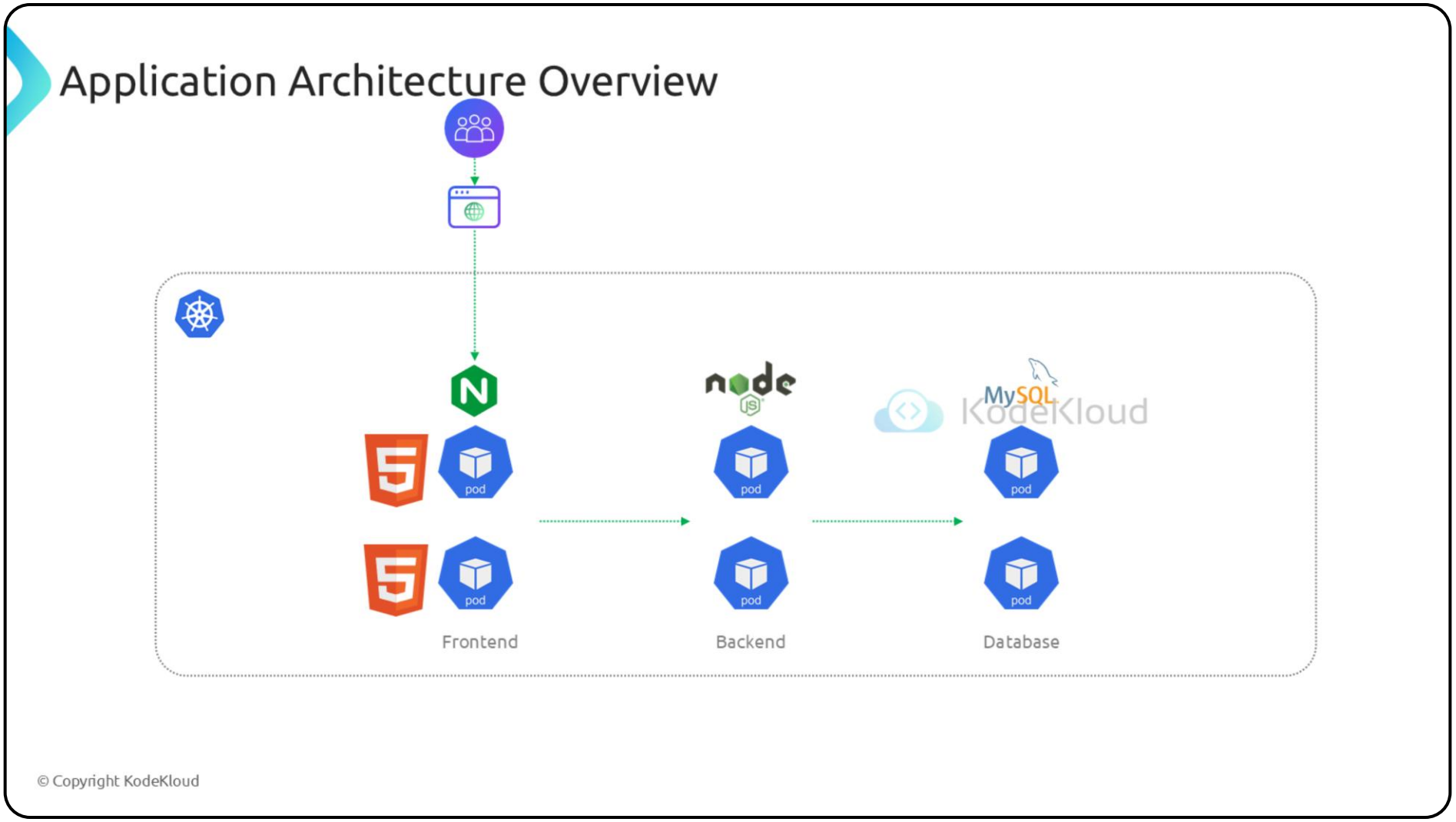
Visit www.kodekloud.com to learn more.

Kubernetes Trust Boundaries and Data Flow



Introduction

Let's start by looking at what is required to secure a multi-tier web application running in a Kubernetes (K8s) environment. This application consists of several key components: A front end, backend and database

First the Frontend, An Nginx web server that serves static content like HTML, CSS, and JavaScript files. It also acts as a reverse proxy, meaning it forwards client requests to the appropriate backend services. In our Kubernetes cluster, this Nginx server

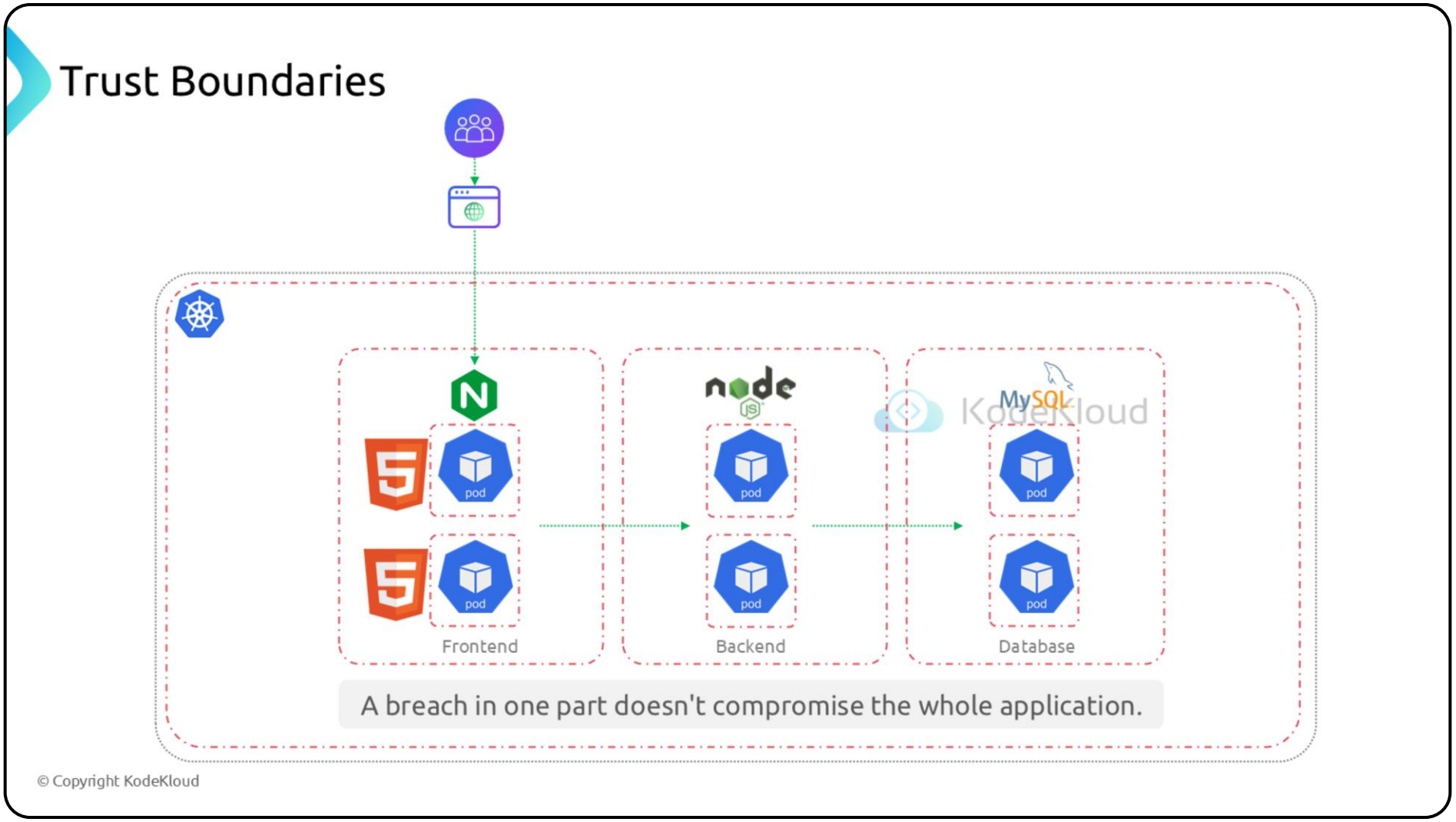
runs in its own pod within the frontend namespace.

And then we have the Backend API, A set of Node.js microservices that handle the business logic of the application, processing requests, and interacting with the database. These microservices run in separate pods within the backend namespace.

Finally the Database, A MySQL database that stores user data and application state. This database runs in a pod within the database namespace.

