsAny rule is specified, validation and mutation of FSGroup is disabled.

12) readOnlyRootFilesystem:
Configuration: `readOnlyRootFilesystem` is used to enforce the use of a read only root filesystem.
Mutation: `spec.containers[].securityContext.readOnlyRootFilesystem` will be enabled if no value is given.
Validation: `spec.containers[].securityContext.readOnlyRootFilesystem` must be enabled

13) runAsUser MustRunAs:
Configuration: `runAsUser` contains the `runAsUser.ranges` array, in which multiple ranges can be specified using `min` and `max` integers to allow multiple ranges of user identifiers. When the `rule` string option is set to MustRunAs, the policy must contain at least one user identifier range.
Mutation: The minimum value of the first range is applied to the `spec.container[].securityContext.runAsUser` value if no value is given in `spec.securityContext.runAsUser` or `spec.container[].securityContext.runAsUser`
Validation: MustRunAs will validate if the `spec.securityContext.runAsUser` and `spec.containers[].securityContext.runAsUser` values are within one of the specified policy ranges.

14) runAsUser RunAsAny:
Configuration: `runAsUser` contains, inter alia, the `rule` string option. When the RunAsAny rule is specified, validation and mutation of runAsUser is disabled.

15) runAsUser MustRunAsNonRoot:
Configuration: `runAsUser` contains, inter alia, the `rule` string option. When the MustRunAsNonRoot rule is specified, the container must contain a non-zero runAsUser integer or have a non-root `USER` variable configured in the container image.
Mutation: enables `spec.containers[].securityContext.runAsNonRoot` if the following fields are not specified:

- `spec.securityContext.runAsUser`
- `spec.containers[].securityContext.runAsUser`
- `spec.securityContext.runAsNonRoot`
- `spec.containers[].securityContext.runAsNonRoot`

Validation: `spec.securityContext.runAsUser` or `spec.containers[].securityContext.runAsUser` must be set to a non-zero integer, or `spec.securityContext.runAsNonRoot` or `spec.containers[].securityContext.runAsNonRoot` must be enabled

16) runAsGroup MustRunAs:
Configuration: `runAsGroup` contains the `runAsGroup.ranges` array, in which multiple ranges can be specified using `min` and `max` integers to allow multiple ranges of group identifiers. When the `rule` string option is set to MustRunAs, the policy must contain at least one group identifiers range.
Mutation: The minimum value of the first range is applied to `spec.containers[].securityContext.runAsGroup` if no value is set in neither `spec.securityContext.runAsGroup` nor `spec.containers[].securityContext.runAsGroup`
Validation: MustRunAs will validate if the values of

- `spec.securityContext.runAsGroup`
- `spec.containers[].securityContext.runAsGroup`

are within one of the specified policy ranges.

17) runAsGroup MayRunAs:
Configuration: `runAsGroup` contains the `runAsGroup.ranges` array, in which multiple ranges can be specified using `min` and `max` integers to allow multiple ranges of group identifiers. When the `rule` string option is set to MayRunAs, the policy must contain at least one group identifiers range.
Validation: MayRunAs will only validate

- `spec.securityContext.runAsGroup`
- `spec.containers[].securityContext.runAsGroup`

if a value is specified. MayRunAs will validate if the specified values of the following fields are within one of the specified policy ranges.

- `spec.securityContext.runAsGroup`
- `spec.containers[].securityContext.runAsGroup`

18) runAsGroup RunAsAny:
Configuration: `runAsGroup` contains, inter alia, the `rule` string option. When the RunAsAny rule is specified,

validation and mutation of runAsGroup is disabled.

19) supplementalGroups MustRunAs:
Configuration: `supplementalGroups` contains the `supplementalGroups.ranges` array, in which multiple ranges can be specified using `min` and `max` integers to allow multiple ranges of group identifiers. When the `rule` string option is set to MustRunAs, the policy must contain at least one group identifiers range.
Mutation: The minimum value of the first range is applied if no value is specified in `spec.securityContext.supplementalGroups`
Validation: MustRunAs will validate if the `spec.securityContext.supplementalGroups` value is within one of the specified policy ranges.

20) supplementalGroups MayRunAs:
Configuration: `supplementalGroups` contains the `supplementalGroups.ranges` array, in which multiple ranges can be specified using `min` and `max` integers to allow multiple ranges of group identifiers. When the `rule` string option is set to MayRunAs, the policy must contain at least one group identifiers range.
Validation: MayRunAs will only validate `spec.securityContext.supplementalGroups` if a value is specified. MayRunAs will validate if the specified `spec.securityContext.supplementalGroups` value is within one of the specified policy ranges.

21) supplementalGroups RunAsAny:
Configuration: `supplementalGroups` contains, inter alia, the `rule` string option. When the RunAsAny rule is specified, validation and mutation of supplementalGroups is disabled.

22) allowPrivilegeEscalation:
Configuration: `allowPrivilegeEscalation` boolean value can be specified to allow privilege escalation capabilities. Capabilities such as *cap_sys_admin* still need to be dropped explicitly.
Validation: when enabled, the pod can contain an enabled `spec.containers[].securityContext.allowPrivilegeEscalation` value.

