Whitepaper

Elevating Kubernetes Security at Fastly

fastly

Table of Contents

About the Author

Roshan Daneshvaran has a diverse work experience in the field of cybersecurity and information technology. Roshan is the founder of Microstack, a company that provides strategic guidance and technical expertise to enhance clients' Cloud and Kubernetes posture and cybersecurity resilience. Roshan prioritizes security and compliance across all services and aims to modernize applications and infrastructure.

Roshan currently holds the position of Senior Principal Security Architect at Fastly, leading high-impact projects to secure cloud, platform, and container environments.

Introduction

Imagine waking up to find that your company's entire operation has ground to a halt because an attacker found a backdoor into your Kubernetes clusters. The chaos, the downtime, the potential loss of customer trust — it's a nightmare scenario for any organization. Now, picture a world where such threats remain just that: a nightmare. At Fastly, we have implemented a fortified security strategy to anticipate and mitigate risks effectively.

As we modernized and scaled our application platforms with Kubernetes, security became the foundation of our design. Kubernetes' immense power comes with its own set of challenges, requiring a structured, proactive approach. As the Senior Principal Security Architect, I've collaborated with our Cloud and Container Services team to elevate the security and compliance posture of Fastly's Kubernetes ecosystem.

This article explores Fastly's security strategy and best practices, which form the foundation of Elevation-Fastly's Kubernetes Ecosystem, and delves into the Kubernetes threat landscape. We discuss threat modeling, image security, network security and AuthZ/AuthN in Kubernetes, and our approach to these.

The Problem: A Complex and Expanding Threat Landscape

Kubernetes has become the backbone of modern cloud-native architectures, providing unmatched flexibility and scalability in managing containerized applications. However, this widespread adoption also makes it a prime target for attackers. With Kubernetes adoption reaching 60% among enterprises, and up to 96% among respondents in CNCF surveys (CNCF), its APIs, ingress controllers, and service endpoints create multiple entry points for exploitation.

As Kubernetes continues to evolve, so do the tactics of malicious actors. Organizations must embed security best practices such as strong access controls, continuous vulnerability scanning, and real-time monitoring at every stage. At Fastly, we recognized these challenges and took decisive steps to ensure our Kubernetes infrastructure is resilient and secure.

Our Solution: The Elevation Ecosystem

At Fastly, we see security not as an obstacle but as an enabler. By embedding security into the very fabric of our development process, we ensure that it serves as a productive ally. Through clear, “guardrailed” pathways, developers are empowered to focus on building amazing applications, free from the burdens of complex security requirements.

We have established a secure Kubernetes environment called **Elevation**, built on open-source and CNCF products, providing a robust foundation for both our critical control plane and internal systems. By balancing innovation and resilience through a proactive, developer-centric approach to security, Elevation empowers application teams to innovate without the constraints and complexities of managing infrastructure. Our robust security strategy effectively addresses the unique challenges of Kubernetes, ensuring Elevation’s resilience against emerging threats.

Our Guiding Principles

1. **Proactive Security: Catch Issues Early, Reduce Costs**

   Security is most effective—and cost-efficient—when addressed early in the software development lifecycle. At Fastly, security is embedded from the start, ensuring vulnerabilities and misconfigurations are caught before they become costly problems in production.
   By integrating security into CI/CD pipelines, we stay ahead of threats rather than reacting to them. Automated vulnerability scans, compliance checks, and policy enforcement ensure that every container image and deployment meets rigorous security standards. This approach minimizes risk while maintaining the speed and agility developers need to innovate. Security isn't a blocker—it's an accelerator, enabling confident and secure development from day one.

2. **Guardrails Over Tollgates: The Secure Path is the Easy Path**

   We make the secure way the easy way by providing automated, compliant, and opinionated workflows that seamlessly integrate security into development. These guardrails act as predefined pathways, ensuring security best practices are enforced without friction.
   Unlike traditional tollgates—manual checkpoints that slow progress—our approach automates security checks and policy enforcement throughout the entire development and deployment process. This creates a seamless experience where developers focus on building, while security is naturally woven into their workflows. Innovation and security are not at odds—they work together.

3. **Platform Engineering: Enabling Developers, Securing Infrastructure**

   We embrace Platform Engineering to remove the burden of infrastructure and security from developers, allowing them to focus on building and shipping great products. The Platform teams manage foundational infrastructure—VMs, container registries, Kubernetes infrastructure, CICD & observability systems, and security tooling—ensuring it remains resilient, scalable, and patched regularly.
   By clearly separating duties, we provide engineers with a reliable, self-service *internal developer platform* that abstracts complexity while enforcing security best practices.

Threat Modeling for Kubernetes

The following section of this article outline our best practices for building and securing a Kubernetes environment, and details our approach to designing Elevation Security's core components.

In any security journey, it's important to know your enemy and the battlefield. Threat modeling is the practice of identifying potential threats, attack vectors, and vulnerable points in a system before an attack happens. We will outline the threat landscape for Kubernetes and examine the sequence of steps an attacker might take to breach a cluster. By understanding how an attack might progress, we can better plan defenses at each step.

Identifying Threat Actors and Vectors

Who might want to attack a Kubernetes cluster and why?

- **Malicious external attackers:** These could be cybercriminals looking to steal data, hijack computing resources (for crypto mining), or disrupt services for ransom or political motives. They typically start with no access and try to find a way in from the outside.
- **Insider threats:** A rogue or careless insider (a developer, admin, or even someone at your cloud provider) could misuse their access. This might not always be intentional malice – sometimes insiders accidentally expose credentials or misconfigure systems – but insider access can bypass many external protections.
- **Supply chain attackers:** These target the components that you use to build or run your cluster – for example, inserting malicious code into a container image or open-source library, or compromising a CI/CD pipeline to push vulnerable deployments. In a Kubernetes context,

a supply chain attack might involve a tainted Docker image or a malicious Helm chart that an organization unknowingly deploys.

Kubernetes, being a complex system, has a broad attack surface:

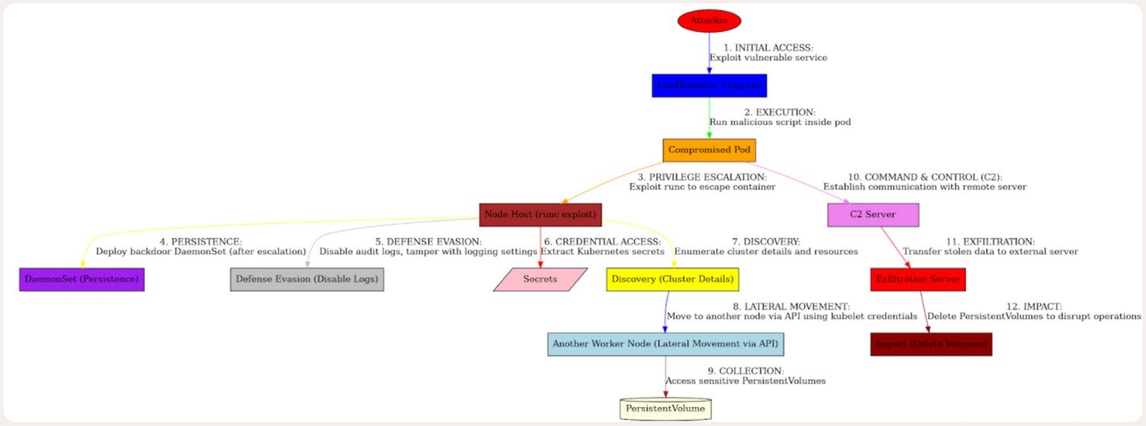
- **Application-level vulnerabilities:** The apps running in your containers might have bugs that can be exploited (just as in traditional servers). For instance, a vulnerable web application could give
an attacker a foothold inside a container
- **Container and runtime vulnerabilities:** The container itself or the runtime (Docker, etc.) might have weaknesses (like the infamous runC vulnerability that allowed container escape). A container running as root or with excessive privileges is especially risky, as it might allow an attacker to break out to the host.
- **Misconfigurations:** Perhaps the most common threat. Kubernetes configuration is powerful, but mistakes happen – e.g., leaving the
Kubernetes Dashboard open with no password, setting overly broad permissions in RBAC, using default credentials for some service, or not restricting public network access to the control plane. Attackers often scan for clusters with
such misconfigurations because they are low-hanging fruit.
- **Credential compromise:** If attackers obtain credentials (cloud API keys, kubeconfig files, Docker registry passwords, etc.), they may authenticate as a legitimate user. This could happen via phishing, reading credentials left in code repositories, or exploiting a pod that has access to sensitive credentials.
- **Denial of service (DoS):** Attackers might aim to crash your cluster or apps, e.g., by exhausting resources. This could be done by exploiting a weakness to cause a memory leak, or simply by spamming your API server with requests. Kubernetes adds resiliency, but also some new DoS angles (like spawning excessive pods if an attacker can trick the system).
- **Underlying infrastructure:** Don't forget, Kubernetes ultimately runs on VMs or physical hosts. If an attacker can compromise the underlying host (through a cloud vulnerability or insecure SSH, etc.), they can subvert the cluster from beneath. Similarly, attacks on the network or storage layers (DNS, load balancers, etc.) can impact Kubernetes.