Threat Modeling

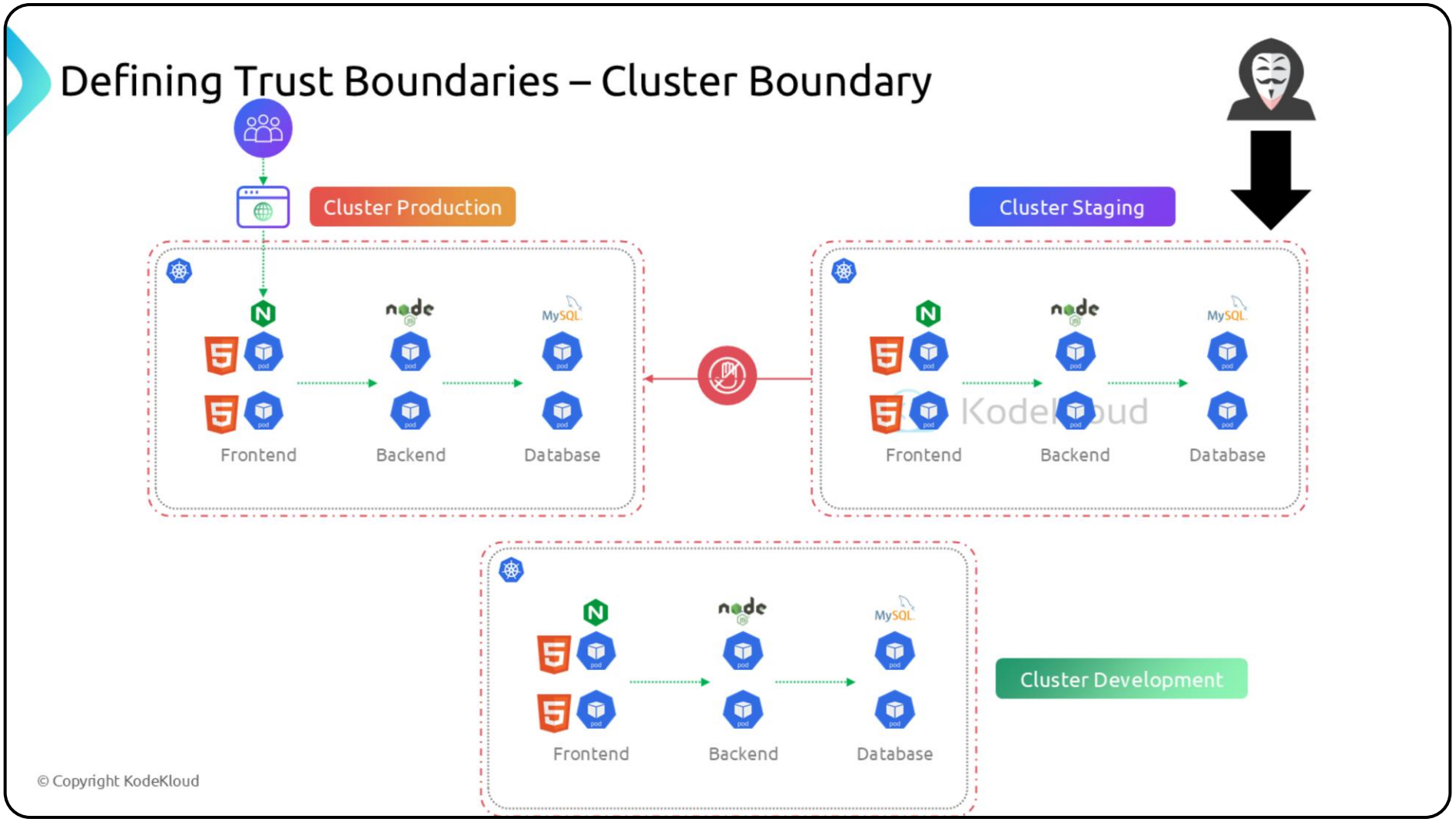
Threat Modeling is a process that helps you find potential threats, understand their impact, and put measures in place to stop them. By using threat modeling, we can better protect our application from various security threats.



The Need for Boundaries in Our Example

To effectively secure our multi-tier application, we need to isolate different parts of the system and enforce specific security measures for each part. This isolation helps us manage and reduce security risks by ensuring that a breach in one part of the system doesn’t compromise the entire application.

We call these isolated areas "trust boundaries." By defining and securing these boundaries, we can better protect each part of our application.



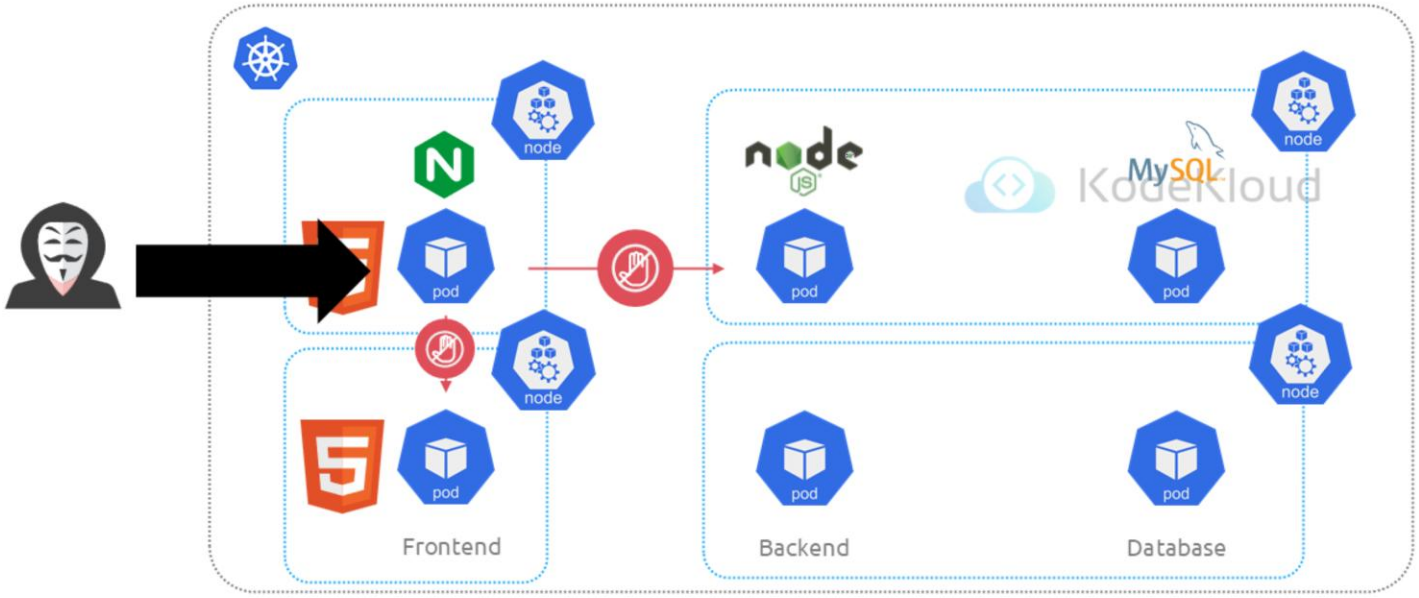
Starting with Step 1: Defining the Trust Boundaries

First we need to identify the Cluster Boundary.

The entire Kubernetes setup, including nodes and control-plane components, forms the cluster boundary. By using separate clusters for different environments – in this case development, staging, and production environments, we ensure that issues in one environment don't affect others. Most importantly, this provides top-level network isolation, meaning that network traffic

is isolated at the cluster level, preventing any potential cross-environment contamination.

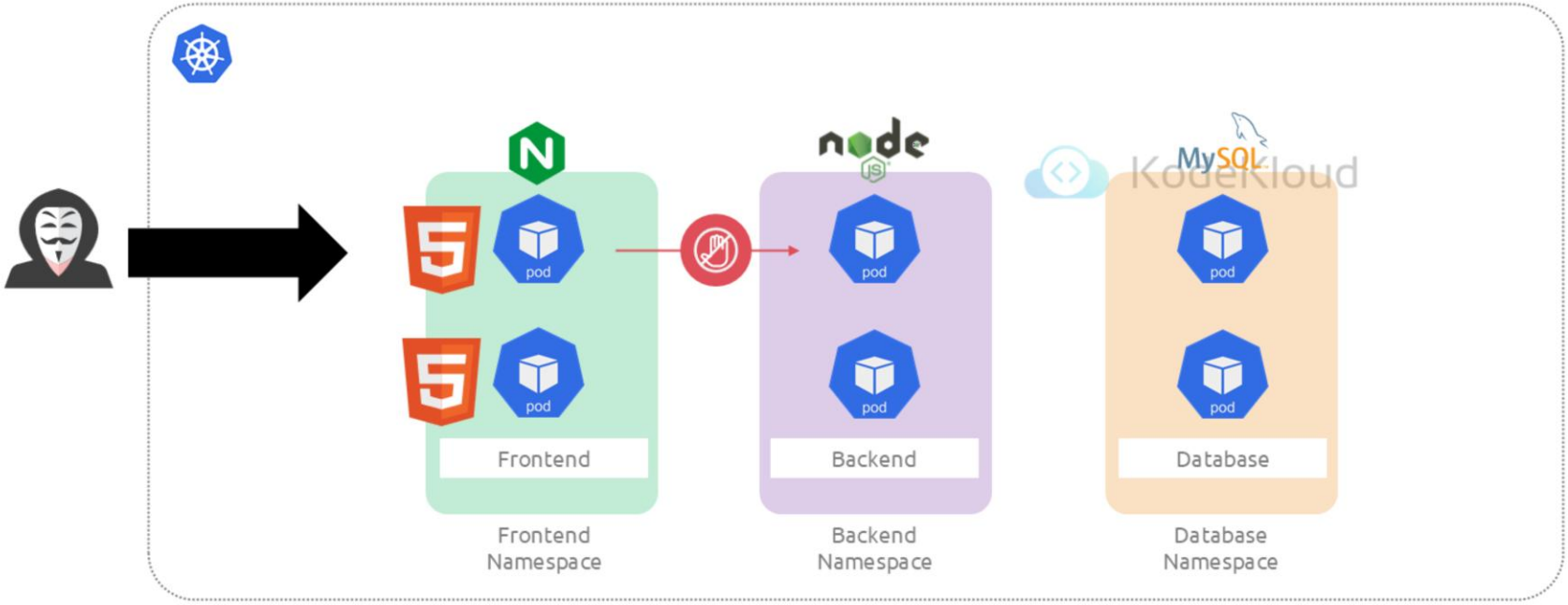
Defining Trust Boundaries – Node Boundary



