AWS Whitepaper

Security Practices for Multi-Tenant SaaS Applications using Amazon EKS

Security Practices for Multi-Tenant SaaS Applications using Amazon EKS: AWS Whitepaper

Table of Contents

Security Practices for Multi-Tenant SaaS Applications using Amazon EKS

Publication date: **June 4, 2021** (*Document history*)

This guide shows you how to securely manage and operate multi-tenant software-as-a-service (SaaS) applications on Amazon Elastic Kubernetes Service (Amazon EKS) clusters.

This document was adapted from the *Amazon EKS Best Practices Guide*. The *Best Practices Guide* is updated frequently. AWS recommends checking for updates periodically, because Amazon EKS and Kubernetes are rapidly evolving. AWS also recommends subscribing to the AWS Containers Blog to receive the latest updates on AWS container services.

Overview

Amazon Elastic Kubernetes Service (Amazon EKS) is frequently used by customers who are building software-as-a-service (SaaS) solutions on AWS. How tenant data and applications are isolated in these SaaS solutions can vary. Some SaaS providers rely on a *siloed* tenancy model where each tenant has its own resources. Others rely on a *pooled* tenancy model where resources are shared by tenants.

The following provides a more detailed overview of how these two models are realized on Amazon EKS:

- The **Pool Model** describes an environment where the EKS resources are shared by tenants with added measures to ensure that any one tenant cannot access the resources of another tenant. Many customers want to run workloads using shared hosts and a common control plane. This approach typically simplifies the operational footprint of a SaaS application and improves the agility, innovation, and cost model of a SaaS environment.
- The **Silo Model** represents a model where each tenant has dedicated EKS resources. This model is often a good fit for tenants that may demand a more absolute isolation boundary. This may be for a variety of reasons (security, noisy neighbors, and so on). There are multiple constructs available in EKS that can be used to realize the Silo model. The resources accessed from a silo could be deployed in a *silo* or *pool* model.

These choices are not exclusive. Some SaaS providers may support both options depending on the tiers or services that are part of their application.

For both of these models, it's important ensure that tenants are unable to:

- Read or write any control-plane information unrelated to the tenant.
- Access any resources not belonging to the tenant.
- Obtain credentials not belonging to the tenant.
- Impersonate other tenants.
- Escape the confines of the tenant's allocated compute, memory, or other resources.

Recommendations

AWS recommendations focus on the following objectives:

- Keeping control plane data strictly separated among tenants.
- Preventing host corruption by tenant containers.
- Preventing tenant containers from *breaking out of jail* and accessing sensitive data on the hosts, such as credentials.

Use multiple clusters to separate tenant workloads

The most secure way to run Silo workloads on EKS is to create a distinct EKS cluster for each tenant. In such a design, even a tenant that runs privileged containers and has access to the hosts cannot impact other tenants. Care must still be taken to not provide credentials related to other tenants on a different cluster, and other AWS security best practices such as proper Security Group rules, virtual private cloud (VPC) separation, or both must be implemented.

This approach does have some disadvantages. Having a separate cluster for each tenant will add more complexity to the operational footprint of your environment. While you can automate much of the operational experience, this approach will impact the efficiency, agility, and cost profile of your SaaS environment.

Use tenant-dedicated Worker nodes

Customers choosing to host multiple tenants on a single cluster should sequester tenant workloads onto dedicated nodes. This will help to ensure that, in the event of a container breakout, no other customer's Pods or data can be observed or tampered with.

AWS Fargate

The easiest way to enforce this constraint is to run tenant Pods on AWS Fargate. Fargate is a managed compute service that can run EKS Pods without having to manage Amazon Elastic Compute Cloud (Amazon EC2) instances. When a Pod is scheduled, capacity is allocated on-demand that is custom fit to match the Pod's resources. With Fargate, no two Pods are run on the same virtual machine (VM), ensuring VM-level isolation as well as container isolation for tenant workloads.

Amazon EC2 Worker nodes

Alternatively, if EC2 instances are used, one way to enforce this constraint is to apply taints to all nodes with a tenant identifier. An example taint might be `tenantID=12345:NoSchedule`. When combined with a matching toleration in a tenant's Pod specification, this will ensure that the tenant's Pods can be placed only on nodes matching the same taint.