These threat vectors often chain together in an attack.

Think Like an Attacker: Understanding the Kubernetes Kill Chain

Let’s step into the mindset of an attacker. Defenders often think in terms of controls, policies, and configurations, but attackers see an interconnected system—one where a single weakness can lead to a full-blown compromise.

The diagram below outlines a realistic Kubernetes attack scenario, showing how an attacker moves from an initial foothold to achieving persistence, lateral movement, data theft, and eventually, operational disruption. By understanding these tactics, we can better design defenses that don’t just react but anticipate attacker behaviors.


Attacker
1. INITIAL ACCESS: Exploit vulnerable service
LoadBalancer / Ingress
2. EXECUTION: Run malicious script inside pod
Compromised Pod
3. PRIVILEGE ESCALATION: Exploit runc to escape container
10. COMMAND & CONTROL (C2): Establish communication with remote server
Node Host (runc exploit)
C2 Server
4. PERSISTENCE: Deploy backdoor DaemonSet (after escalation)
5. DEFENSE EVASION: Disable audit logs, tamper with logging settings
6. CREDENTIAL ACCESS: Extract Kubernetes secrets
7. DISCOVERY: Enumerate cluster details and resources
11. EXFILTRATION: Transfer stolen data to external server
DaemonSet (Persistence)
Defense Evasion (Disable Logs)
Secrets
Discovery (Cluster Details)
Exfiltration Server
8. LATERAL MOVEMENT: Move to another node via API using kubelet credentials
12. IMPACT: Delete PersistentVolumes to disrupt operations
Another Worker Node (Lateral Movement via API)
Impact (Delete Volumes)
9. COLLECTION: Access sensitive PersistentVolumes
PersistentVolume

At a high level, an attacker starts by exploiting a weakness—often a misconfigured or vulnerable public-facing service. From there, they escalate privileges, disable security controls, steal credentials, and move laterally across the cluster. The goal is to extract valuable data or disrupt operations, often by targeting persistent storage, which can be catastrophic in cloud-native environments.

Looking at this attack tree, we can identify key inflection points where defenders have opportunities to detect and mitigate threats. Whether it's hardening ingress points, securing secrets, or enforcing network policies, thinking like an attacker allows us to strengthen Kubernetes security in a methodical way.

As we progress through this article, we'll map these adversarial techniques to structured security framework **MITRE ATT&CK** and explore how to systematically defend against them. By the end, you should not only recognize these attack patterns but also be able to construct similar attack trees, applying this mindset to assess and fortify your Kubernetes environments.

The Kubernetes Attack Lifecycle: From Initial Access to Impact

Let's break down a typical Kubernetes attack into stages, from an attacker's perspective, based on the MITRE ATT&CK framework, which also serves as the basis for our Threat Modeling and Strategy in securing Elevation. Not every attack will follow this exact order, but these stages are commonly observed in successful compromises. We will also briefly note what defenders can do at each stage.

1. **Reconnaissance:** The attacker collects information about the target. In the context of Kubernetes, this may involve scanning the internet or the target's network for open ports that signify the presence of a Kubernetes component. In more advanced scenarios, social engineering tactics may be employed. Here are some typical methods used for reconnaissance:

   - **Port Scanning:** The attacker scans the internet or internal network for open Kubernetes components, such
     as an exposed API server, etcd database, or dashboard UI.
   - **Credential Hunting:** Searching for leaked kubeconfig files, credentials in GitHub repositories, or other misconfigurations that could expose Kubernetes secrets.
   - **Social Engineering:** Gathering intelligence from social media, conference talks, or public documentation to infer Kubernetes architecture and potential weaknesses.

Defender's angle: *Reduce your public footprint. Don't expose the Kubernetes API to the internet unless necessary (use VPNs or trusted IPs), avoid default ports when possible, and monitor for scanning activity. Use tools like kube-hunter in a controlled manner to see what an attacker might find during recon.*

2. **Initial Access:** This is where the attacker actually gains a foothold. It could happen in many ways. They may employ a combination of the following techniques for getting an entry point:

    - **Application Exploit:** Gaining access via an insecure containerized application (e.g., exploiting a web app vulnerability to execute code in a pod).
    - **Kubernetes Component Exploit:** Taking advantage of a known vulnerability in the Kubernetes API server, kubelet, or other cluster components.
    - **Misconfiguration Abuse:** Exploiting weak security settings, such as an unprotected Kubernetes Dashboard or overly permissive RoleBindings.
    - **Stolen Credentials:** Using compromised kubeconfig files or leaked service account tokens to authenticate to the cluster.

Once they have some access – say, the ability to execute commands in one container, or access to the API with limited rights – the kill chain has begun in earnest.

Defender's angle: *Apply strong perimeter defenses and hardening. This means keeping Kubernetes updated (to fix known vulnerabilities), eliminating easy misconfigurations, and using authentication everywhere. Also, minimal privileges: if an attacker lands in a frontend web server container, that container should not have credentials or high privileges that allow more damage easily.*

3. **Execution:** If the initial access didn't already execute code (e.g., if they got credentials, now they use them to run some malicious workload), this stage is about the attacker running their code or commands inside the cluster. For instance, after getting into a container, they might run a reverse shell to start controlling it interactively, or they might drop a malicious binary (like a crypto miner) to run inside the pod. If they have API access, they might deploy a new pod (perhaps a privileged one) to do their bidding.Application Exploit: Gaining access via an insecure containerized application (e.g., exploiting a web app vulnerability to execute code in a pod).

    - **Running Malicious Commands:** Executing shell commands within a compromised pod to gain further control.
    - **Deploying Malicious Workloads:** Creating rogue deployments, DaemonSets, or CronJobs to execute attacker-controlled code persistently.
    - **Abusing API Access:** Using valid but excessive API permissions to modify running workloads or escalate access.