Node Boundary: Each node, whether virtual or physical, acts as a trust boundary in the cluster. Nodes host multiple pods and system components such as the kubelet and should only access the resources required to perform their tasks.

For example, If an attacker compromises a node running frontend services, they cannot directly access backend services, reducing the risk of spreading the attack.

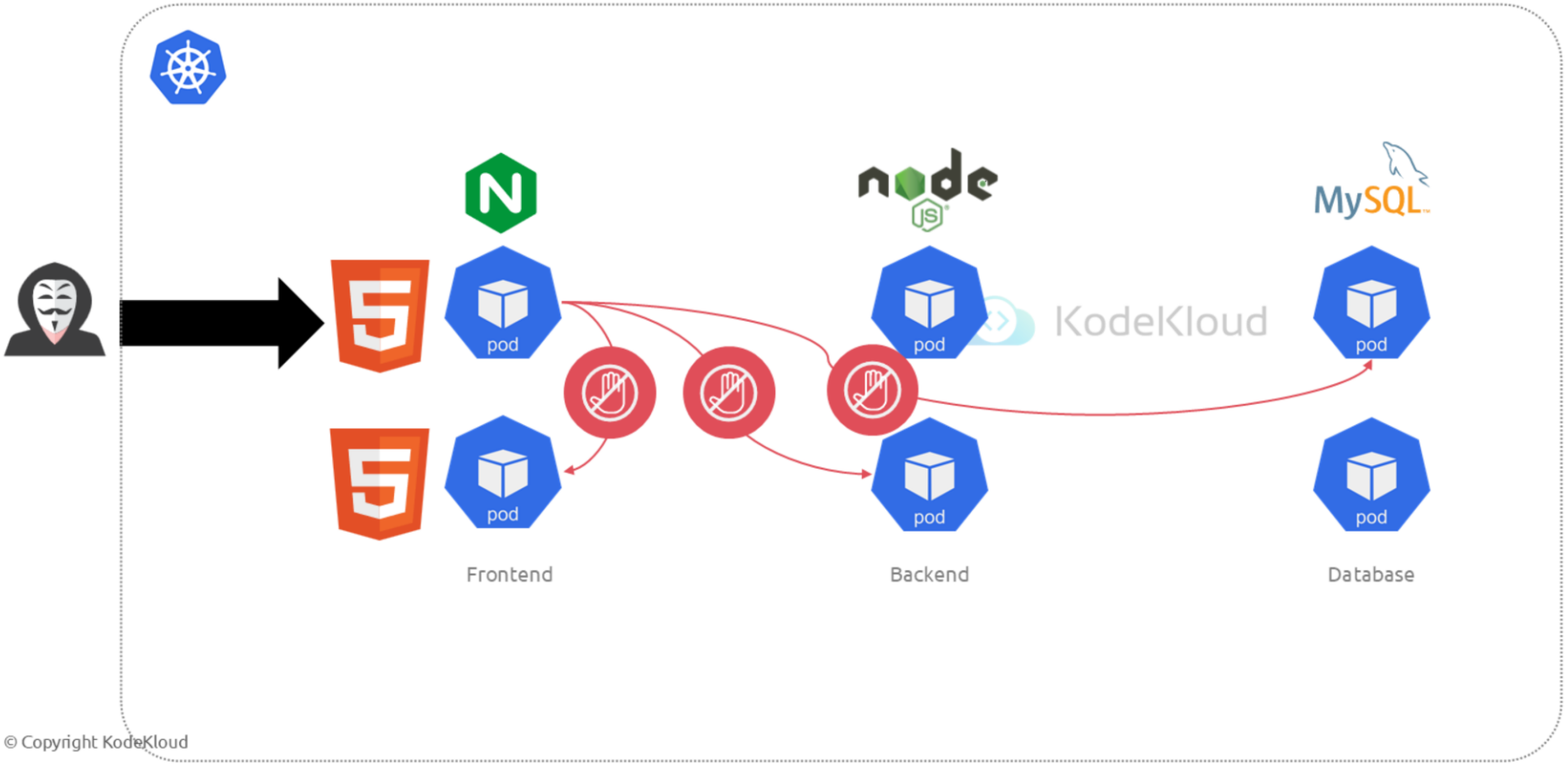
Defining Trust Boundaries – Namespace Boundary



Namespace Boundary: Within our cluster, we create namespaces to group related resources. We have namespaces for the frontend, backend, and database. This helps manage resource access and isolation. They serve as the basic unit for authorization in Kubernetes.

Example: By isolating the frontend, backend, and database components into separate namespaces, we can control access specifically for each component, preventing unauthorized access.

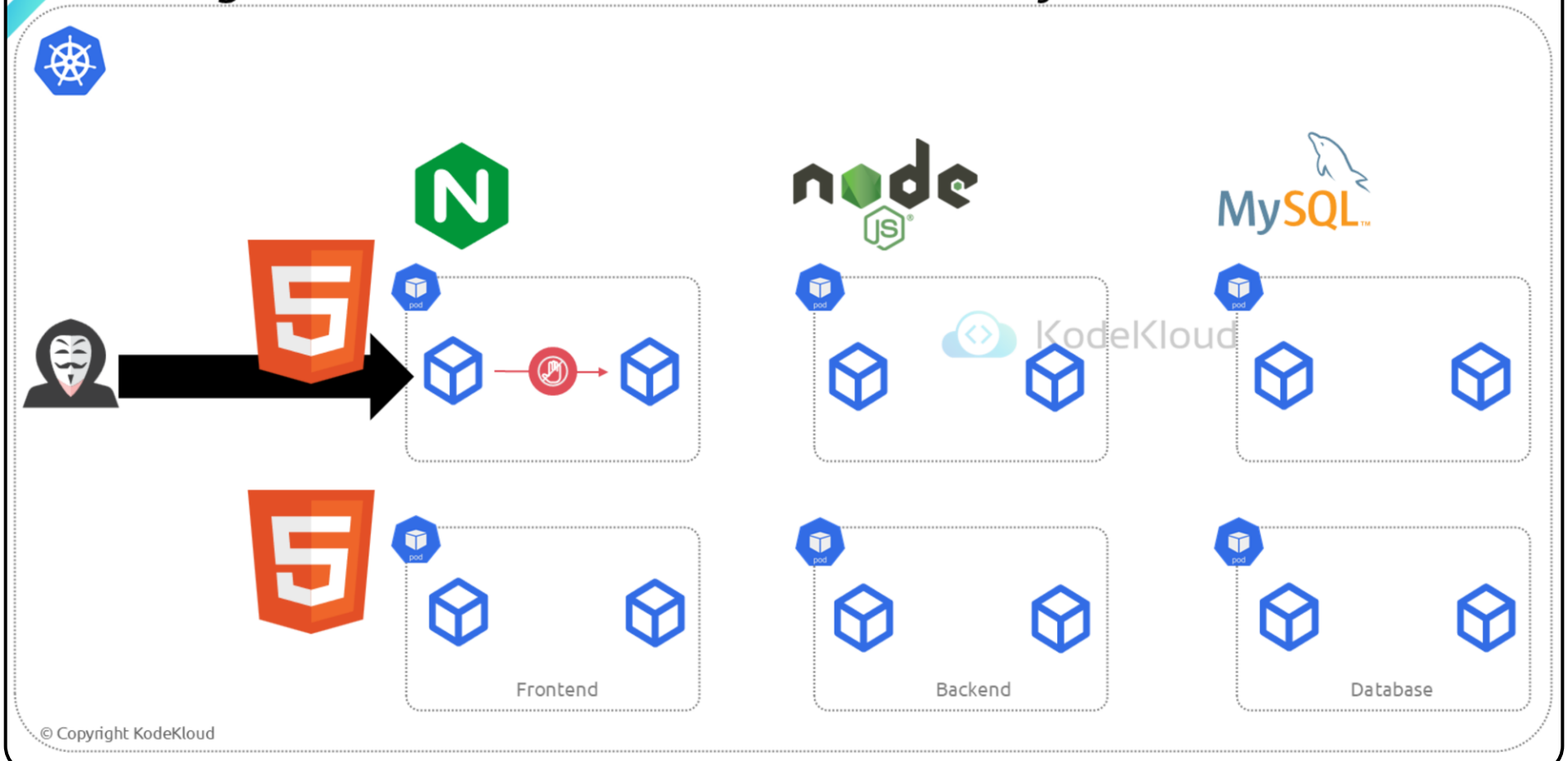
Defining Trust Boundaries – Pod Boundary



Pod Boundary: Each application component runs in its own pod. For instance, the Nginx server runs in a frontend pod, backend API services run in backend pods, and the MySQL database runs in a database pod. This ensures that each component is isolated within its own runtime environment and security contexts and network-level isolation can be defined at the pod level.