23) defaultAllowPrivilegeEscalation:
Configuration: `defaultAllowPrivilegeEscalation` boolean value can be specified to control the default AllowPrivilegeEscalation value
Mutation: configures a `spec.containers[].securityContext.allowPrivilegeEscalation` boolean if no value is specified.

24) allowedCapabilities:
Configuration: `allowedCapabilities` string array can be used to whitelist additional Linux Capabilities
Validation: strings in the `spec.containers[].securityContext.capabilities.add` array must be explicitly allowed in the policy.

25) requiredDropCapabilities:
Configuration: `requiredDropCapabilities` string array can be used to drop capabilities explicitly from the containers, in a case-insensitive manner.
Mutation: appends a set of strings to the `spec.containers[].securityContext.capabilities.drop` array
Validation: rejects pods that contain strings in the `spec.containers[].securityContext.capabilities.add` array that are specified in the policy.

26) defaultAddCapabilities:
Configuration: `defaultAddCapabilities` string array can be used to specify a default set of capabilities that are added to the containers. Capabilities that are specified in `spec.containers[].securityContext.capabilities.drop` (upper or lower case) will not be added.
Mutation: strings will be appended to the `spec.containers[].securityContext.capabilities.add` string array
Validation: string will be appended if they are absent from the `spec.containers[].securityContext.capabilities.drop` string array

27) seLinux MustRunAs:
Configuration: `seLinux` contains the `seLinux.seLinuxOptions` array, in which the SELinux labels can be specified. When the `rule` string option is set to MustRunAs, the policy must contain the seLinuxOptions.
Mutation: The seLinuxOptions of the policy are applied to `spec.securityContext.seLinuxOptions` if no value is specified.
Validation: MustRunAs will validate if the `spec.securityContext.seLinuxOptions` and `spec.containers[].securityContext.seLinuxOptions` values correspond to the SELinux policy options.

28) seLinux RunAsAny:
Configuration: `seLinux` contains, inter alia, the `rule` string option. When the RunAsAny rule is specified, validation and mutation of SELinux labels is disabled.

29) allowedProcMountTypes:
Configuration: `allowedProcMountTypes` string array can be used to allow certain ProcMountTypes.
Validation: `spec.containers[].securityContext.procMount` string value must occur in the policy.

30) AppArmor allowedProfileNames:
Configuration: AppArmor is in beta state and must therefore be configured in the annotations of the policy.

A list of allowed AppArmor profiles can be specified in `apparmor.security.beta.kubernetes.io/allowedProfileNames`. The values must be comma-separated.
Validation: the AppArmor profile string value is validated, specified in the annotation `container.apparmor.security.beta.kubernetes.io/[container_name]`. These operations might change once the AppArmor feature graduates to general availability (GA). [24]

31) AppArmor defaultProfileName:
Configuration: AppArmor is in beta state and must therefore be configured in the annotations of the policy. A default AppArmor profile can be specified in `apparmor.security.beta.kubernetes.io/defaultProfileName`.
Mutation: the AppArmor profile string value is added in the annotation `container.apparmor.security.beta.kubernetes.io/[container_name]` if no value is specified. These operations might change once the AppArmor feature graduates to general availability (GA). [24]

32) seccomp allowedProfileNames:
Configuration: while seccomp is general available (GA) since Kubernetes 1.19 [25], the policy configuration is still done via annotations. A list of allowed seccomp profiles can be defined in `seccomp.security.alpha.kubernetes.io/allowedProfileNames`. The values must be comma-separated.
Validation: when configured, the `spec.containers[].securityContext.seccompProfile` string value, `spec.securityContext.seccompProfile` string value and `seccomp.security.alpha.kubernetes.io/pod` annotation' string value must correspond with the allowed seccomp profile in the policy. These operations might change within a couple of releases, given that seccomp became GA in 1.19. [25]

33) seccomp defaultProfileName:
Configuration: while seccomp is general available (GA) since Kubernetes 1.19, the policy configuration is still done via annotations. A default seccomp profile can be specified in `seccomp.security.alpha.kubernetes.io/defaultProfileName`
Mutation: when configured, the seccomp annotation `seccomp.security.alpha.kubernetes.io/pod` and `spec.securityContext.seccompProfile` value will be added, with the string value that is specified in the policy. These operations might change within a couple of releases, given that seccomp became GA in 1.19. [25]

34) forbiddenSysctls:
Configuration: `forbiddenSysctls` string array can be used to block sysctls configurations. If a path ends with "$\star$", it will be considered a prefix.
Validation: when configured, the `spec.securityContext.sysctls` array will be validated. If a string in the field "name" corresponds to a string or prefix in the policy, the pod will be rejected.

35) allowedUnsafeSysctls:
Configuration: PSP will block all unsafe sysctls by default, even though they are enabled in kubelet. Therefore, the `allowedUnsafeSysctls` string array can be used to allow unsafe sysctl configurations. If a path ends with "$\star$", it will be considered a prefix.
Validation: when configured, the `spec.securityContext.sysctls` array will be validated. If an unsafe sysctl is specified, the pod will be allowed if the string in the field "name" corresponds to a string or prefix in the policy.

B. Phase 2 - Assessment of replacement capabilities

This subsection will present the findings of the second research phase, where we assessed the ability of alternative admission controllers to replicate PSP functions.