Another (and somewhat weaker) way to enforce the constraint is to label nodes with tenant identifiers, and using `nodeSelector` or `affinity` rules in Pod specifications to ensure tenant Pods are scheduled only on the correct nodes. Customers who decide to implement the constraint this way should use admission controllers, discussed in Use admission controllers to enforce security policies, to ensure those fields are supplied for all customer Pods.

Node Authorization Mode

As a mitigating control, in EKS clusters, all requests from nodes are subject to the Node Authorization Mode. This prevents nodes from accessing Secrets, ConfigMaps, Persistent Volume Claims, or Persistent Volumes unless they are related to pods running on the node itself. See Using Node Authorization for additional information.

Do not provide direct access to Kubernetes or EKS APIs

Accepting untrusted input from tenants and passing it to a security-sensitive system such as Kubernetes may expose your cluster or its tenants to risks, such as unauthorized modifications and data access. AWS recommends placing a discrete management layer between tenants and the EKS clusters on which their workloads run. Similar to a Web Application Firewall (WAF), this layer allows requests to be examined and filtered before taking further action. Invalid requests should be rejected immediately, while valid requests should be decorated with identifying information and security-related modifications before being passed to the Kubernetes control plane.

Use Namespaces to separate tenant workloads

Kubernetes uses namespaces as a logical partitioning system for organizing objects such as Pods and Deployments. Namespaces also operate as a privilege boundary in Kubernetes' Role-Based Access Control (RBAC) system. For example, Pods created in `namespace A` do not have access to secrets in `namespace B` (and vice-versa).

AWS advises customers to assign each tenant to their own unique namespace. When assigning privileges to tenants, ensure each tenant can only access Kubernetes objects in the tenant's assigned namespace. Customers can automate this assignment by enabling a mutating admission webhook that requires a tenant-specific label on all customer-related objects and ensures the objects are placed in the tenant's namespace.

Restrict container privileges

Tenant containers should run **unprivileged** by default. If a tenant's container requires privileges, those privileges should be limited only to those required to successfully run the container.

Privileges are specified in a container's `SecurityContext`. Privileges can be specified in one of two ways:

- By setting the `privileged` attribute to `true`. This is practically identical to having root access on the host.
- By specifying a list of one or more capabilities to `add` or `drop` in the `capabilities` list.

On EKS nodes that run the Docker container runtime, which includes those that use the EKS Optimized Amazon Machine Image (AMI), each container has the following default capabilities:

```
CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_FOWNER, CAP_FSETID, CAP_KILL,
CAP_SETGID, CAP_SETUID, CAP_SETPCAP, CAP_NET_BIND_SERVICE, CAP_NET_RAW,
CAP_SYS_CHROOT, CAP_MKNOD, CAP_AUDIT_WRITE, CAP_SETFCAP
```

AWS recommends dropping all unnecessary capabilities from this list, because most software does not need them. AWS recommends examining the full list of Linux capabilities, and allowing each tenant to select only those capabilities you permit. Capabilities are documented on the Capabilities Linux manual page.

AWS also recommends disallowing any containers from running with the privileged attribute set to true. It is much safer to provide fine-grained privileges by granting specific capabilities instead. For example, a container that needs to bind to a lownumbered port can be run with the `CAP_NET_BIND_SERVICE` capability instead of running with full privileges. AWS also recommends disallowing the `CAP_SYS_ADMIN` and `CAP_NET_ADMIN` capabilities, because they allow near-privileged access to the host.

Admission controllers, discussed in Use admission controllers to enforce security policies, can help enforce these restrictions.

Forbid running tenant containers as root

To simplify administration, Kubernetes containers share a user-ID namespace with the host by default. This means that UID `100` inside the container is identical to UID `100` on the host. The same is true for UID `0` (for example, the root user).

By default, containers start running as UID `0` (root). This poses several risks. For example, if an unauthorized user compromises the application, they could read and write files inside the container's filesystem or gain remote access to it. If any host filesystems are mounted into the container, the attacker could read and write any files within them. Finally, if the container is run in privileged mode, the attacker unauthorized user could obtain host-level access. This could compromise not only the host itself, but also the control plane.

AWS recommends that each `Dockerfile` used to build a tenant's container image specify a USER directive that is a non-root user name or ID. In addition, AWS recommends each tenant's container be run with a specific user ID, group ID, and `fsGroup` (equal to the group ID) in the `SecurityContext` of a Kubernetes container specification.