Example: If a backend pod is compromised, the attacker cannot directly interact with the frontend or database pods without additional security breaches.

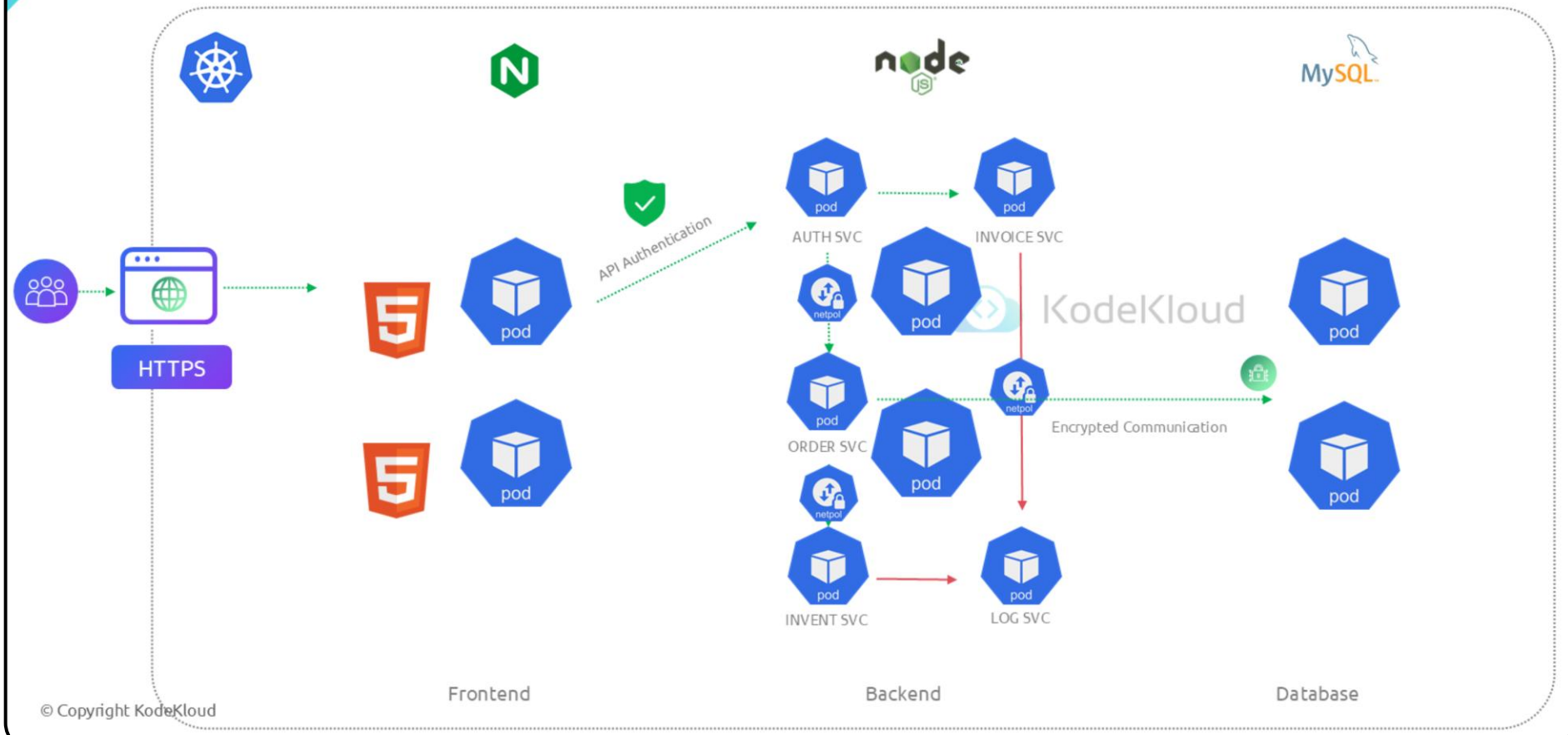
Defining Trust Boundaries – Container Boundary



Container Boundary: Inside each pod, containers run the actual services. For example, an Nginx container within the frontend pod, Node.js containers within the backend pods, and a MySQL container within the database pod. Containers are the smallest unit of our trust boundaries, providing application-level isolation.

Example: This means that even if one container within a pod is compromised, the damage is limited to that container, providing another layer of isolation.

Exploring Data Flow in a Multi-Tier Application



Now that we've defined our trust boundaries, it's important to understand how data flows through our application. Data flow is crucial because it helps us understand how information moves through our system. By mapping out this flow, we can identify potential security risks at each stage and apply appropriate controls.

Let’s look at our multi-tier application to see how data flows through it…

Let's expand on the backend services as there are mutiple microservices application configured. Here we have an auth service,

invoice service, order service , inventory service and log service.

First, users access the application via the Nginx web server in the frontend. This entry point must be secure with measures like HTTPS and authentication to protect user data right from the start.

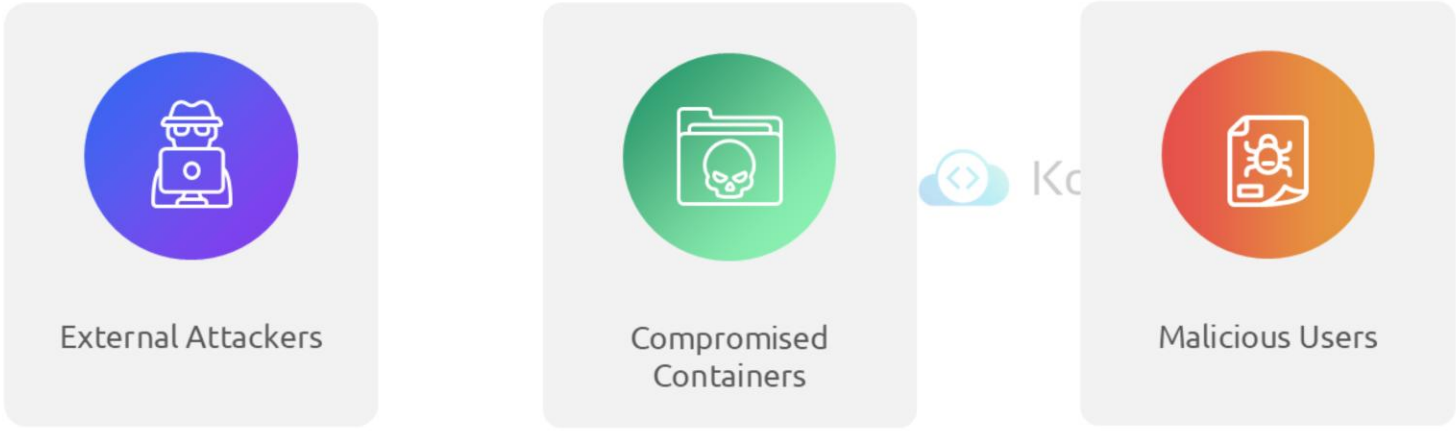
Next, the Nginx server forwards the user requests to the backend API services. It’s essential to have secure communication channels and proper API authentication here to ensure that only legitimate requests are processed.

The backend services then need to access the database to fetch or store user data. This interaction with the MySQL database must be tightly controlled and encrypted to prevent unauthorized access and protect sensitive information.

Finally, backend services often need to communicate with each other to fulfill the user’s request. Network Policies in Kubernetes can help restrict which pods can communicate with each other, enhancing security by preventing unauthorized inter-pod communication.

By understanding this data flow, we can pinpoint where security measures are needed and implement them effectively.