In Table I, an overview is given of the PSP functionality coverage in alternative admission controllers. Afterward, explanations for the results are given.

#	PSP Function Name	Gatekeeper	Kyverno	k-rail
1	privileged	✓	✓	✓
2	hostPID	✓	✓	✓
3	hostIPC	✓	✓	×
4	hostNetwork	✓	✓	✓
5	hostPorts	✓	✓*	×
6	volumes	✓	✓*	×
7	allowedHostPaths	✓	×	×
8	allowedFlexVolumes	✓	✓*	×
9	fsGroup MustRunAs	✓	✓*	×
10	fsGroup MayRunAs	✓	✓*	×
11	fsGroup RunAsAny	N/A	N/A	N/A
12	readOnlyRootFilesystem	✓	✓	×
13	runAsUser MustRunAs	✓	✓*	×
14	runAsUser RunAsAny	N/A	N/A	N/A
15	runAsUser MustRunAsNonRoot	≈*	✓*	×
16	runAsGroup MustRunAs	✓	✓*	×
17	runAsGroup MayRunAs	✓	✓*	×
18	runAsGroup RunAsAny	N/A	N/A	N/A
19	supplementalGroups MustRunAs	✓	✓*	×
20	supplementalGroups MayRunAs	✓	✓*	×
21	supplementalGroups RunAsAny	N/A	N/A	N/A
22	allowPrivilegeEscalation	✓	✓	×
23	defaultAllowPrivilegeEscalation	✓	✓*	×
24	allowedCapabilities	✓	✓*	×
25	requiredDropCapabilities	×	×	×
26	defaultAddCapabilities	×	×	×
27	seLinux MustRunAs	✓	✓*	×
28	seLinux RunAsAny	N/A	N/A	N/A
29	allowedProcMountTypes	✓	✓	×
30	AppArmor allowedProfileNames	✓	✓	×
31	AppArmor DefaultProfileName	✓	✓*	×
32	seccomp allowedProfileNames	✓	✓	✓
33	seccomp DefaultProfileName	✓	✓*	×
34	forbiddenSysctls	✓	✓*	×
35	allowedUnsafeSysctls	✓*	✓	×

TABLE I: Coverage of PSP requirements by alternative admission controllers
✓ = Feature or template available to replace PSP function
* = Template is inadequate or not available, but can be created
≈ = Imitates PSP feature partially
× = PSP feature cannot be imitated

Justification The functions are numbered and correspond with the row numbers of Table I.

1) privileged:
Gatekeeper: Verified sample template: *"privileged-container"* (validating). Template imitates PSP function.
Kyverno: Verified sample template: *"Disallow Privileged Containers"* (validating). Template imitates PSP function.
k-rail: Verified feature: *"pod-no-privileged-container"* (validating) Feature imitates PSP function.

2) hostPID:
Gatekeeper: Verified sample template: *"host-namespaces"* (validating). Template imitates PSP function.
Kyverno: Verified sample template: *"Disallow Host Namespaces"* (validating). Template imitates PSP function.
k-rail: Verified feature: *"pod-no-host-pid"* (validating). Feature imitates PSP function.

3) hostIPC:
Gatekeeper: Verified sample template: *"host-namespaces"* (validating). Template imitates PSP function.
Kyverno: Verified sample template: *"Disallow Host Namespaces"* (validating). Template imitates PSP function.
k-rail: Does not offer a replacement for the PSP `hostIPC` function.

4) hostNetwork:
Gatekeeper: Verified sample template: *"host-network-ports"* (validating). Template imitates PSP function.
Kyverno: Verified sample template: *"Disallow Host Namespaces"* (validating). Template imitates PSP function.
k-rail: Verified feature: *"pod-no-host-network""* (validating). Feature imitates PSP function.

5) hostPorts:
Gatekeeper: Verified sample template: *"host-network-ports"* (validating). This constraint template (Gatekeeper terminology

for policy template [13]) is almost complete, but does not support multiple integer ranges. Extend the template and use the following REGO logic to match multiple ranges: {1 | val >= ranges[j].min; val <= ranges[j].max}

Kyverno: A feature blocking template is provided by Kyverno: (*"Disallow Host Ports"*) (validating). The template does not meet the requirements. To replicate the PSP function, a template can be created that only allows use of whitelisted host ports, using the =() operator. Multiple allowed host port ranges can be allowed by using the following operators: "<,>,&,|"

k-rail: Does not offer a replacement for the PSP hostPorts function.

6) volumes:

Gatekeeper: Verified sample template: *"volumes"* (validating). Template imitates PSP function.

Kyverno: A feature block template is provided by Kyverno: (*"Restrict Volume Types"*) (validating). But, to imitate the PSP function, a validating template can be created that only allows use of whitelisted volumes using the =() operator. Multiple strings can be defined by a delimiting "|"

k-rail: Does not offer a replacement for the PSP volumes function, but can block persistent host path volumes using *"persistent-volume-no-host-path"*.

7) allowedHostPaths:

Gatekeeper: Verified sample template: *"host-filesystem"* (validating). Template imitates PSP function.