Defender's angle: *This is where runtime security tools come in. Technologies like [Falco](#) (an open source runtime security tool) can detect suspicious behavior (e.g., a shell spawning inside a container that normally doesn't use one, or unusual file access patterns). Network monitoring might catch connections from a pod to an unknown external server (for C2 communication). Admission controllers can also prevent execution of unapproved workloads (e.g., don't allow an untrusted image to be run, which could stop some malicious deployments).*

4. **Persistence:** Skilled attackers, once in, will try to ensure they can stay in. In Kubernetes, if they popped a single pod, that pod might be ephemeral (it might get restarted or replaced). So they might want to create a backdoor for persistence. Examples:

   - **Backdoor Accounts:** Creating high-privilege service accounts with secret tokens for long-term access.
   - **Malicious DaemonSets:** Deploying workloads that restart persistently across nodes, making removal difficult.
   - **Scheduled Jobs:** Using CronJobs to execute malicious actions at regular intervals.

Defender's angle: *This is tricky – the best approach is to prevent getting to this stage. However, monitoring changes in the cluster can help; for example, suddenly seeing a new high-privilege service account or an unfamiliar deployment in the cluster should raise red flags. Kubernetes audit logs and tools that can baseline your cluster state (like configuration management tools) are useful. In cloud-managed clusters, ensure CloudTrail (AWS) or equivalent logs for API calls are enabled to catch creation of new roles or resources.*

5. **Privilege Escalation:** The attacker attempts to gain higher-level privileges than they initially had. If they started in a low-privilege container, they might look for ways to become root in the container or escape to the node. Kubernetes-specific escalation might involve:

   - **Container Escape:** Exploiting a misconfigured or vulnerable container to gain access to the host.
   - **Service Account Abuse:** Using an overprivileged service account to create pods or access cluster-wide secrets.
   - **API Server Exploitation:** Taking advantage of Kubernetes vulnerabilities (such as CVE-2018-1002105) to escalate privileges via the kubelet API.

Defender's angle: *Principle of least privilege is key. Ensure that if one part of your system is compromised, it can't easily escalate. Concretely: run containers as non-root with minimal Linux capabilities, avoid hostPath volumes (or restrict them via Admissiom Controllers), lock down RBAC so service accounts have only what they absolutely need.*

6. **Defense Evasion:** Once inside the cluster, an attacker will attempt to evade detection to maintain access and continue their objectives undisturbed. This involves concealing their activities, bypassing monitoring tools, or disabling security controls. In Kubernetes, common evasion techniques include:

   - **Disabling Logging and Monitoring:** Attackers with API or node access may attempt to disable Kubernetes audit logging, delete logs, or tamper with monitoring agents (e.g., killing a Falco DaemonSet or modifying Prometheus alert rules).
   - **Masquerading as Legitimate Workloads:** Instead of deploying obviously malicious containers, an attacker might modify existing Deployments, inject malicious sidecars, or use kubectl exec to blend in with normal operations.
   - **Process Hiding and Kernel-Level Manipulation:** If an attacker gains privileged access to a node, they may use techniques like LD_PRELOAD hijacking, eBPF-based process hiding, or modifying cgroups to conceal their activity.
   - **Tampering with Security Policies:** Attackers might edit or delete Network Policies, Pod Security Standards, or Admission Controllers to relax security restrictions without raising immediate alarms.
   - **Cleaning Up Artifacts:** To cover their tracks, attackers may remove container file system modifications, delete evidence of executed commands from .bash_history, or restore modified Kubernetes resources to their original state.

Defender's angle: *Ensure logs are stored externally and monitor for sudden gaps or deletions. Restrict API and RBAC permissions to prevent attackers from disabling security tools or modifying defenses. Runtime security tools like Falco can catch suspicious activity, while network policies and file integrity monitoring help detect tampering. The goal is to make evasion difficult, increasing the chances of early detection.*

7. **Lateral Movement:** Now the attacker has some level of access, perhaps even high privileges on one node or in one namespace, and they want to move to other parts of the environment. They might:

   - **Network Scanning:** Mapping internal services by probing open ports within the cluster.
   - **Service Account Abuse:** Using an overprivileged service account to access different namespaces or workloads.
   - **Node Compromise:** If an attacker gains access to a node, they may attempt to exploit the kubelet API or move laterally across other nodes.

Defender's angle: *Network segmentation and strict RBAC are the primary defenses. Network Policies should limit which pods can talk to which, making it harder for an attacker in one compromised pod to reach others. Also, not all pods should be able to talk to control plane components except through the API server. If an attacker has compromised a container, having strong isolation between namespaces (and using separate credentials for different applications) can limit what they can do next. Monitoring lateral movement can be done by network flow logs or IDS systems, but those can be complex in Kubernetes – however, unusual connections (like a frontend pod suddenly querying a database it never touched before) might be detectable.*

8. **Credential Access:** The attacker will search for secrets, credentials, and configuration info to further their goals. They might attempt:

   - **Reading Kubernetes Secrets:** If access allows, attackers may dump secrets to find API keys, database passwords, or cloud credentials.
   - **Extracting Service Account Tokens:** Using a pod's automatically mounted service account token to authenticate with the Kubernetes API.
   - **Harvesting Config Files:** Checking environment variables and mounted config files for sensitive data.

Defender's angle: *Secrets management is vital – Encrypt secrets at rest (so even if etcd is accessed, it's not plain text), and strictly control access to them (RBAC and possibly external secret managers like HashiCorp Vault). Also, avoid putting secrets in environment variables.*

9. Discovery

- **Enumerating API Resources:** Using kubectl or API queries to list all services, routes, and workloads.
- **Probing Network Services:** Scanning for internal services with open ports and weak authentication.
- **Examining Cluster Configurations:** Looking for misconfigurations, unused service accounts, or exposed endpoints.

Defender's angle: *You cannot prevent an attacker from querying information they are authorized to access, but implementing the principle of least privilege ensures that they cannot access data they shouldn't. Audit logging can help track unusual behavior, such as an individual listing a large number of resources in an atypical manner. It's important to restrict Role-Based Access Control (RBAC) access to API calls and monitor for excessive resource enumeration. Additionally, network segmentation can help prevent attackers from scanning the cluster freely.*

10. **Impact (Exfiltration or Damage):** Finally, the attacker achieves their goal, which could be:

- **Data Exfiltration:** Copying out sensitive data (user data, intellectual property). For example, exporting database contents, or even making an entire etcd backup if they have that level of access. They might send data out over the internet from a pod or use cloud APIs if credentials are available (e.g., accessing an S3 bucket).
- **Cryptojacking (Resource Theft):** Installing crypto mining software across pods or nodes to harness your compute for their gain
- **Ransom / Destruction:** Encrypting data or deleting Kubernetes resources to cause denial of service, then demanding ransom. Or simply disrupting services as an end in itself.
- **Cluster Takeover:** Using your cluster as part of a botnet or to pivot into your other infrastructure (maybe from Kubernetes into your corporate network if not isolated).

Defender's angle: *By this stage, if all earlier defenses failed, you're in damage control. Your monitoring and alerting should catch unusual outbound traffic or large data transfers. Egress controls can help (limiting which pods can talk out to the internet, or using DLP systems to monitor data egress). Regular backups of data and cluster state are essential so you can recover if things are wiped or ransomed. And having an incident response plan is key to containing the damage.*

Securing the Kubernetes Software Supply Chain: Images, Builds, and Deployment Pipelines

The first link in our Kubernetes security chain is the software we deploy. If the applications or containers themselves are malicious or vulnerable from the start, it's game over – the attack doesn't even need to break in, because we invited it in. This is why **supply chain security** is critical. In a Kubernetes context, supply chain security focuses on container images, build processes, and deployment pipelines.

Why Supply Chain Security Matters

Consider that a typical container image might be built on a base image (like Debian or Alpine Linux) and include dozens of libraries. Each of those could have known vulnerabilities. Attackers know this, and they actively search for deployments running outdated images that they can exploit. Even more insidious, attackers may hide malware in images that appear benign – for example, a public Docker Hub image named similarly to a popular one, hoping someone will use it by mistake.