Identifying and Mitigating Threats



Having defined the trust boundaries and analyzed the data flow in our Kubernetes environment, the next step is to understand the various types of threats our application might face and how to mitigate them. Threat actors are entities that pose threats to our system, such as external attackers, compromised containers, or malicious users. By identifying these threat actors, we can implement security measures to protect our application.

Summary

01 Use trust boundaries to isolate and secure application components

02 Analyze data flow to identify potential security risks

03 Implement Network Policies and RBAC for access control

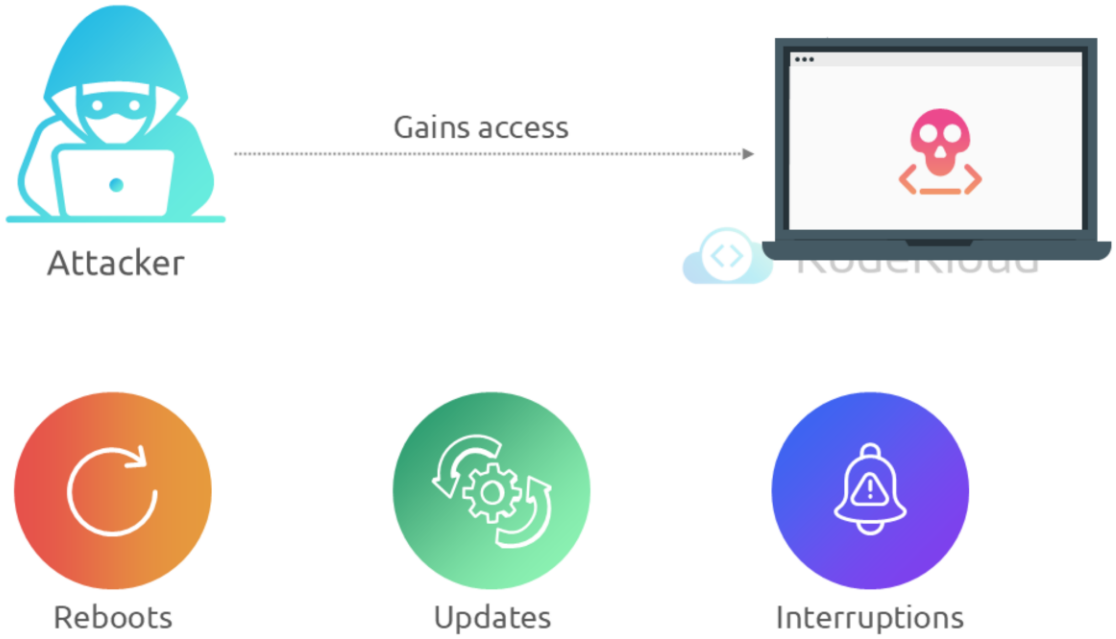
04 Apply encryption and authentication to protect data in transit

05 Conduct threat modeling to identify and mitigate security threats

Introduction

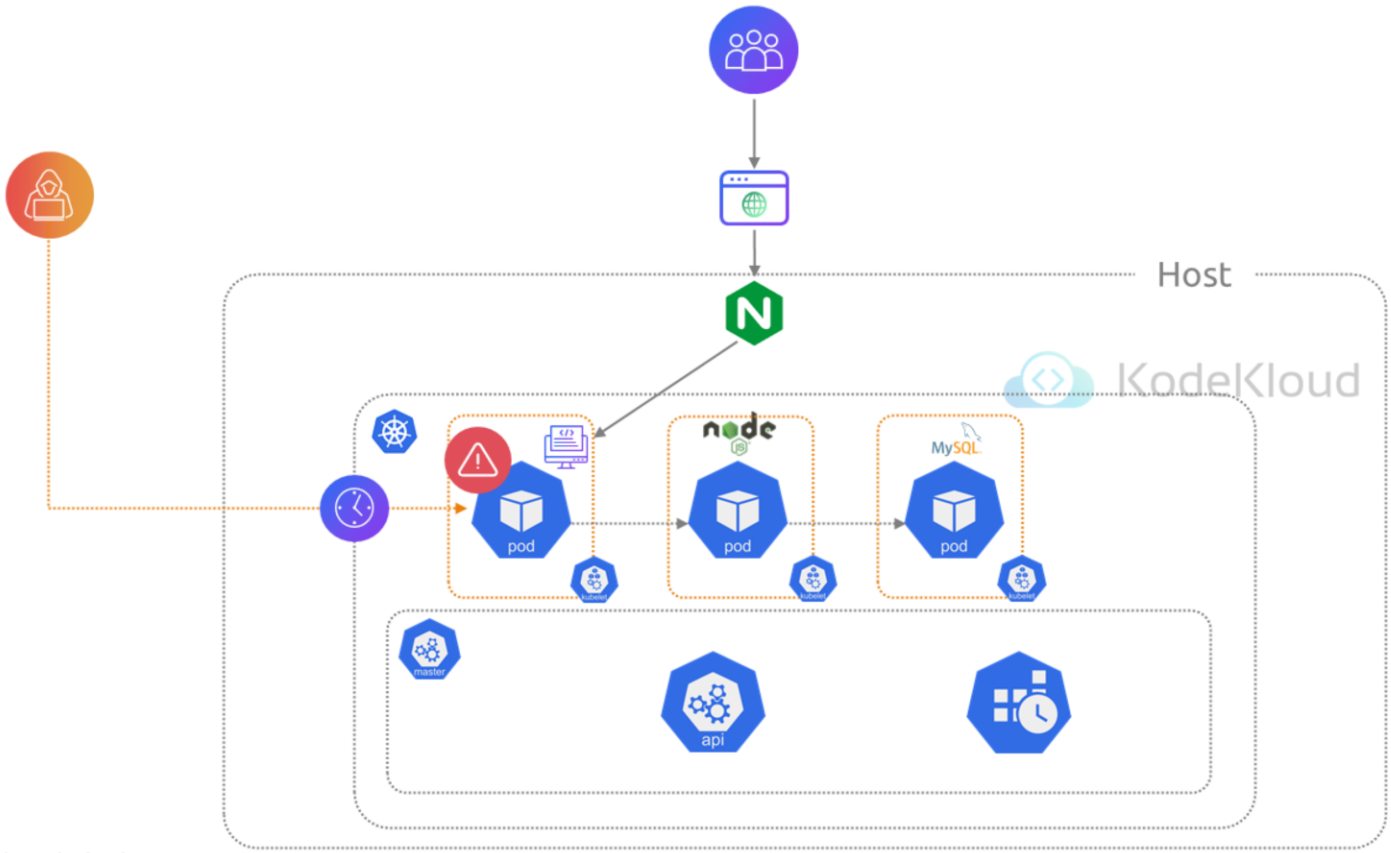
In this lesson, we will explore the concept of how an attacker establishes persistence in Kubernetes and gain a foothold in your cluster for extended periods.

Understanding Persistence in Kubernetes



Persistence refers to the ability of an attacker to maintain access to a compromised system even after reboots, updates, or other interruptions. In a Kubernetes cluster, this can be achieved through various methods, such as exploiting misconfigurations, reading secrets, or leveraging container vulnerabilities.