Kyverno: Verified sample template: *"Disallow Host Path"*. Unfortunately, it misses the conditional filtering on read-only volumeMounts that are connected to volumes with a specific host path. A policy can be created without this read-only feature using the =() operator, but this effectively allows attackers to traverse the file system outside of the allowed host paths. [5]

k-rail: Does not offer a replacement for the PSP allowedHostPaths function.

8) allowedFlexVolumes:

Gatekeeper: Verified sample template: *"flexvolume-drivers"* (validating). Template imitates PSP function.

Kyverno: No template available. A validating template can be created that only allows use of whitelisted flexvolume drivers using the =() operator. Multiple strings can be defined by a delimiting "|"

k-rail Does not offer a replacement for the PSP allowedFlexVolumes function.

9) fsGroup MustRunAs:

Gatekeeper: Verified sample templates: *"users"* (validating & mutating). Templates imitate PSP function when combined.

Kyverno: Verified sample template: *"Validate User ID, Group ID, and FS Group"* (validating). The templates do not meet the requirements. A mutating and validating template can be created. To enforce a default fsGroup value, use a mutation rule to add the fsGroup tag if not present, using the +() operator. Multiple allowed fsGroup ranges can be whitelisted by using the following operators: "<,>,&,|"

k-rail: Does not offer a replacement for the PSP fsGroup function.

10) fsGroup MayRunAs:

Gatekeeper: Verified sample template: *"users"*. Template imitates PSP function.

Kyverno: Verified sample template: *"Validate User ID, Group ID, and FS Group"* (validating). The template does not meet the requirements. To imitate the PSP function, a validating template can be created that only allows use of whitelisted integers using the =() operator. Multiple allowed fsGroup ranges can be whitelisted by using the following operators: "<,>,&,|"

k-rail: Does not offer a replacement for the PSP fsGroup function.

11) fsGroup RunAsAny: N/A

12) readOnlyRootFilesystem:

Gatekeeper: Verified sample templates: *"read-only-root-filesystem"* (validating & mutating). Templates imitate PSP function when combined.

Kyverno: No template available. Template to replace the PSP feature can be created with the following properties:

- Mutation rule that uses the () operator to inspect if the readOnlyRootFilesystem value is absent (null). The readOnlyRootFilesystem can be added using the +() operator. This ensures the value is only appended to the pod when the field is absent.
- Validation rule with pattern which sets the readOnlyRootFilesystem to true.

k-rail: Does not offer a replacement for the PSP readOnlyRootFilesystem function.

13) runAsUser MustRunAs:

Gatekeeper: Verified sample templates: *"users"* (validating & mutating). Templates imitate PSP function when combined.

Kyverno: No template available. A mutating and validating template can be created. To enforce a default user identifier, add (mutate) the tag if not present, using the +() operator. The value must be added to runAsUser in the container securityContext.(name): "*" can be used to target all containers of the manifest.

The validating phase of this function also needs to be done in both contexts. Multiple allowed user identifier ranges can be whitelisted by using the following operators: "<,>,&,|"

k-rail: Does not offer a replacement for the PSP runAsUser function.

14) runAsUser RunAsAny: N/A

15) runAsUser MustRunAsNonRoot:
Gatekeeper: Verified sample templates: *"users"* (validating & mutating). Does not exactly imitate PSP function: no conditional logic is available to apply mutations only when neither runAsNonRoot, nor runAsUser are specified. However, the validation template in combination with the mutation template (which appends the runAsNonRoot value) will replicate the PSP behavior, except for one aspect: the pod manifest will look different for requests that did initially only specify runAsUser.
Kyverno: No template available. A mutating and validating template can be created for both the pod- and container context. By using the `()` operator on runAsUser `null` and runAsNonRoot `null` and the `+()` operator for runAsNonRoot in one mutation rule, Kyverno will only add runAsNonRoot if both values are absent. A validation rule can futher verify if the runAsUser and runAsNonRoot values are set to `0>` and `true`, respectively.
k-rail: Does not offer a replacement for the PSP `runAsUser` function.

16) runAsGroup MustRunAs:
Gatekeeper: Verified sample templates: *"users"* (validating mutating). Template imitates PSP function.
Kyverno: Verified sample template: *"Validate User ID, Group ID, and FS Group"* (validating). Does not imitate the PSP function. The template must meet the same requirements as the `fsGroup MustRunAs` function, except that both the PodSpec and the container specification must be validated using multiple validation rules and the runAsGroup value must be mutated in the container specification to replace PSP' behavior. Therefore, the `()` operator can be used to ensure the mutation is applied when the runAsGroup value is absent (`null`) in both the PodSpec as well as in the container specification.
k-rail: Does not offer a replacement for the PSP `runAsGroup` function.

17) runAsGroup MayRunAs:
Gatekeeper: Verified sample template: *"users"* (validating). Template imitates PSP function.
Kyverno: Verified sample template: *"Validate User ID, Group ID, and FS Group"* (validating). Does not imitate the PSP function. The template must meet the same requirements as the `fsGroup MayRunAs` function, except that both the PodSpec and the container specification must be validated using multiple validation rules.
k-rail: Does not offer a replacement for the PSP `runAsGroup` function.