Real-world example: In 2018, a security researcher found several malicious images on Docker Hub that had been downloaded millions of times. They contained cryptominers that would start mining cryptocurrency as soon as the container ran, unknowingly costing the users (and their cloud providers) money. This kind of attack doesn't exploit a vulnerability in Kubernetes itself – it exploits trust. If you trust an image from an unverified source, you might be running an attacker's code inside your cluster.

Another vector is the build pipeline. If an attacker can compromise your CI/CD system or the process of building and pushing images, they can insert backdoors. A famous example outside containers is the SolarWinds incident, where attackers compromised the build system to insert malicious code. In container land, that could translate to pushing a malicious layer into your image or stealing credentials from a CI job to later push a fake image.

Supply chain attacks are a big enough concern that even government agencies emphasize them. supply chain risks are one of the top three sources of compromise highlighted by NSA/CISA Kubernetes guidance

The takeaway: we need to ensure that the images and configurations we deploy to Kubernetes are as secure and verified as possible.

Best Practices for Container Image Security

1. **Use Trusted Base Images:** Start your Dockerfiles FROM well-known and trusted base images. Official images from Docker Hub or your OS vendor (like the official Node.js or Python images, or distroless base images from Google) are usually better maintained. Be cautious with random images published by unknown users.
2. **Minimize Image Contents:** The more software in your image, the more potential vulnerabilities. Follow a minimalistic approach:
   - Use slim or alpine variants if possible (e.g., `python:3.11-slim` instead of full `python:3.11`).
   - Remove unnecessary packages and dependencies. If you only need a single binary, consider using scratch or distroless images which contain almost nothing except your app and minimal runtime.
   - Multi-stage builds can help produce lean final images (build in one stage, then copy only the needed artifacts into a small runtime image).

This reduces the attack surface. NIST recommends using minimal base images and avoiding unnecessary software

3. **Regularly Scan Images for Vulnerabilities: Incorporate vulnerability scanning into your development pipeline. There are many tools (both open-source and commercial):**
   - **Trivy** (open source by Aqua Security) – scans container images for known vulnerabilities in OS packages and language-specific dependencies.
   - **Anchore Engine** or its CLI tools like **Syft/Grype** – open source image scanners.
   - **Clair** – an open source scanner that can integrate with registries like Quay.
   - **Snyk, Docker Hub's built-in scanner**, etc. – there are also services that scan images in registries.

Scanning should be done before deployment (e.g., as part of CI), in addition to runtime. If critical vulnerabilities are found, fix them (update the package or image). Some organizations enforce policies to block deploying images with high-severity vulnerabilities until they're addressed. The NSA/CISA guidance actually suggests using an admission controller to prevent vulnerable images from running – meaning Kubernetes would reject a pod if its image hasn't passed a vulnerability scan (though implementing that requires some pipeline integration).

4. **Update Base Images and Dependencies Frequently:** New vulnerabilities appear all the time. If you built an image six months ago and haven't updated it, it likely has known issues now. Establish a routine to rebuild images with updated base images and libraries. Dependabot or similar tools can alert on outdated dependencies even in container contexts. Also track security advisories for your base images.
5. **Verify Image Provenance (Image Signing):** To ensure that the image you deploy is exactly the one you intended and not tampered with, use image signing. Docker has a content trust feature (Notary v1) and the modern approach is using Sigstore Cosign to sign images. This involves generating a cryptographic signature for an image and then configuring your cluster (or deployment pipeline) to verify that signature.
    - For example, you can sign an image with Cosign, and then an admission controller (like an OPA Gatekeeper policy or Kyverno rule, or Cosign's own webhook) can check that any image being launched is signed by your trusted key. This prevents an attacker from tricking you into running an image they built.
    - While implementing signing can have setup overhead (managing keys, etc.), it's increasingly recommended for production environments, especially where compliance requires ensuring software integrity.
6. **Use Private Registries and Access Controls:** Host your images in a private registry that

    you control (or a trusted cloud registry like ECR, GCR, etc., with proper permissions). Public registries are fine for base images and open-source components, but your custom app images should be in a private repo. Lock down who can push images to the registry, and use unique, strong credentials for the registry. This reduces the risk of an attacker pushing a malicious image with the same name as one of your apps.
7. **Enforce Image Policies in Kubernetes:** Kubernetes has some built-in and add-on mechanisms:
    - The AlwaysPullImages admission plugin (when enabled on the API server) forces every new pod to pull its image from the registry, rather than using a potentially cached copy on a node. This ensures that if someone had somehow planted an image on a node with the same name, it won't be used. It also means you always get the latest version of the image (if you updated it).
    - You can use Open Policy Agent (OPA) Gatekeeper or Kyverno to write policies like "only allow images from my registry" or "disallow the :latest tag in images" (because using `:latest` is discouraged – it's better to use an explicit version tag to avoid unpredictability).
    - Kubernetes' Pod Security Standards (baseline/restricted) also cover some aspects like not running as root, which is more runtime, but image policy can tie into that (ensuring USER in Dockerfile, etc).

By combining these, you ensure that even if a developer accidentally tries to run `randomuser/ubuntu:latest`, the system will reject it because it's not from your approved repository or it's not signed.

8. **Secure Build and CI/CD Systems:** This goes slightly beyond Kubernetes itself, but since images come through CI/CD:

    - Lock down CI credentials (don’t embed kubeconfig with cluster-admin in plaintext in CI scripts, for example).
    - Ensure the CI environment is patched and monitor it – an attacker who gets into CI could manipulate your manifests or images.
    - Use ephemeral CI runners for sensitive jobs, so there’s no long-lived environment to infect.
    - If using Kubernetes to deploy (like GitOps or Flux, or Argo CD), ensure those deployment pipelines are also secured (Argo should have RBAC, etc.).
    - Consider employing tools to scan your Kubernetes manifests (YAML or Helm charts) for misconfigurations before they are applied. Tools like conftest, kube-score, kube-linter, or Checkov can automate static analysis of configs (ensuring, for instance, you didn’t accidentally set `allowPrivilegeEscalation: true` or leave a debug port open).

Kubernetes Network Security – Segmentation and Traffic Control

Kubernetes networking allows all pods and services to talk to each other by default, which is convenient for development but potentially dangerous in production. If an attacker compromises one pod, they could move laterally by connecting to other pods, including those in different namespaces, unless we put guardrails in place. In this chapter, we focus on network security for Kubernetes: how to restrict traffic between pods (and to/from the cluster) using network policies and other mechanisms, and how to introduce layers like service mesh for zero-trust networking.

The Basics of Kubernetes Networking

Out of the box, Kubernetes imposes a flat network:

- Every pod gets an IP address, and by default **pods can communicate with each other by IP across the cluster, regardless of namespace or which node they're on.** There is no built-in isolation; network isolation is a responsibility of the network plugin via NetworkPolicies (if used).
- Services provide stable IPs and load balancing to pods, but from a network perspective, they typically don't restrict source/destination (unless using specific Service types or external firewall rules).
- The Kubernetes **kube-proxy** handles routing for Services (using iptables or IPVS rules on nodes). It doesn't filter traffic; it just ensures it goes to the right pod.
- By default, Kubernetes doesn't come with a firewall between pods. It assumes a trust zone (the cluster network).

This means:

- If Mallory gains access to one pod, she can scan the entire cluster's pod IP range and likely find and connect to other pods (e.g., database pods, internal services) unless additional measures are in place.
- All pods can also typically reach the internet (for updates, etc.), unless restricted by cloud-level egress rules or network policies.

Network Policies – Kubernetes Firewall

NetworkPolicy is a Kubernetes resource (in the `networking.k8s.io` API group) that acts like a firewall policy for pods at the IP level (Layer 3/4). Network policies allow you to specify which connections are allowed to or from a set of pods; any traffic not explicitly allowed by any applicable policy is blocked (default deny).