Scenario: Gaining Persistence in Our Application

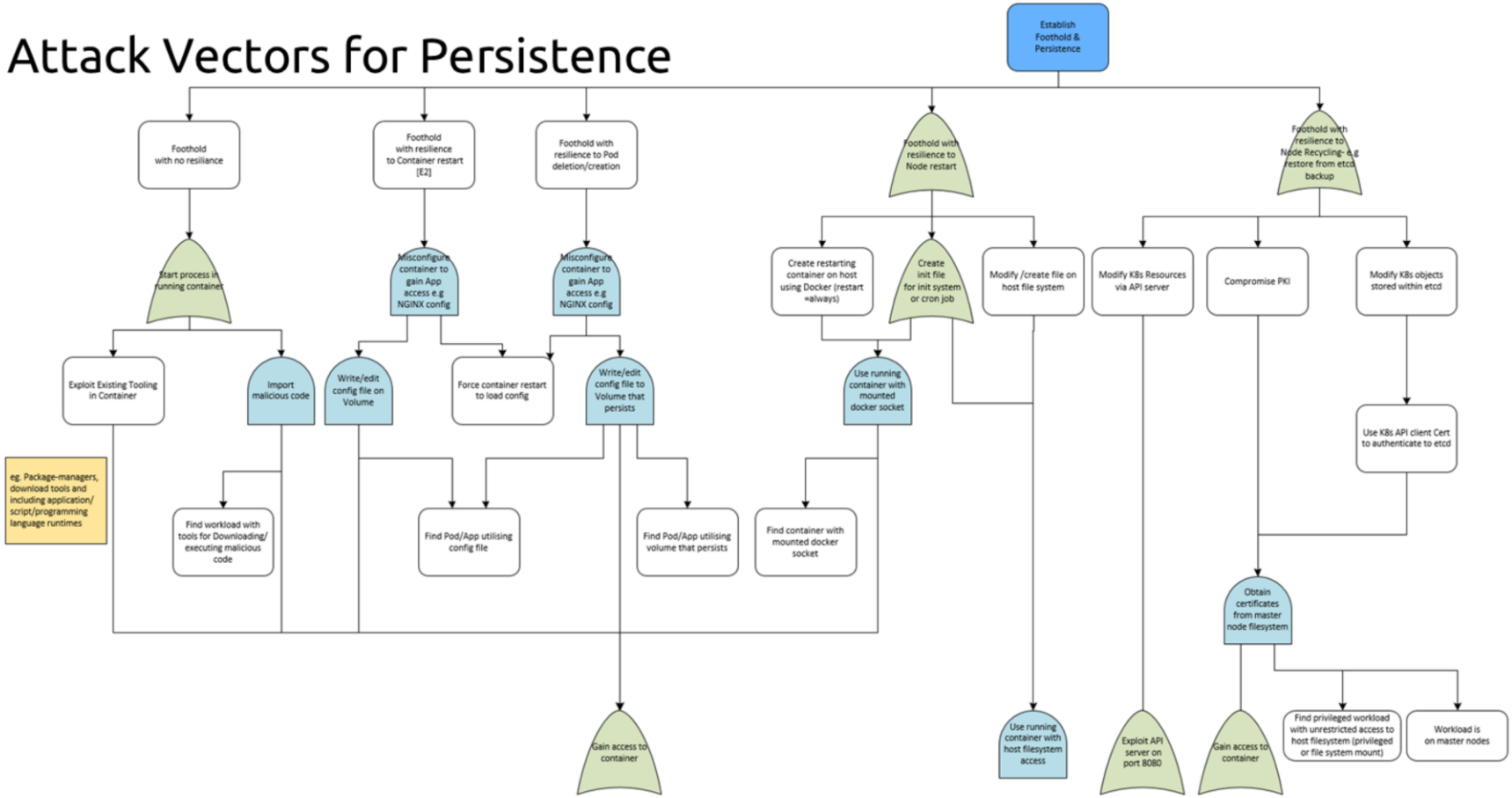


Scenario: Gaining Persistence in Our Application

Let's consider our multi-tier web application with its frontend (Nginx), backend (Node.js microservices), and database (MySQL) components. An attacker might gain initial access through a compromised application running in a container.

Once inside, their goal is to establish persistence, allowing them to maintain control over the cluster despite efforts to remove their access.

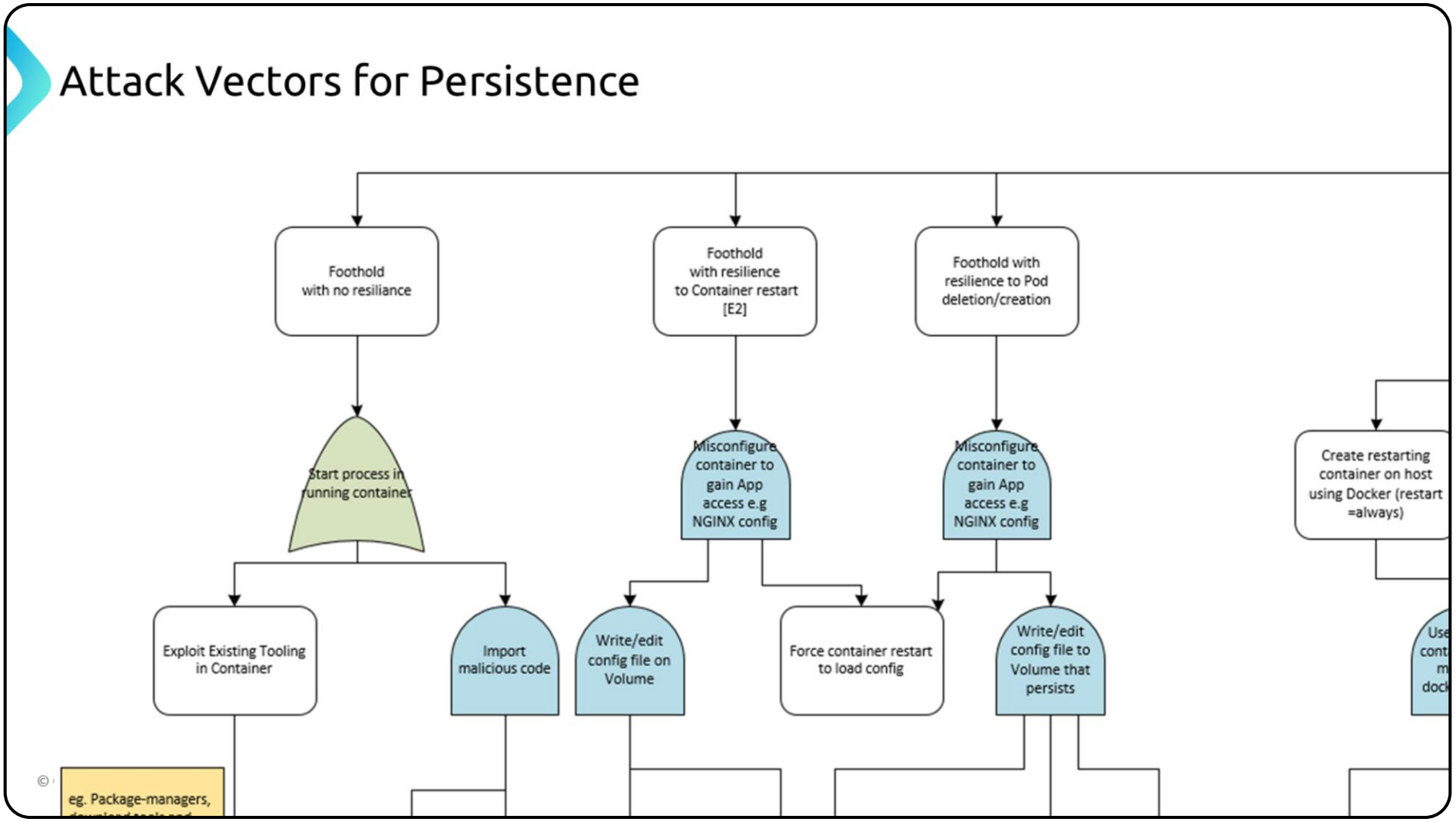
Attack Vectors for Persistence



Source: https://github.com/cncf/financial-user-group/blob/main/projects/k8s-threat-model/AttackTrees/EstablishPersistence.md

The CNCF Financial User Group has provided an attack tree to illustrate how attackers might achieve persistence in a cluster. The aim of this tree is to discover the several ways an attacker can attempt to gain persistence in the cluster with differing periods of longevity.

Attack Vectors for Persistence



Let's go over a brief of the top level areas. First Node - Foothold with No Resilience:

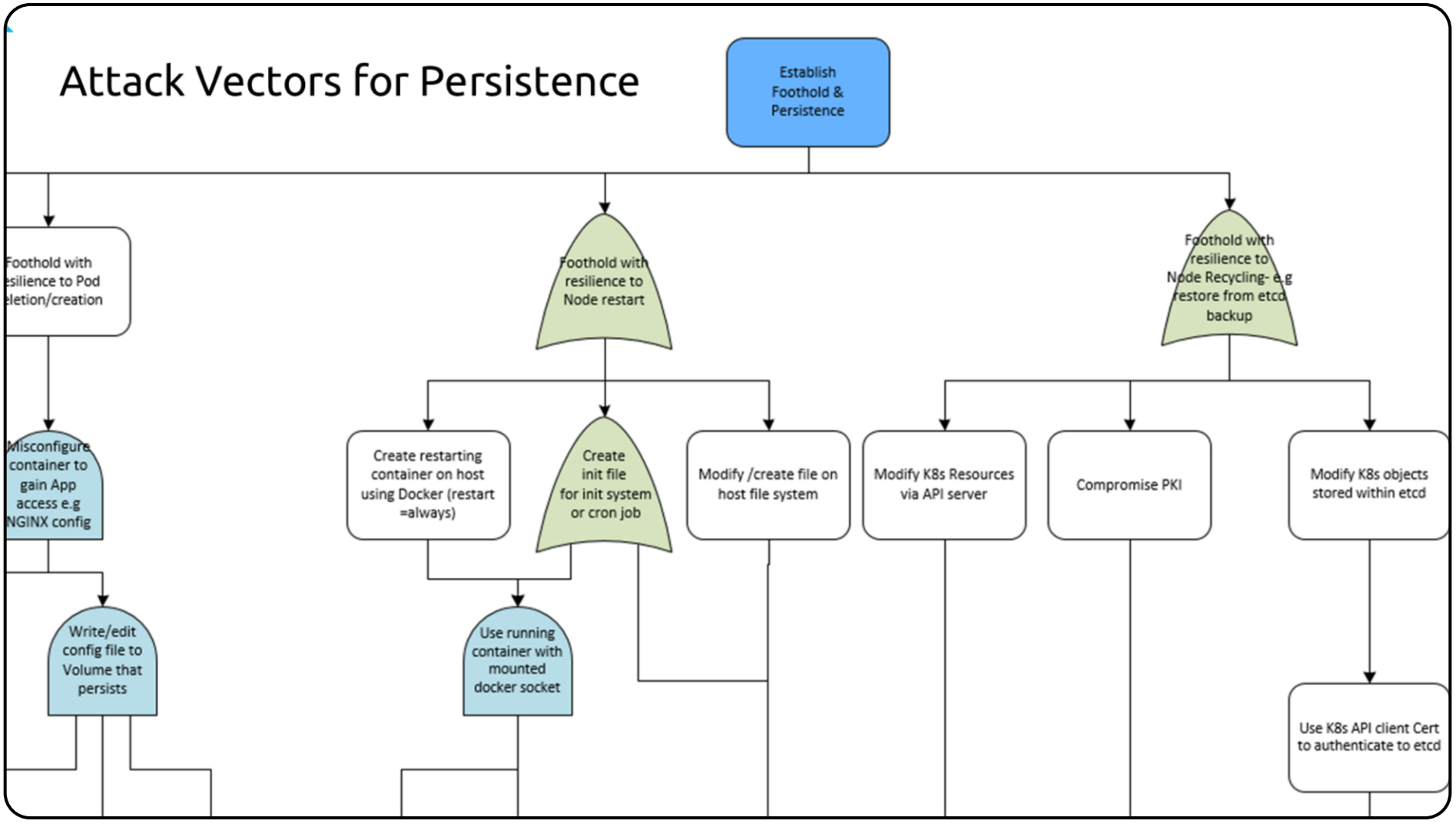
•This is the most basic form of access, where an attacker gains a foothold without implementing any resilience strategies. If the system or environment experiences a minor disruption, like a container restart or pod deletion, this foothold is likely lost.