18) runAsGroup RunAsAny: N/A

19) supplementalGroups MustRunAs:
Gatekeeper: Verified sample templates: *"users"* (validating & mutating). Template imitates PSP function.
Kyverno: No template available. The template must meet the same requirements as the `fsGroup MustRunAs` function.
k-rail: Does not offer a replacement for the PSP `supplementalGroups` function.

20) supplementalGroups MayRunAs:
Gatekeeper: Verified sample template: *"users"* (validating). Template imitates PSP function.
Kyverno: No template available. The template must meet the same requirements as the `fsGroup MayRunAs` function.
k-rail: Does not offer a replacement for the PSP `supplementalGroups` function.

21) supplementalGroups RunAsAny: N/A

22) allowPrivilegeEscalation :
Gatekeeper: Verified sample template: *"allow-privilege-escalation"* (validating). Template imitates PSP function.
Kyverno: Verified sample template: *"Deny Privilege Escalation"* (validating). Template imitates PSP function.
k-rail: Does not offer a replacement for the PSP `allowPrivilegeEscalation` function.

23) defaultAllowPrivilegeEscalation:
Gatekeeper: Verified sample templates: *"allow-privilege-escalation"* (mutating). Template imitates PSP function.
Kyverno: No template available. A mutating template can be created, that mutates Privilege Escalation to a certain value when none is specified. Therefore, use the `()` operator to apply the value if none is present (`null`). Validation rules must be absent. This policy may be combined with `allowPrivilegeEscalation`
k-rail: Does not offer a replacement for the PSP `defaultAllowPrivilegeEscalation` function.

24) allowedCapabilities:
Gatekeeper: Verified sample template: *"capabilities"* (validating). Template imitates PSP function.
Kyverno: A block feature template is provided by Kyverno: (*"Disallow Add Capabilities"*, validating). Does not imitate PSP function. The template must meet the same whitelisting requirements as the `fsGroup MustRunAs` function.
k-rail: Verified feature: *"pod-no-new-capabilities"* (validating). This is not a complete replacement for the PSP `allowedCapabilities` function, but can reject pods with additional capabilities.

25) requiredDropCapabilities:
Gatekeeper: Verified templates: *"capabilities"* (validating & mutating). Template imitates PSP function.
Mutation module provides no way to strip values from the request. This results in having the same value in the `add` and `drop` fields.
Kyverno: Verified templates: *"Drop All Capabilities"* and *"Disallow add capabilities"* (validating). These templates do not meet the requirements. `requiredDropCapabilities` cannot be implemented

correctly in Kyverno as the "!" operator matches in a case-sensitive manner. The consequence is that dropping capabilities is non-effective, as an attacker is able to substitute a lowercase string character for an uppercase character. In practice, the kube-apiserver and kubelet will process and add the capabilities that are present in both the "add" and "drop" arrays, which makes this replacement vulnerable for attacks.
k-rail: Does not offer a replacement for the PSP `requiredDropCapabilities` function.

26) defaultAddCapabilities:
Gatekeeper: Verified sample templates: *"capabilities"* (validating & mutating). Non-working template. Value will be overwritten instead of appended.
Kyverno: No template available. No working template can be created, since it is not possible to first verify if a capability is specified in the "drop" array before it is appended to the "add" array. This is because the `!` operator will match strings in a case-sensitive manner.
k-rail: Does not offer a replacement for the PSP `defaultAddCapabilities` function.

27) seLinux MustRunAs:
Gatekeeper: Verified sample templates: *"selinux"* (validating & mutating). Template imitates PSP function.
Kyverno: No template available. A mutating and validating template can be created. To enforce a default set of selinux options, add the `seLinuxOptions` tag if not present (`null`) with a mutation rule and the `+()` operator. Afterward, validate if default values are added in both the pod- and container context.
k-rail: Does not offer a replacement for the PSP `seLinux` function.

28) seLinux RunAsAny: N/A

29) allowedProcMountTypes:
Gatekeeper: Verified sample template: *"proc-mount"* (validating). Template imitates PSP function.
Kyverno: Verified sample template: *"Require Default Proc Mount"* (validating). Template imitates PSP function.
k-rail: Does not offer a replacement for the PSP `allowedProcMountTypes` function.

30) AppArmor allowedProfileNames:
Gatekeeper: Verified sample template: *"apparmor"* (validating). Template imitates PSP function.
Kyverno: Verified sample template: *"Restrict AppArmor"* (validating). Template imitates PSP function.
k-rail: Verified feature: *"pod-deny-unconfined-apparmor-policy"* (validating). This feature is not a complete replacement for the PSP `allowedProcMountTypes` function, but can deny the "unconfined" AppArmor profile.

31) AppArmor defaultProfileName:
Gatekeeper: Verified sample template: *"apparmor"* (mutating). Template imitates PSP function.
Kyverno: No template available. A mutating policy can be created which adds "`container.apparmor.security.beta.kubernetes.io/pod`" and the AppArmor profile as a key-value pair to the `metadata.annotations` field if not present.
k-rail: Does not offer a replacement for the PSP `AppArmor defaultProfileName` function.

32) seccomp allowedProfileNames:
Gatekeeper: Verified sample template: *"seccomp"* (validating). This template works by verifying the annotations. seccomp did become general available (GA) since Kubernetes 1.19 [25], and therefore an updated template should be created in the future, when the annotation support is removed.
Kyverno: Verified sample template: *"Restrict Seccomp"* (validating). Template imitates PSP function.
k-rail: Does not offer a replacement for the PSP `seccomp allowedProfileNames` function.