Important points:

- NetworkPolicies are **namespaced**. They only govern pods within their namespace (they can, however, allow/deny traffic to/from pods in other namespaces if selected).
- They are implemented by the CNI plugin (Calico, Cilium, Weave, etc.). If your cluster's network plugin doesn't support network policies, creating the resource will do nothing. Most managed Kubernetes offerings have network policy support (either via their default CNI or an option to enable one like Calico).
- A NetworkPolicy can have rules for **ingress (incoming to pods)** and/or **egress (outgoing from pods)**. You can restrict one or both.
- They work by selecting pods via labels (similar to how Services or Deployments select pods).

By default, if you don't create any NetworkPolicy, it's as if there is an allow-all policy (i.e., no restrictions). Once you create one or more policies that select a pod, any traffic not allowed by those policies is blocked. To effectively lock down, one common pattern is to create a "default deny" policy for a namespace (which selects all pods but allows nothing), then create specific allow policies for the traffic you need.

Example scenario: In a typical microservice app, you might want:

- The frontend pods to be allowed to talk to the backend API pods.
- The backend API pods to talk to a database service.
- But you might want to prevent the frontend from directly querying the database, or one microservice from talking to another if it's not needed.
- Also, pods from team A's namespace should likely not talk to pods in team B's namespace (if those are unrelated applications), unless via some public interface.

NetworkPolicies can enforce all that, effectively creating micro-segments in your cluster's network.

Key fields in a NetworkPolicy:

- `podSelector:` which pods the policy applies to (often you select by app label or just everything in the namespace).
- `policyTypes:` Ingress and/or Egress.
- `ingress:` list of ingress rules, each rule can allow traffic from certain peers (specified by pod
  labels, namespace labels, and/or IP blocks) to certain ports/protocols.
- `egress:` similarly, rules for outgoing traffic.
- If a pod is selected by a policy and no rule matches a particular traffic, it's denied. If a pod is not selected by any NetworkPolicy, it's unaffected (all traffic allowed).

Implementing Network Policies – Best Practices

- **Start with Namespace Isolation:** If you have distinct applications or environments in separate namespaces, consider a default deny policy for each, so that cross-namespace traffic is prohibited unless explicitly allowed. You can label namespaces and use those labels in network policy selectors to allow only specific cross-namespace communications.
- **Use a Default Deny (Isolation) Policy:**

E.g.,

```
kind: NetworkPolicy
metadata:
   name: default-deny
   namespace: prod
spec:
   podSelector: {} # selects all pods in "prod"
   policyTypes:
   - Ingress
```

Beyond NetworkPolicy: Advanced Traffic Control

Service Mesh (mTLS and Authorization): While NetworkPolicies work at IP/port level, a service mesh (like Istio, Linkerd, Consul Connect) operates at the application layer. Service meshes can encrypt traffic between services with mutual TLS (so even if someone sniffs the network, they see only encrypted data). They can also do request-level authorization (e.g., service A is allowed to call service B's endpoints, but not service C). This is an additional layer (and adds complexity/ resources), but in high-security environments, it ensures zero-trust even within the cluster – every service authenticates to every other service on each call.

Ingress/Egress Controllers: For traffic entering or leaving the cluster, ensure you use a secure Ingress controller or API Gateway. For example, an NGINX ingress controller should be configured to only allow TLS, use security headers, etc. For egress, some companies route all outbound traffic through a proxy for monitoring. Kubernetes doesn't have a built-in egress gateway by default (Istio provides one, or you implement at network level).

DNS Policies: Kubernetes doesn't natively restrict DNS by policy, but some CNI plugins (like Cilium) let you create rules based on DNS names for egress (e.g., only allow access to *.mycompany.com). Also, ensure that your CoreDNS (the internal DNS server) is not accessible from outside and is patched (there have been vulnerabilities in CoreDNS in the past).

Network Segmentation Outside the Cluster: Ensure the cluster's nodes or VPC are segmented from your other infrastructure appropriately. For example, if your Kubernetes cluster is in the same flat network as your database servers not under Kubernetes, a compromised pod might directly try to hit a database. Better to put Kubernetes nodes in their own subnet/VPC and tightly control what they can reach outside of Kubernetes context (some access might be needed, e.g., to a central database, but then use cloud security groups or firewall rules to only allow the Kubernetes master IPs or specific pod CIDRs).

Hardening Kubernetes Access: Authentication, Authorization, and Service Identity

Kubernetes' power is concentrated in its API server – whoever (or whatever) can send commands to the API server effectively controls the cluster. Thus, securing access to the Kubernetes API is one of the most critical aspects of cluster defense. In this section, we discuss how Kubernetes authentication and authorization work, best practices for Role-Based Access Control (RBAC), and how to ensure that only the right people and services have the minimum permissions they need.

Kubernetes Authentication Basics

Authentication is about confirming who you are (or what entity you are). Kubernetes supports multiple authentication methods:

- **Client Certificates:** When Kubernetes is set up (via kubeadm or in managed services), each component and each admin user can have X.509 certificates. For example, you might have a certificate for user "admin". Kubectl uses these (via the kubeconfig file) to authenticate to the API server. Certificates are verified by the API server's CA.
- **Bearer Tokens:** These are like API keys. Kubernetes historically could use static tokens set in a file (not recommended now) or dynamically issued tokens. Service accounts (more on them soon) use bearer tokens – essentially long random strings that are presented with requests.
- **Authentication Plugins:** You can configure the API server to use an external authentication provider. A common practice in enterprises is to use OIDC (OpenID Connect) to integrate with an identity provider (like Keycloak, Dex, or cloud IAM services). That way, users can log in with corporate credentials (SSO) and get a token for Kubernetes.
- **Cloud Provider IAM:** Some managed solutions allow tying into their IAM. For instance, AWS EKS can allow IAM entities to authenticate and then maps them to Kubernetes roles via a config map. Similarly, Azure and GCP have their mechanisms.

By default, in most clusters you interact with, you probably use a kubeconfig file that contains either a client certificate or a token (or it might exec an external auth command to get a token via OIDC). As an admin, you should ensure there are no unused or overly privileged credentials floating around. Disable basic authentication (username/password) and legacy token file auth if they're not needed.

One important built-in account is kubernetes-dashboard (if you have the Dashboard installed) – ensure that is secured (the Dashboard now doesn't have full admin rights by default, but earlier it was a risk). Also, if using kubeconfig files, treat them like passwords; don't accidentally commit them to git repositories (attackers actively search GitHub for kubeconfigs or cloud keys).

Role-Based Access Control (RBAC)

Once a user is authenticated, Kubernetes needs to decide what that user (or service account) is allowed to do. This is authorization. Kubernetes uses RBAC as the main mode (it can also be configured for ABAC or other modes, but RBAC is the standard).

RBAC has a few key concepts:

- **Role and ClusterRole:** A Role defines a set of permissions (verbs on resources) within a specific namespace. For example, a Role can allow "get, list, watch" on pods in the dev namespace. A ClusterRole is similar but not bound to a namespace (it can give access to cluster-wide resources like nodes, or be used in any namespace).
- **RoleBinding and ClusterRoleBinding:** These bind a Role (or ClusterRole) to a user, group, or service account. A RoleBinding is within a namespace (e.g., bind a user to a Role in that namespace), and a ClusterRoleBinding can grant cluster-wide access (or access to a ClusterRole's permissions across all namespaces).

Service Accounts: These are special accounts meant for applications (pods) to use. Every namespace has a default service account, and pods use it by default (unless you specify a different service account in the pod spec). Service accounts have associated tokens that pods can use to call the API server (for example, a pod might list other pods or read a ConfigMap if it has rights). Managing what service accounts can do via RBAC is crucial to limit what a compromised application can do.