> **Note**
>
> Pods that need to access Secrets or utilize IAM roles for service accounts, and that are not running as root, must specify an `fsGroup` in their `securityContext` that matches the group ID. This will prevent permission errors related to file ownership.

Admission controllers, discussed in Use admission controllers to enforce security policies, can help enforce these restrictions.

Restrict mounting host filesystems

Containers have the ability to mount volumes from the host into them. This is a useful feature in some circumstances, but poses significant risks.

First, containers might be able to view Secrets from the host or other containers. For example, if `/var/lib` is mounted from the host into the container, files in other containers—including Secrets—would be visible as well.

Containers that run as root will have unrestricted write access to the host file system. This could allow an unauthorized user to modify kubelet settings, create symbolic links to directories or files in another sensitive location (such as `/etc/shadow`), install Secure Shell (SSH) keys, corrupt essential files, or perform other malicious activities.

AWS recommends restricting containers from mounting host filesystems unless strictly necessary. It is rare for a container in a Software as a Service (SaaS) environment to need access to the host. Where it is required, AWS recommends enforcing read-only mounts so that files cannot be written on the host.

Admission controllers, discussed in Use admission controllers to enforce security policies, can help enforce these restrictions.

Restrict the use of host networking and block access to instance metadata service

The EC2 Instance Metadata Service (IMDS) is accessible to all EC2 instances by default. This service provides useful introspection facilities, such as determining a node's availability zone, instance ID, and so forth. In addition, IMDS provides access to IAM credentials that allow applications to assume the instance's IAM role.

By default, every EC2 node in an EKS cluster is provided certain privileges necessary to bootstrap itself and assign IP addresses to pods. For example, a node can attach a VPC network interface and discover information about the EKS cluster it attaches to. While these privileges are required for the node to operate effectively, it is not usually desirable that the pods running on the node inherit these privileges.

One way to block pod IMDS access is to apply a network policy, enforced by the Amazon VPC CNI or an add-on such as Calico, to ensure pods are unable to reach the Instance Metadata Service. To do this, configure your network policy to block egress traffic to `169.254.0.0/16`.

Another way to block pod IMDS access is to require IMDS version 2 (`IMDSv2`) to be used, and to set the maximum hop count to 1. Configuring IMDS this way will cause requests to IMDS from pods to be rejected, provided those pods do not use host networking.

Additionally, AWS recommends forbidding untrusted pods from using host networking.

Admission controllers, discussed in Use admission controllers to enforce security policies, can help enforce this prohibition.

Restrict creation of services with external IP addresses

A core feature of Kubernetes is a Service abstraction. Service abstractions work in part by creating Domain Name System (DNS) entries in the cluster, visible by all pods, that point to IP addresses. These IP addresses might be pods, or they might be external addresses.

Additionally, any Kubernetes Service that has an external IP address will cause all traffic to that address from any of the pods in the cluster to be sent to that service— even that IP address actually belongs to a third party.

To illustrate, consider a hypothetical service on the internet with an address of `1.2.3.4`. If a tenant creates a Kubernetes service with an external IP address of `1.2.3.4`, *all* traffic destined for `1.2.3.4` from inside the cluster will be intercepted by that service. This poses a significant security risk for a man-in-the-middle (MITM) attack.

AWS recommends forbidding tenants from creating Services having external IP addresses. Customers can enforce this by using admission controllers, including these controllers available on GitHub.

Additionally, AWS recommends forbidding tenants from being able to patch any status fields of any Kubernetes objects. This is not normally permitted, but care should be taken not to enable it by any cluster RBAC policies.

Apply a Seccomp profile to containers

Seccomp is a Linux kernel feature that restricts programs from making unauthorized system calls, or syscalls. Syscalls are how programs interact with the Linux kernel. For example, a program that wants to write to standard output might use the `write(2)` syscall. Many syscalls are harmless, but others can be used to escalate privileges, adjust kernel settings, or perform other undesirable actions.

By default, containers will be run "unconfined," which allows them to invoke any syscall. Instead, AWS recommends enabling the default Seccomp profile provided by the container runtime. This profile allows most system calls, but excludes some that are considered high risk. See Seccomp security profiles for Docker for a list of default permitted and denied syscalls.

To enable this profile, in each Pod or container's `SecurityContext`, specify a `seccompProfile` with a type of `RuntimeDefault`. See Set the Seccomp Profile for a Container for more information.