33) seccomp defaultProfileName:
Gatekeeper: Verified sample template: *"seccomp"* (mutating). This template works by verifying the annotations (AssignMetadata). To update this template for future Kubernetes versions without seccomp annotation support, an Assign template can be created with a pathTest.
Kyverno: Verified sample template: *"Restrict Seccomp"* (validating). This template does not replace the PSP function, as it is validating. A mutating policy can be created, that adds the seccomp profile type when not specified using the `=()` operator. Policy can be combined with `seccomp allowedProfileNames`.
k-rail: Verified feature: *"pod-default-seccomp-policy"*. Feature imitates PSP function.

34) forbiddenSysctls:
Gatekeeper: Verified sample template: *"forbidden-sysctls"* (validating). Template imitates PSP function.
Kyverno: Sample whitelist template*"Restrict Sysctls"* (validating) is available. Because this PSP function requires a blacklist function, a template should be created, that mimics the blacklisting behavior of the `forbiddenSysctls` PSP function using the `!` operator. Use the `^()` operator to match a list of forbidden sysctls. When at least one of the forbidden sysctls is specified, the pod creation request will be blocked.
k-rail: Does not offer a replacement for the PSP `forbiddenSysctls` function.

35) allowedUnsafeSysctls:
Gatekeeper: Verified sample template: *"forbidden-sysctls"* (validating). This whitelist template can be refactored to a blacklist template by inverting the `forbidden-sysctl` REGO function with the negation function `not`. This is also done in (e.g.) the procmount validation template. Also, the safe sysctls must be added to the `forbiddenSysctls` comparison.

Kyverno: Verified sample templates: *"Restrict Sysctls"* (validating). This whitelist tempalte is offered by Kyverno, but the safe sysctls must be added to it to replicate the PSP behavior. A list of safe sysctls can be retrieved from the Kubernetes sysctl documentation. [26]
k-rail: Does not offer a replacement for the PSP `allowedUnsafeSysctls` function.

36) Other results: In the previous section, we found that the RunAsAny functions are not applicable to alternative admission controllers. The reason for this is that PSP blocks some pod functionality by default. Only if these features is configured explicitly, the parameters can be set in the pod manifest. RunAsAny functions basically disable the validation of the related fields and values, which the other controllers will do by default if no policy is configured for those features.

Another noteworthy observation is that Gatekeeper, Kyverno, and k-rail validate and mutate not only pod objects, but also Deployments, StatefulSets and DaemonSets. However, the "JSONPatch" and "patch strategic merge" mutation methods in Kyverno do not automatically support other resource types than pods. For Gatekeeper the REGO constraint templates must be extended: the template should iterate through the `input.review.object.spec.template.spec` fields and the constraints should match the Deployment, StatefulSet or DaemonSet kinds.

V. Discussion

In this section, we discuss the results described in the previous section.

A. Missing details in PSP confirmed

The results of our first research phase confirmed our assumption about missing details in the API reference and PSP documentation. The following PSP characteristics are missing:

- The documentation about "MustRunAs" function rules does not describe if the pod specification (PodSpec) or the container specification is mutated. Based on our experiments, we conclude that the container specification is mutated, when the PodSpec or container specification does not contain any configuration regarding the related feature.
- The documentation about `readOnlyRootFilesystem` does not describe if pods are mutated or only validated. Based on our experiments, we conclude that the feature does also mutate.
- The documentation does not provide any information about the validation and mutation of replaced annotations of beta functions that did become GA in the past or will become in the future. Based on our experiments, we conclude that the annotations of the feature that recently became GA (seccomp) are also validated and mutated by PSP.

This information is crucial to be able to formulate the requirements for alternative admission controllers.

B. Template-based versus built-in set of features

Another remarkable result is the amount of replaceable PSP functions in Gatekeeper and Kyverno. The template-based policy system gives the administrator a lot of flexibility when sample templates are missing, since we could create a lot of working templates ourselves that meet the requirements of the PSP function. Let us discuss the details.

C. Gatekeeper

Gatekeeper offers a lot of templates in their GitHub library [20] to replace PSP features. When we look into the templates and try them, we find that all PSP validation functionality can be imitated, but the mutation feature is still in the alpha development phase and is not able to imitate all PSP mutation functions.

D. Kyverno

Kyverno also offers templates on their Kyverno Policies website to replace PSP features [21]. However, a lot of templates are missing and some templates do not meet the requirements of the PSP function. Fortunately, for a large majority of these cases, a template can be created. For the remainder of the features, the cluster administrator can use a template that blocks the feature completely.

A noteworthy detail is that Kyverno will remove the "overlay" function to replace it with the "patch strategic merge" or "JSONPatch" methods in version 1.5 [27]. However, it is not known whether the replacement methods will cover all overlay functionality. During our reseach, we missed the following functionality:

- JSONPatch does not support a wildcard operator to mutate all items in an array. This is necessary to mutate container specifications
- PatchStrategicMerge does not support mutation when no value is provided (setting default values), as this method overwrites them
- Other types of resources (i.e. Deployments, StatefulSets and DaemonSets) must be mutated separately, while overlay covers them automatically

E. k-rail

k-rail does not offer a lot of features that are able to imitate PSP functionality. The large majority of k-rail features that do replicate PSP functionality only validate boolean values. However, it is possible to program additional policy features for k-rail, although this is out of the scope of this research.