RBAC Best Practices:

- **Least Privilege:** Follow the principle of least privilege religiously. Only give users or service accounts the minimum permissions they need to do their job (Role Based Access Control Good Practices | Kubernetes). For example, if an application only needs to read from a specific ConfigMap in its namespace, don't give it a role that can read all ConfigMaps or, worse, all Secrets.
- **Use Namespaced Roles where possible**: If someone only needs access in one namespace, scope a Role to that namespace instead of using a ClusterRole. ClusterRoles should be reserved for cluster-wide operations or for convenience when you truly need the same access in all namespaces.

- **Avoid Wildcards in Permissions:** Rather than giving "*" access to all resources or all verbs, enumerate only the specific ones needed (Kubernetes RBAC: Best Practices & Examples - Kubecost). For instance, an app might need read access to ConfigMaps, but not to Secrets – don't just give it access to all `configmaps, secrets` in one rule if Secrets are not required.
- **Don't Use Cluster-Admin Lightly:** The built-in ClusterRole `cluster-admin` basically gives full control. Only bind this to superuser accounts. If a user just needs to debug pods in all namespaces, consider a custom ClusterRole that lists pods in all namespaces, rather than cluster-admin.
- **Service Account Isolation:** Create separate service accounts for different deployments, especially if they need different levels of access. By default, the service account token is mounted in pods – if they get compromised, the attacker gets that token. If all pods use the default service account which has broad permissions, that's a big risk. Instead, give each app its own service account with limited RoleBinding. If an app doesn't need to call the API at all, you can even set `automountServiceAccountToken: false` in its pod spec to not give it a token at all.
- **Group Management:** If you have many users, manage them via groups in your identity provider and bind the group to roles. For instance, a "developers" group might get view access on a dev namespace, whereas an "ops" group gets broader access. This way, adding a user to the appropriate group in your SSO automatically gives them the right access in Kubernetes.
- **Periodic Audits:** Regularly review who has what access. Kubernetes doesn't have a built-in report for this, but you can use commands like `kubectl get rolebindings --all-namespaces` and `kubectl describe clusterrolebindings` to see mappings. There are also tools (like Fairwinds RBAC Lookup, or OpenShift's oc adm policy who-can) to help enumerate access. Prune any accounts or bindings that are no longer needed (for example, if someone left the team, remove their account binding).
- **Beware of Impersonation Permissions:** Kubernetes RBAC has an ability where a user can be allowed to "impersonate" another user or service account. Don't grant impersonation (verbs like "impersonate" on users) unless required, as it could allow someone to assume a more privileged identity.
- **Default Lockdown:** By default, new clusters have no default RoleBindings giving broad access to normal users (in fact, most clusters start with an empty RBAC, meaning unless you configure it, no one except admin cert-holders can do anything). That's good – you then deliberately grant access. In older versions or some setups, you might have had a "cluster-admin" binding for the `system:authenticated` group (meaning every authenticated user had full run of the cluster!) – ensure that's not the case.

Protecting the API Server Endpoint

RBAC helps control what an authenticated entity can do. But we also should control *who can even reach the API*. Some considerations:

- **Network Access:** Ideally, the Kubernetes API server should not be open to the entire internet. In cloud setups, use security groups / firewalls to restrict the source IP ranges that can talk to the API (maybe only your office network, your VPN, etc.). Attackers can't exploit an API they can't even connect to.
- **Enable Audit Logging:** This isn't access control per se, but it's crucial for visibility. Turn on API server audit logs (with a reasonable policy) so you have an audit trail of who did what. Many compliance regimes (like PCI, NIST 800-53) require auditing administrative actions. Audit logs will show if someone tried to perform actions they weren't allowed to, or if a certain account suddenly did something suspicious (like reading all secrets).
- **Avoid Anonymous Access:** Kubernetes used to allow requests with no auth to be treated as "anonymous" user. Ensure `--anonymous-auth=false` is set on the API server (most modern setups do this by default, or at least RBAC denies anonymous actions). Similarly, disable `--insecure-port` (an old non-TLS port for API, which is gone in recent versions).
- **Protect etcd:** This is slightly aside from API, but the etcd database behind the API is essentially another way to get cluster info. Make sure etcd requires authentication (certs) and is not externally reachable. If an attacker directly accessed etcd (which should be on a protected network segment or localhost), they could read or write cluster state bypassing the API's authz checks.

Service Mesh Security – Enforcing Zero Trust for Workloads

While RBAC and network policies control access at the Kubernetes API and namespace levels, they do not automatically encrypt traffic between workloads or verify service identities at runtime. This is where a service mesh (such as Istio, Linkerd, or Consul) provides an additional layer of security by enforcing mTLS (mutual TLS), workload identity, and fine-grained authorization policies for service-to-service communication.

Mutual TLS (mTLS) – Encrypting Pod-to-Pod Traffic

By default, network traffic inside a Kubernetes cluster is unencrypted. Attackers who compromise a pod or intercept traffic may eavesdrop on sensitive data or perform Man-in-the-Middle (MITM) attacks.

- A service mesh automates TLS encryption for all service-to-service communication (not just API-to-service).
- Even if attackers move laterally inside the cluster, they cannot decrypt intercepted traffic.
- Example: Istio and Linkerd automatically inject sidecar proxies to establish mTLS between workloads.

Workload Identity – Moving Beyond IP-Based Security

Traditional Kubernetes NetworkPolicies rely on IP-based restrictions, which can be bypassed if an attacker gains access to an allowed pod. Service mesh enforces identity-based security instead.

- Each workload receives a cryptographic identity (e.g., SPIFFE IDs in Istio).
- Service-to-service authentication happens at Layer 7 (not just IP-based network controls).
- Example: Even if an attacker spoofs an IP or moves laterally, they cannot impersonate a valid workload identity unless they compromise the mesh itself.

Fine-Grained Authorization – Beyond RBAC

- RBAC ensures users have correct Kubernetes API permissions, but it does not control which microservices can talk to each other.
- Service meshes enable service-to-service RBAC-like policies using authorization policies.
- Example: Istio Authorization Policies define which workloads can call which services based on identity, JWT tokens, HTTP headers, etc.
- This prevents attackers from pivoting between microservices, even if they breach one.

Workload Security – Hardening Pods and Containers

Even with strong perimeter defenses, we must assume that an attacker might eventually run code in a container (via a software vulnerability or misconfiguration). Workload security is about limiting what an attacker can do inside a compromised container. This means configuring pods and containers to run with the least privileges necessary, and using features of Kubernetes and Linux to sandbox and restrict their capabilities. In this section, we'll cover how to harden your pods: avoiding running as root, dropping Linux capabilities, using read-only filesystems, and more.

Pod Security Standards and Policies

Kubernetes has a built-in concept of Pod Security Standards (PSS) which defines three levels:

- **Privileged:** Essentially no restrictions, pods can do all sorts of host-level access (should be avoided except in special cases).
- **Baseline:** Restricted to prevent known bad practices, but still allows some flexibility. E.g., baseline disallows privileged containers, but might allow running as root if needed.
- **Restricted:** Tightest level – enforce nearly all best practices (must run as non-root, no host namespace access, etc).

As of Kubernetes v1.25, PodSecurity admission is enabled by default to replace the deprecated PodSecurityPolicy (PSP). You can enforce PSS by namespace:

- By adding labels to a namespace like `pod-security.kubernetes.io/enforce: restricted` (and similar for `audit` or `warn` levels), the API server will reject any pod that doesn't meet the restricted standard in that namespace.

- For example, if a developer tries to deploy a pod that runs as root or has a privileged flag in a restricted namespace, they'll get a rejection.