It is also possible to run a container with a custom Seccomp profile. This can be used to further restrict the syscalls that may be invoked, or permit syscalls that would otherwise be forbidden. Tools such as `strace(1)` or Sysdig Inspect can be used to determine which syscalls an application makes.

Apply SELinux profiles to containers

SELinux is an enhanced security feature that is available in Linux. It was originally developed by the United States National Security Agency (NSA) to provide mandatory access controls to the operating system.

SELinux goes well beyond the basic UNIX permission model by introducing the concept of labeling to processes and files, and fine-grained policies that control what sorts of permissions processes have to access files and perform sensitive operations. If a policy permits the operation, access is granted. Otherwise—even if the UNIX permission model would allow it—access is denied.

AWS recommends enabling SELinux on EC2 instances that host multi-tenant EKS workloads. This requires an SELinux-enabled Linux distribution such as Bottlerocket, Red Hat Enterprise Linux 7 or later, or CentOS 7 or later. On non-Bottlerocket distributions, it also requires an SELinux-enabled container runtime engine such as Docker CE 19 or later. SELinux is not available with Amazon Linux 2 at this time.

When SELinux is enabled, most non-privileged pods will automatically have their own multi-category security (MCS) label applied to them. This MCS label is unique per Pod, and is designed to ensure that a process in one Pod cannot manipulate a process in any other Pod or on the host. Even if a labeled Pod runs as root and has access to the host filesystem, it will be unable to manipulate files, make sensitive system calls on the host, access the container runtime, or obtain kubelet's secret key material.

Here is an example of how to configure an SELinux MCS label for a Pod. In this case, the category IDs are `c123` and `c456`, which you can associate with a unique Pod. (SELinux requires a process have at least two category IDs.)

```
securityContext:
  seLinuxOptions:
    level: "s0:c123,c456"
```

Note

AWS recommends assigning a unique MCS label for each Pod in a cluster. There are edge cases in which MCS labels are not automatically applied, such as when a container has the `hostPID` flag enabled.
Privileged Pod processes have an SELinux label: (`system_u:system_r:spc_t:s0`) that allows them full access to the container host. Therefore, it remains necessary to supplement SELinux with additional controls that prevent creating privileged pods or enabling the `hostPID` flag.
The AWS VPC Container Networking (CNI) controller must be run in privileged mode on nodes running SELinux.

Use admission controllers to enforce security policies

Admission controllers are a powerful feature in Kubernetes. These controllers intercept requests to create new objects or mutate existing objects in a cluster, and take one or more actions. Admission controllers can modify a request to conform to a designated policy (a "mutating webhook"), or they can reject a request altogether (a "validating webhook").

AWS recommends that customers running multi-tenant clusters implement one or both of the following security policy enforcement mechanisms.

Pod Security Policies (PSPs)

Every EKS cluster comes with a built-in admission controller capable of enforcing Pod Security Policies (PSPs). These policies are ordinary Kubernetes objects that a cluster administrator can create. For details, see Pod Security Policies.

Here is an example of a policy that forbids running privileged Pods:

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: DisallowPrivilegedPods
spec:
  privileged: false
  # The rest fills in some required fields.
  seLinux:
```

```
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  volumes:
  - '*'
```

A more complex policy can be found in Appendix: Strict pod security policy for an untrusted tenant. This policy does the following:

- Disallows privileged pods
- Disallows privilege escalation
- Requires all capabilities be dropped
- Forbids host volumes from being mounted
- Forbids using host networking, Inter-Process Communication (IPC) with the host, and using host process IDs (PIDs)
- Forbids running as root
- Requires a default Seccomp profile

By default, EKS provides an unrestricted Pod Security Policy. AWS recommends removing the default cluster role binding of the `eks.privileged` policy to all authenticated users. You can do this by editing the `eks:podsecuritypolicy:authenticated` cluster role binding to remove the `system:authenticated` group from the subject list. If you have created an alternative administrator group for your cluster, you can replace the system:authenticated group with your administrator group instead:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: eks:podsecuritypolicy:authenticated
  annotations:
    kubernetes.io/description: 'Allow all authenticated users to create privileged
 pods.'
  labels:
    kubernetes.io/cluster-service: "true"
```

```
    eks.amazonaws.com/component: pod-security-policy
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: eks:podsecuritypolicy:privileged
subjects:
  - kind: Group
    apiGroup: rbac.authorization.k8s.io
    # Replace this with your administrator group name
    name: system:authenticated
```

> **Warning**
>
> Be careful when making these or other changes to your cluster. They may prevent you from creating new pods until replacement policies, appropriate roles, and role bindings are created.