Initially, Table I might give the false impression that k-rail is only able to limit 4 pod features. However, this is not the case, since k-rail offers features to block pod functionality, as we see in section IV-B6, IV-B24 and IV-B30. Nevertheless, these k-rail features do not meet the requirements to replicate the PSP functionality completely.

In retrospect, k-rail might not have been a suitable alternative for the PSP functionality comparison.

F. Significance of validating and mutating functions

The results show that Gatekeeper and Kyverno cover most validating functions. When it comes to mutation support of the admission controllers overall, we see that this is incomplete and experimental. However, we argue that the lack of mutation functions should not be a significant security concern, since being able to validate and block vulnerable configurations actually minimizes the attack surface. Mutation is used to set default values, which can be convenient, but validation is needed to protect a cluster. Therefore, to minimize the attack surface of a cluster, the validating functions need to be considered more important than mutating functions. However, mutational policies serve a good purpose. They can accelerate automation, as they can be a tool in CI/CD development pipelines to set a security baseline for all sorts of Kubernetes deployments.

G. PSP limitations

As mentioned in the introduction section, PSP is getting deprecated because it is considered to be confusing. For instance, the authorization model is complex [3]. Therefore, it might be interesting to discuss our observations regarding the complexity of the alternative admission controllers.

During our research, we noticed that the REGO policy language for Gatekeeper constraint templates is versatile, which allows for highly complex conditional policies. However, we consider the REGO language to be quite complex. Fortunately, a lot of sample templates are available [20], but when the cluster administrator wants to create a new constraint template, he might need to face a steep learning curve.

Kyverno does not confront the cluster administrator with a complex template language. On the other hand, creating policies for more complex validation conditions might not be possible, as we saw in section IV-B7, IV-B25 and IV-B26.

k-rail can be configured easily in the values.yaml file before deployment of the controller.

Another point that made PSP confusing is the lack of an audit mode [3]. Fortunately, Gatekeeper (dry-run), Kyverno (audit mode) and k-rail (passive report-only mode) all offer such a feature.

The last point that made PSP confusing is the prioritization of policies [3], which is done alphabetically. Unfortunately, we did not research the prioritization of policies in the other admission controllers, since this was out of the research scope.

H. Additional functionality

As the name would imply, Pod Security Policy only validates or mutates pod configurations. But, security in a Kubernetes cluster goes beyond pod configurations. During our research, we saw that the alternative admission controllers offer a lot of features outside of the pod context that might improve security. A cluster administrator might want to enable some of these policies to further minimize the attack surface.

I. Role-based access control

We are aware that being able to control what policy users, groups or service accounts will use (role-based access control) might be crucial for cluster administrators. Therefore, our results give only partial insight in the ability of alternative admission controllers to replace PSP.

VI. Conclusion

We have obtained comprehensive results showing the ability of the alternative admission controllers to replace PSP functionality. In conclusion, Gatekeeper and Kyverno are able to cover the large majority of PSP functions. Our work has also led to the conclusion that k-rail is not able to cover a lot of PSP functions.

Based on literature research, we conclude that some details are missing in the PSP documentation. This creates room for interpretation. We were able to fill in the gaps with technical experiments to create a complete list of requirements for alternative admission controllers.

The results of this study indicate that the experimental mutation feature of Gatekeeper does not offer as much versatility as the validation feature does. Therefore, Gatekeeper is not able to implement all the mutation functionality of PSP, although this should not be a security concern. Kyverno, on the other hand, misses some validation capabilities, which might lead to gaps in the attack surface of a cluster. k-rail does offer a very limited amount of PSP validation and mutation features.

Our work has demonstrated that all studied admission controllers have their disadvantages. Gatekeeper uses the complex REGO language to define constraint templates, while Kyverno misses a lot of templates. Taken together, these findings implicate that Gatekeeper might be the most powerful tool to minimize the attack surface on a Kubernetes cluster, whereas Kyverno covers most of the PSP features while also offering readability of policy configurations.

Finally, a potential limitation needs to be considered. Our research only studied the validation and mutation functions of PSP. Therefore, our study gives only partial insight into the ability of admission controllers to replicate PSP functionality. For this reason, this study is the first major step to compare the abilities of PSP with other admission controllers.

We hope that our study will be useful when considering a replacement for PSP or an admission controller in general.

VII. Future Work

We are currently in the process of publishing a GitHub repository that includes the missing templates for Gatekeeper and Kyverno. This might accelerate transitions from PSP to Gatekeeper or Kyverno.

The following topics are reserved for future work:

Firstly, to create a complete picture of the ability of alternative admission controllers to replace PSP, future work on the prioritization of policies and role-based access control (RBAC) possibilities needs to be carried out, as our study mainly focused on the validation and mutation features. We know that mutating policies are applied before the validating

policies are processed [7], but it is relevant to know in what manner the admission controllers apply multiple policies. The controller might merge the policies to apply the union of them, or the controller might apply them in a certain order. Additionally, an investigation of the differences of the role-based RBAC possibilities in PSP and the RBAC possibilities in other admission controllers would be an important research topic, since cluster administrators might want to limit the privileges of certain users or service accounts.