Our recommendation: aim for the restricted profile on all production namespaces. If something truly needs to be looser, you can use baseline on a case-by-case basis (or a separate namespace for that component).

If your cluster is older or if you prefer more control:

- **PodSecurityPolicy (PSP):** was an older mechanism where you create PSP objects and grant users permission to use them. PSP is removed in 1.25+, but some environments (1.23 or earlier, or OpenShift's SCC which is similar) may still use it. The concepts are similar (restrict host mounts, user, etc).
- **Custom Policies via Gatekeeper/Kyverno:** You can also enforce custom rules using policy engines. For instance, using Open Policy Agent (OPA) Gatekeeper with a policy template to disallow containers running as root, or ensure specific labels, etc. Kyverno (another policy engine) can also mutate or block pods that don't meet criteria.

Container Privileges and Linux Capabilities

User IDs (Don't run as root): By default, if you don't specify a user, containers run as UID 0 (root) inside the container. While this root is namespaced (it's not root on the host unless combined with other privileges), it's still dangerous. If the container process escapes or accesses the host somehow, and it's root, you've essentially given the attacker root on the host. The NSA warns that a container running as root that escapes can provide complete control of the worker node (A Closer Look at NSA/CISA Kubernetes Hardening Guidance | Kubernetes).

Best practice:

- In your Dockerfile, use a non-root USER (like `USER 1000` or a specific user).
- Or at least, in the Kubernetes pod spec, set `securityContext.runAsUser` to a non-zero UID and possibly `runAsNonRoot: true` (which tells Kubernetes to refuse to run the container if the image doesn't have a non-root user).
- Also set `securityContext.runAsGroup` if needed for file permissions, and consider `fsGroup` for shared volume permissions.

By running as a less-privileged user, even if an attacker breaks out of the application, they face another hurdle for host compromise (they might need a kernel exploit to elevate privileges). It also limits damage in multi-container pods (one container might not be able to tamper with another's files if permissions are set right).

Privileged flag: Pods (actually containers) have a boolean flag `securityContext.privileged`. If true, that container essentially has all the capabilities of the host's root (it's as if it's not in a namespace for certain restrictions). Privileged containers can modify the host (load kernel modules, access all devices, etc). You should **almost never need privileged** in a normal app. It's used for low-level system daemons or debugging. Keep it off (false) for all application pods. Many admission policies will outright forbid privileged containers cluster-wide.

Linux Capabilities: Linux divides root privileges into subsets called capabilities. By default, Docker (and Kubernetes) give containers a limited set of capabilities (like the ability to send raw network packets, etc.) and drop others (like no capability to modify kernel parameters, no raw socket by default? Actually, NET_RAW is typically included by default in Docker's baseline, which is why ping works out-of-the-box).

- You can explicitly drop additional capabilities. For example, `securityContext.capabilities.drop: ["NET_RAW"]` will remove the ability to create raw network packets, which means tools like `ping` or `nmap` might not work. This is good to limit network reconnaissance from within a container.
- If a container needs a specific capability (like a monitoring container might need NET_ADMIN to adjust networking), you can add only that capability and drop all others to keep the privilege set minimal.

The default set can be seen in Docker docs; it includes things like AUDIT_WRITE, CHOWN, NET_RAW, etc. Often you want to drop NET_RAW at least, unless needed. Some security benchmarks recommend dropping all capabilities and then adding back needed ones – though in practice you might drop all and add back e.g. SETPCAP if needed.

No New Privileges: There's a securityContext flag `allowPrivilegeEscalation`. This should be set to `false` for most containers. It effectively prevents a process from gaining more privileges (for example via calling setuid). If you run as a non-root user and set allowPrivilegeEscalation=false, then even if some binary has the setuid bit, it won't let the process elevate to root. (In restricted PSS, this is required to be false if you run as non-root.)

Filesystem and Device Permissions

Read-Only Root Filesystem: If your container image doesn't need to write to the root filesystem, you can mount it read-only (`securityContext.readOnlyRootFilesystem: true`). This means even if an attacker gets in, they can't easily install malware or modify scripts on the fly in the container's own filesystem (they would be limited to writing to any ephemeral or mounted volumes that you provided). Many stateless apps can run with a read-only filesystem if coded properly (they might write to /tmp or a mounted emptyDir if needed for temp files). The NSA guide highlights this as an often overlooked but effective hardening step.

Filesystem Groups: Using `fsGroup` in a pod securityContext ensures that mounted volumes are accessible to the intended non-root user. It's more a functionality thing than security, but it prevents situations where someone would be tempted to run as root just to access a volume.

Device Access: By default, containers have no access to host devices (except possibly /dev/random etc). If you don't need to expose any host device (like a GPU or a USB), don't use the `devices`: feature of securityContext. If you do need (say GPU for ML), Kubernetes usually requires using device plugins which handle granting limited access.

Host Namespaces: Avoid using `hostNetwork: true, hostPID: true, hostIPC: true` in pod specs unless absolutely needed. These settings put the container in the host's network, PID, or IPC namespace respectively, which breaks isolation:

- `hostNetwork:true` means the pod shares the node's network stack (it can see all host interfaces, open ports on the host IP, etc.). This might be needed for network infrastructure pods, but it means that pod can sniff host traffic.
- `hostPID:true` means the pod can see all processes on the node. An attacker in such a pod could potentially attach to other processes (with ptrace) if not blocked by something like AppArmor.
- `hostIPC:true` similarly shares IPC namespace (rarely needed).

HostPath volumes: Mounting a hostPath (a directory from the node's filesystem) into a pod is very powerful. It can be useful (for example, a log collector might need to read `/var/log/` on nodes). But it's also a way to break out: e.g., if you mount the host's `/` or `/etc`, a container can modify host files. Or mounting the Docker socket `/var/run/docker.sock` essentially gives full control of the host's Docker (and thus the node). Avoid hostPath unless required, and if you must, scope it narrowly (e.g., only a specific subdirectory, read-only if possible). Admission policies can restrict hostPath usage (e.g., only allow specific directories, or only allow certain trusted service accounts to use them).

Seccomp, AppArmor, and SELinux

These are Linux security mechanisms that can sandbox or confine processes:

- **Seccomp (Secure Computing Mode):** Allows filtering of system calls that a process can make. Docker has a default seccomp profile that blocks many dangerous syscalls (like `keyctl`, which could tamper with kernel keyrings, etc). Kubernetes can use that default if you don't override it. You can also specify `securityContext.seccompProfile` in recent Kubernetes (v1.19+). It's wise to use at least the default seccomp profile for all containers (which is automatically applied in many runtimes). For high security, you might design custom seccomp profiles to disallow even more syscalls for specific containers that don't need them. In Kubernetes 1.25, there's even an alpha feature SeccompDefault which can apply the default profile cluster-wide by default.
- **AppArmor:** A kernel module (on Ubuntu and some other distros) that can restrict file access and capabilities of a process. Kubernetes can set an AppArmor profile per container via an annotation (e.g., `container.apparmor.security.beta.kubernetes.io/<container_name>: runtime/default` or a custom profile name). The `runtime/default` is usually an AppArmor profile that is the Docker default (often unconfined or very lenient). You can create stricter AppArmor profiles (outside K8s) and load them on nodes, and then have pods enforce them. For example, you could have a profile that prevents a container from reading certain host files even if it somehow got access. AppArmor is not available on all systems (and not on COS or Red Hat which use SELinux).

- **SELinux:** Used primarily in Red Hat based systems (and OpenShift clusters). SELinux labels and types can isolate containers. In OpenShift, by default, every pod runs with a unique SELinux context (type) that prevents it from accessing other pods' files even if it somehow could reach them. SELinux also confines what system calls or file paths a process can use based on policy. On vanilla Kubernetes, you can request an SELinux context in the securityContext if your nodes use SELinux (fields: `seLinuxOptions` specifying type, level, user, role). Most people don't manually do this unless they have a specific requirement, but ensuring SELinux is enabled (not set to Permissive or disabled) on the host can add an additional layer of defense.