Open Policy Agent (OPA)

Open Policy Agent (OPA) is a powerful, open-source general-purpose policy agent. At its core, OPA evaluates configurations against a set of rules you define, using a domain-specific language called Rego. Although OPA is flexible enough to work with just about any kind of structured data, it is most frequently used to enforce policies inside Kubernetes clusters.

OPA is capable of providing much more extensive policy management than a Pod Security Policy. PSPs are limited to Pods, while OPA can manage nearly any kind of Kubernetes object. And while PSPs are only able to apply a limited set of policies to a pod, OPA can apply powerful validators such as pattern matchers to any field in an object. For example, with OPA, you can also require that all container images be pulled from a trusted image repository.

The following is an example of a Rego policy that prohibits the creation of privileged containers:

```
package kubernetes.admission

deny[message] {
    # match only if a Pod is being created
    input.request.kind.kind == "Pod"

    # examine each container
```

```
    container := input.request.object.spec.containers[_]
    # match if privileged is set
    container.securityContext.privileged
    message := sprintf("Container %v runs in privileged mode.", [container.name])
}
```

OPA is rapidly evolving. Customers can choose from several different implementations to run in their EKS clusters. Kube-mgmt is the original implementation and is still widely used. Gatekeeper is the newest implementation and has a powerful template-based configuration model.

Conclusion

Multiple approaches and methods exist to secure multi-tenant workloads in Amazon EKS clusters. The best way to ensure complete separation of mutually-untrusted workloads is by operating a dedicated EKS cluster for each tenant. Nevertheless, there are many mitigating controls you can apply that can help you achieve a higher level of security for multi-tenant workloads on a shared cluster.

New techniques for improving container isolation are on the horizon. Technologies such as Firecracker (an AWS-built open-source lightweight virtual machine manager) and Bottlerocket (an AWS-built open-source container-oriented Linux distribution) are undergoing development. Eventually, AWS expects these technologies to be incorporated into production-grade solutions for AWS customers running siloed multi-tenant workloads on Kubernetes. AWS will provide updates as these solutions become available.

Contributors

Contributors to this document include:

- Michael Fischer, Senior Specialist Solutions Architect (Containers), Amazon Web Services
- Tod Golding, Principal Partner Solutions Architect (SaaS), Amazon Web Services

Document history

To be notified about updates to this whitepaper, subscribe to the RSS feed.

Change	Description	Date
Initial publication	Whitepaper first published.	June 4, 2021

Note

To subscribe to RSS updates, you must have an RSS plug-in enabled for the browser that you are using.

Appendix: Strict pod security policy for an untrusted tenant

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: Tenant
  annotations:
    seccomp.security.alpha.kubernetes.io/allowedProfileNames: 'docker/default,runtime/
default'
    seccomp.security.alpha.kubernetes.io/defaultProfileName:  'runtime/default'
spec:
  privileged: false
  # Required to prevent escalations to root.
  allowPrivilegeEscalation: false
  # This is redundant with non-root + disallow privilege escalation,
  # but we can provide it for defense in depth.
  requiredDropCapabilities:
    - ALL
  # Allow core volume types.
  volumes:
    - 'configMap'
    - 'emptyDir'
    - 'projected'
    - 'secret'
    - 'downwardAPI'
    # Assume that persistentVolumes set up by the cluster admin are safe to use.
    - 'persistentVolumeClaim'
  hostNetwork: false
  hostIPC: false
  hostPID: false
  runAsUser:
    # Require the container to run without root privileges.
    rule: 'MustRunAsNonRoot'
  seLinux:
    # This policy assumes the nodes are using AppArmor rather than SELinux.
    rule: 'RunAsAny'
  supplementalGroups:
    rule: 'MustRunAs'
    ranges:
      # Forbid adding the root group.
```

```
    - min: 1
      max: 65535
  fsGroup:
    rule: 'MustRunAs'
    ranges:
      # Forbid adding the root group.
      - min: 1
        max: 65535
```

AWS Glossary

For the latest AWS terminology, see the AWS glossary in the *AWS Glossary Reference*.

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided “as is” without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.