Secondly, as mentioned in the Introduction (I), a simplified replacement for PodSecurityPolicy (KEP 2579) is currently being developed on the initiative of the Kubernetes community[3]. An initial alpha implementation is planned for Kubernetes 1.22 [3]. As soon as this module is in a more mature development phase, additional research should be done to compare its functionality with PSP and alternative admission controllers.

Lastly, it might be interesting to study to what extent admission controllers can mitigate other vulnerable configurations outside of the pod context, to minimize the attack surface.

References

[1] Kubernetes, *Release release 1.0.0*, Jun. 2015. [Online]. Available: https://github.com/kubernetes/kubernetes/releases/tag/v1.0.0.

[2] L. Artem, B. Tetiana, M. Larysa, and V. Vira, "Eliminating privilage escalation to root in containers running on kubernetes," *Scientific and practical cyber security journal*, 2020.

[3] T. Sable, *Podsecuritypolicy deprecation: Past, present, and future*, Apr. 2021. [Online]. Available: https://kubernetes.io/blog/2021/04/06/podsecuritypolicy-deprecation-past-present-and-future/.

[4] Kubernetes, *Kubernetes 1.4: Making it easy to run on kubernetes anywhere*, Sep. 2016. [Online]. Available: https://kubernetes.io/blog/2016/09/kubernetes-1-4-making-it-easy-to-run-on-kuberentes-anywhere/.

[5] ——, *Pod security policies*, Feb. 2021. [Online]. Available: https://kubernetes.io/docs/concepts/policy/pod-security-policy/.

[6] *A guide to kubernetes admission controllers*, Mar. 2019. [Online]. Available: https://kubernetes.io/blog/2019/03/21/a-guide-to-kubernetes-admission-controllers/.

[7] ——, *Using admission controllers*, Jun. 2021. [Online]. Available: https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/.

[8] *Moving forward from beta*, Sep. 2020. [Online]. Available: https://kubernetes.io/blog/2020/08/21/moving-forward-from-beta/.

[9] k-rail, *Open-policy-agent/gatekeeper: Github stargazers*, Jun. 2021. [Online]. Available: https://github.com/open-policy-agent/gatekeeper/stargazers.

[10] C. N. C. Foundation, "Cloud native computing foundation," *CNCF*, Feb. 2021.

[11] Gatekeeper, *Introduction*. [Online]. Available: https://open-policy-agent.github.io/gatekeeper/website/docs.

[12] ——, *Gatekeeper - mutation*. [Online]. Available: https://open-policy-agent.github.io/gatekeeper/website/docs/mutation.

[13] ——, *How to use gatekeeper*. [Online]. Available: https://open-policy-agent.github.io/gatekeeper/website/docs/howto.

[14] k-rail, *Kyverno/kyverno - github stargazers*, Jun. 2021. [Online]. Available: https://github.com/kyverno/kyverno/stargazers.

[15] Kyverno, *Kyverno/kyverno - kubernetes native policy management*. [Online]. Available: https://github.com/kyverno/kyverno.

[16] ——, *Kyverno - validating resources*, Jun. 2021. [Online]. Available: https://kyverno.io/docs/writing-policies/validate/.

[17] k-rail, *Cruise-automation/k-rail: Github stargazers*, Jun. 2021. [Online]. Available: https://github.com/cruise-automation/k-rail/stargazers.

[18] ——, *Cruise-automation/k-rail: Kubernetes security tool for policy enforcement*, Jun. 2021. [Online]. Available: https://github.com/cruise-automation/k-rail.

[19] Kubernetes, *Kubernetes api reference v1.20*. [Online]. Available: https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.20/.

[20] Gatekeeper, *Open-policy-agent/gatekeeper-library: The opa gatekeeper policy library*. [Online]. Available: https://github.com/open-policy-agent/gatekeeper-library.

[21] Kyverno, *Policies*. [Online]. Available: https://kyverno.io/policies/.

[22] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun, and Q. Zhou, "A measurement study on linux container security: Attacks and countermeasures," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 418–429.

[23] M. S. I. Shamim, F. A. Bhuiyan, and A. Rahman, "Xi commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices," in *2020 IEEE Secure Development (SecDev)*, IEEE, 2020, pp. 58–64.

[24] Kubernetes, *Restrict a container's access to resources with apparmor*, Mar. 2021. [Online]. Available: https://kubernetes.io/docs/tutorials/clusters/apparmor/.

[25] ——, *Kubernetes change log 1.19*, Jun. 2021. [Online]. Available: https://github.com/kubernetes/kubernetes/blob/master/CHANGELOG/CHANGELOG-1.19.md.

[26] ——, *Using sysctls in a kubernetes cluster*, Apr. 2021. [Online]. Available: https://kubernetes.io/docs/tasks/administer-cluster/sysctl-cluster/.

[27] Kyverno, *Kyverno - mutating resources*, Jun. 2021. [Online]. Available: https://kyverno.io/docs/writing-policies/mutate/.