In summary, seccomp/AppArmor/SELinux are like safety nets: even if an attacker gets some privilege in the container, these can stop certain actions. They are complex to fine-tune, but using the defaults (seccomp default, AppArmor docker-default, SELinux enforcing) already improves security.

Implementing and Verifying Workload Security Settings

Tools and Automation:

- Use config scanners (like kubeaudit by Shopify or kubesec or Checkov) to check YAMLs for risky settings. For instance, kubeaudit can flag pods running as root or with privilege.
- Enforce via CI: e.g., reject deployment YAMLs that don't meet your security baseline (maybe via a GitOps pipeline, or using admission controllers as mentioned).
- Gatekeeper (OPA) can use policies like "no hostPath unless in a specific namespace" or "containers must drop NET_RAW".
- Kyverno can even mutate incoming pods to add safe defaults (like automatically insert `runAsNonRoot: true` if not set).
- Kubernetes now (since 1.19) also surfaces some of this in `kubectl describe` or events if a PSP or PodSecurity prevents something.

Verifying at runtime:

- You can exec into a running pod and check if it's running as expected user: e.g., run `id` command to see UID.
- Check effective capabilities: one way is `grep CapEff /proc/1/status` in the container process. Or install `capsh` to list capabilities.
- Try writing to a filesystem path that should be read-only (it should fail).

Monitoring and Threat Detection in Kubernetes

Despite all preventative measures, we need visibility into our cluster's activity to detect potential intrusions or misuse. Kubernetes introduces new layers to monitor (applications, containers, orchestration events), so our monitoring strategy must cover:

- **Application and Pod Logs:** for diagnosing what apps are doing.
- **Cluster Logs and Events:** like Kubernetes events, audit logs from the API server.
- **Infrastructure Metrics:** CPU, memory, network usage that might indicate anomalies (e.g., crypto mining typically causes high CPU).
- **Security Alerts:** specialized detection of suspicious behavior (like someone spawning a shell in a container, or accessing a sensitive file inside a container).

In this section, we'll discuss how to set up logging and monitoring, and how to leverage open-source tools like Falco for runtime threat detection.

Centralized Logging and Monitoring

Application Logs: Each container writes stdout/stderr which Kubernetes can collect. By default, `kubectl logs` shows these. For a production setup, use a log aggregator:

- **EFK stack (Elasticsearch, Fluentd, Kibana):** A common combination where Fluentd (or Fluent Bit) runs on each node to tail container log files and send to Elasticsearch, and Kibana provides a UI to search logs. Many cloud Kubernetes services offer integrated logging (e.g., GCP's Stackdriver, AWS CloudWatch, Azure Monitor) that can capture these logs without you managing the infra.

- **Loki (by Grafana) + Promtail + Grafana:** A lighter-weight logging solution that stores logs in a time-series database.

By aggregating logs, you can create alerts for certain log patterns (e.g., an application output "ERROR" or "Unauthorized access attempt" could alert the dev team or sec team).

Kubernetes Audit Logs: If enabled, every request to the API server can be logged. You might configure these to be sent to a secure store (like an S3 bucket or Elasticsearch). Audit logs are crucial to detect things like "was there an attempt to create a ClusterRoleBinding giving cluster-admin to an unexpected user?" or "who deleted resource X?". They can be noisy, so tune the audit policy to focus on sensitive actions (like write requests, or specifically auth failures etc.). Tie this into your SIEM (Security Information & Event Management) system if you have one, so alerts can be generated on suspicious API calls.

Metrics Monitoring: Use **Prometheus** to gather metrics from Kubernetes and applications. Prometheus can scrape the Kubernetes metrics (via components like the metrics-server or kube-state-metrics for resource stats). Unusual metrics can indicate trouble:

- A sudden spike in outbound network traffic from a pod that usually stays quiet could indicate data exfiltration.
- Continuous high CPU usage in a pod that should be idle might indicate crypto mining or a hung process.
- Prometheus Alertmanager can be configured to alert on such conditions (e.g., if CPU > 90% for 5 minutes on a database pod unexpectedly, notify).

Event Monitoring: Kubernetes events (viewable via `kubectl get events`) provide info on things like pods failing, images not pulling, etc. Some security-relevant events might include frequent container restarts (could be a crash loop from an attack or probe), scheduling failures (if someone tried to run a privileged pod but was denied). Tools like Prometheus with Alertmanager or even custom controllers can watch events for patterns.

Runtime Threat Detection with Falco

Preventive controls reduce risk, but we also want to catch an attacker in the act if they manage to bypass prevention. **Falco** is a CNCF project (a runtime security tool) that monitors system calls on a node and applies a set of rules to detect abnormal behavior. Think of it as an IDS (Intrusion Detection System) specifically tuned for container and cloud-native environments.

What can Falco detect? For example:

- A shell or exec into a container (which might indicate someone got inside or a dev is debugging).
- Reading of sensitive files within a container (Falco has rules for files like `/etc/passwd`, `/etc/shadow`, certain token paths).
- Unexpected network connections (e.g., a container opening a port it usually doesn't, or connecting to an IP on a blacklisted range).

- Changes to configuration files or binaries in containers or on the host.
- Kernel module loading (in case someone tries to load a rootkit via a container).
- Privilege escalation attempts (like calling setns, or creating device files).

Falco works by running a daemon on each node (with the ability to tap into the kernel's system call stream via eBPF or a kernel module) and uses a rules file to filter events. When a rule is triggered, Falco can log it and also send it out (to stdout, a file, or via something like Falcosidekick to various endpoints).

Falcosidekick: This is a side tool that can take Falco alerts and forward them to many outputs: Slack, Teams, Elasticsearch, etc., which is useful for integrating into your alerting workflows.

Other Tools:

- **Tracee (by Aqua Security):** Another eBPF-based detector similar to Falco.
- **Kube-hunter:** Not a runtime detector, but an active scanner you can run (or have in CI) to find common misconfigurations. It's like a pen-test tool for K8s (e.g., it will see if your etcd is openly accessible, or if the dashboard is exposed).
- **GuardDuty for EKS (AWS)** or other cloud-specific anomaly detection: Cloud providers have their own services that analyze cloud trail logs and network traffic for suspicious patterns (like an EC2 instance (node) making odd DNS queries, etc).

Tying it Together with Compliance

- **NIST CSF Detect function:** All the monitoring and Falco stuff falls under the Detect (DE) function of the NIST Cybersecurity Framework. Having these capabilities helps meet controls about continuous security monitoring.
- **PCI Requirement 10:** If you're under PCI, there's a big emphasis on logging and monitoring all access to cardholder data and detecting anomalies. Using Kubernetes audit logs and Falco to monitor access attempts to sensitive data in containers would support those requirements.
- **NSA/CISA Hardening:** The guide suggests enabling audit logging and leveraging threat detection tools, though it doesn't name Falco, it aligns with that idea of behavioral monitoring.
- **MITRE ATT&CK for Containers:** Monitoring with Falco can help detect many of the techniques in the MITRE ATT&CK container matrix (like Tactics for Execution, Persistence, etc.). For instance, the "Execution of shell" technique or "reading /etc/passwd" technique would be flagged by Falco rules.

Conclusion

By leveraging the MITRE ATT&CK framework and Elevation's layered security controls, Fastly has built a Kubernetes security strategy that is proactive, robust, and adaptive. We've combined preventative, detective, and responsive controls to ensure threats are identified and neutralized at every stage, keeping our Kubernetes environments resilient and secure.