Foothold with Resilience to Container Restart (E2):

•Here, the attacker's foothold is more durable. It survives basic container restarts, ensuring that even if a container restarts, they retain access. This resilience is crucial for maintaining an ongoing presence within the container.

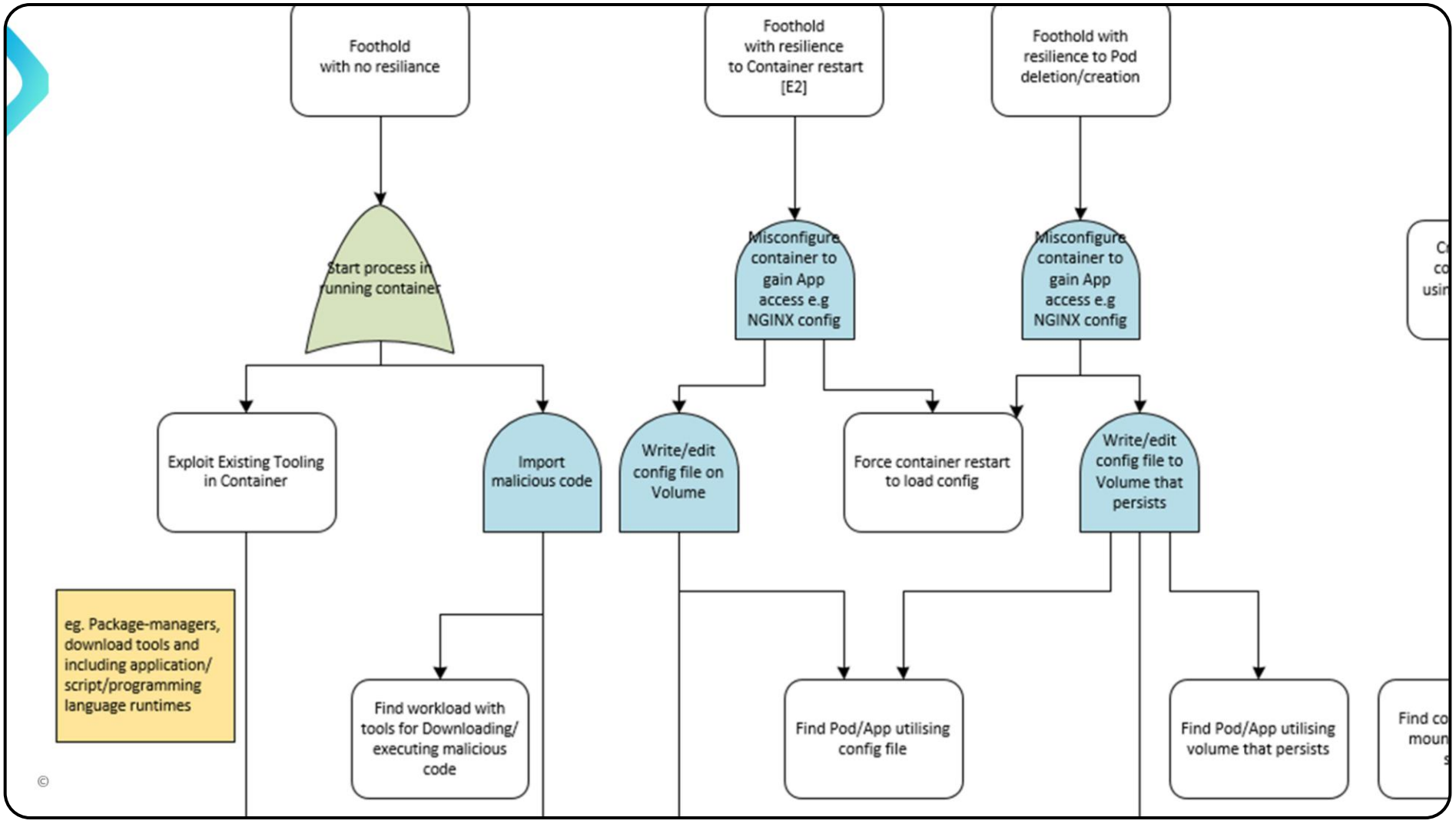
Foothold with Resilience to Pod Deletion/Creation:

•At this level, the attacker’s foothold withstands actions like pod deletion or recreation. This means they have likely embedded themselves in the system or have ways to re-establish themselves automatically, even if the pod is deleted and recreated.



Let's go over a brief of the top level areas. First Node - Foothold with No Resilience:
•This is the most basic form of access, where an attacker gains a foothold without implementing any resilience strategies. If the system or environment experiences a minor disruption, like a container restart or pod deletion, this foothold is likely lost.
Foothold with Resilience to Container Restart (E2):
•Here, the attacker's foothold is more durable. It survives basic container restarts, ensuring that even if a container restarts, they retain access. This resilience is crucial for maintaining an ongoing presence within the container.
Foothold with Resilience to Pod Deletion/Creation:

•At this level, the attacker’s foothold withstands actions like pod deletion or recreation. This means they have likely embedded themselves in the system or have ways to re-establish themselves automatically, even if the pod is deleted and recreated.



Foothold with No Resilience:

•This represents the initial and basic access point for an attacker within a Kubernetes environment. It's a foothold that lacks any sort of persistence or resilience. If the container or pod were to restart or be recreated, this foothold would be lost, making it a transient and weak access point.

Start Process in Running Container:

•After gaining an initial foothold, the attacker starts a new process in the running container. This is often the first step in launching malicious activities from within the environment. By running processes within an existing container, attackers avoid

the need to deploy new workloads, making detection slightly more challenging.

Exploit Existing Tooling in Container:

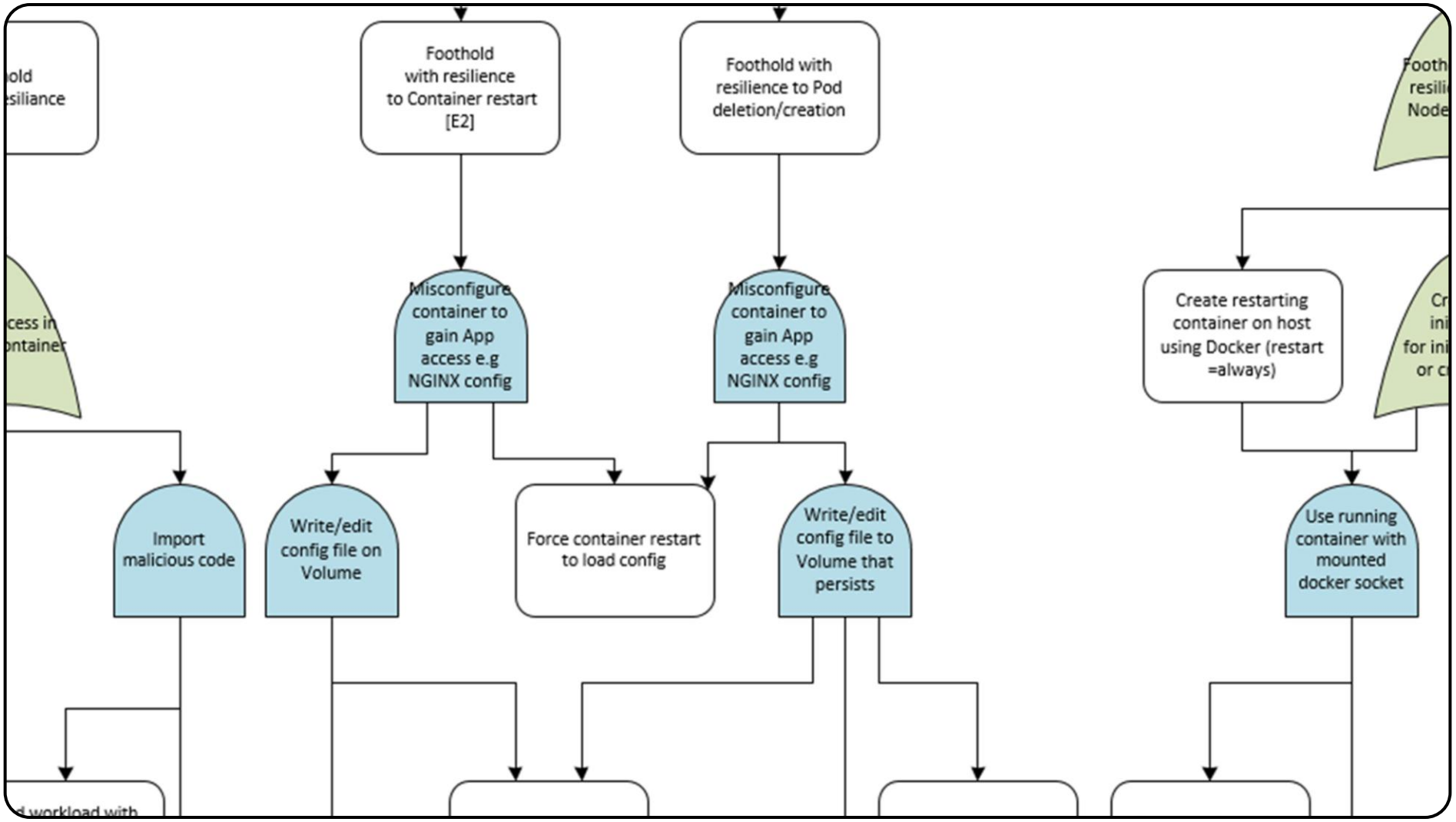
•With access to the container, attackers exploit built-in tools already present. Containers often come with tools like package managers (e.g., apt, yum, or pip), scripting language runtimes (e.g., Python, Node.js), and download utilities (e.g., curl, wget). By leveraging these, attackers can avoid introducing new binaries or files that might raise alarms, using existing tools to download or execute further malicious code.

Import Malicious Code:

•At this stage, the attacker imports or brings in their own malicious code into the container. This could be through downloading from an external source, copying into the container filesystem, or encoding it within scripts or configuration files. The code often includes functionalities that allow the attacker to escalate privileges, maintain persistence, or communicate with command-and-control servers.

Find Workload with Tools for Downloading/Executing Malicious Code:

•Attackers scan for workloads or containers that contain specific tools they need to download or execute malicious code. Not all containers have utilities like curl or wget, so the attacker searches for containers where these are available, enabling them to initiate or automate the download and execution of their code. This step increases the chances of finding a compatible container environment for their attack, making the exploitation process more efficient and flexible.



Foothold with Resilience to Container Restart (E2):

•At this level, the attacker has established a foothold that persists even if the container restarts. They may have modified system-level configurations, files, or environment variables to ensure their presence remains intact even after disruptions like restarts, making this foothold harder to remove.

Misconfigure Container to Gain App Access (e.g., NGINX Config):

•The attacker exploits configuration files within the container to gain broader access to applications or services. For instance, they might misconfigure an NGINX configuration to enable access to sensitive files or bypass authentication mechanisms. This

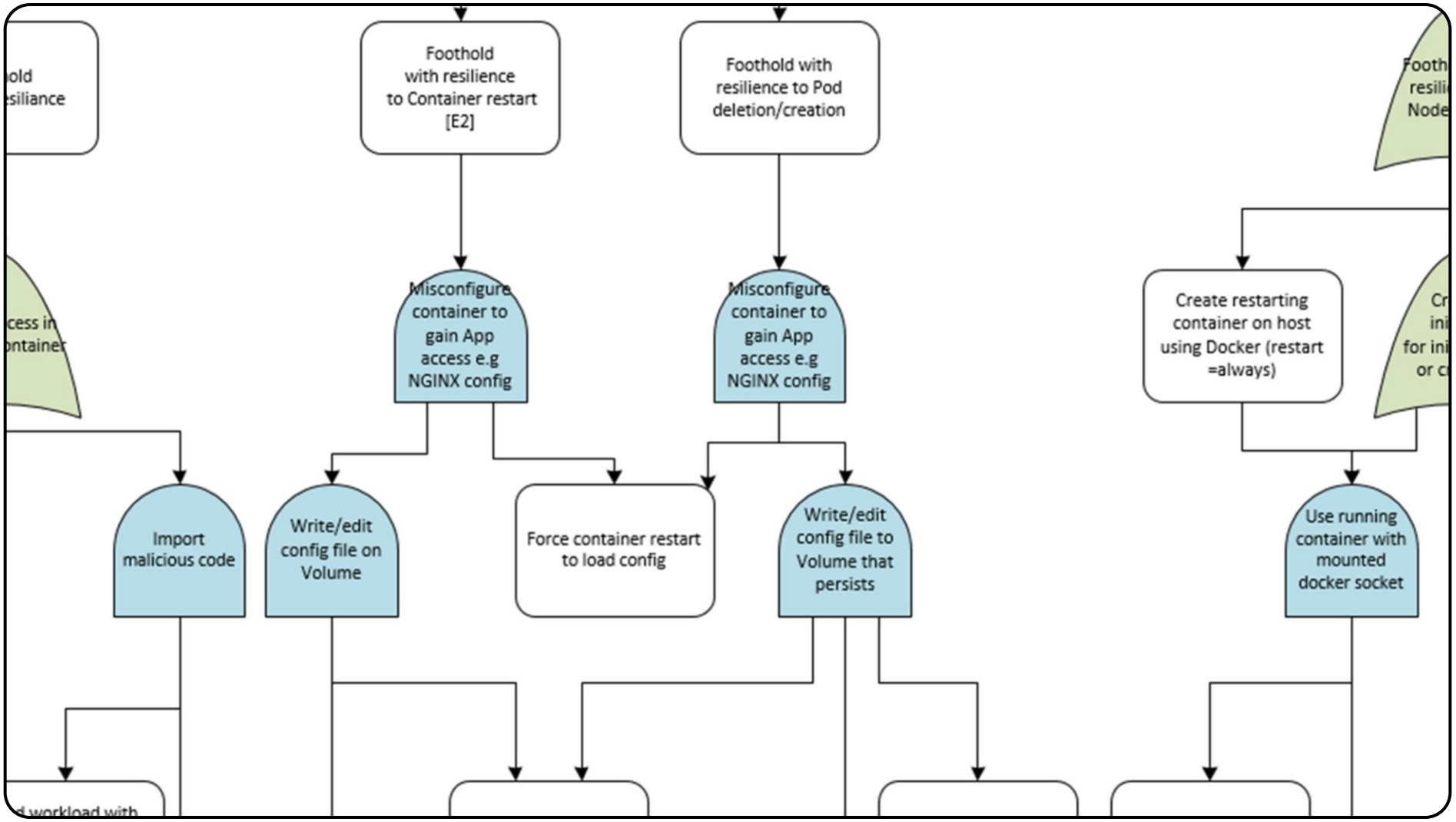
step involves manipulating the application settings to either lower security barriers or increase privileges within the container.

Write/Edit Config File on Volume:

•By writing or editing a configuration file stored on a volume, the attacker ensures that their changes are persistent across container restarts. Volumes are designed to store data outside of the container's lifecycle, so any configuration change made here will survive a restart, ensuring the foothold remains resilient.

Force Container Restart to Load Config:

•To apply their changes, the attacker may need to restart the container, forcing it to load the newly edited or maliciously modified configuration file. This step makes the attack persistent, as the misconfigured settings will now be in effect each time the container starts.



Foothold with Resilience to Pod Deletion/Creation:

•At this stage, the attacker's foothold is more persistent. Unlike container restarts, which may not affect a running pod, pod deletion and recreation often trigger a fresh deployment. The attacker has embedded their foothold deeply enough that even if the pod is deleted and recreated, their access or modifications remain intact. This level of persistence often requires writing data to a volume or configuration that is mounted across pod lifecycles.

Misconfigure Container to Gain App Access (e.g., NGINX Config):

•Similar to the previous step, the attacker misconfigures the container's application-level settings to increase their access. For

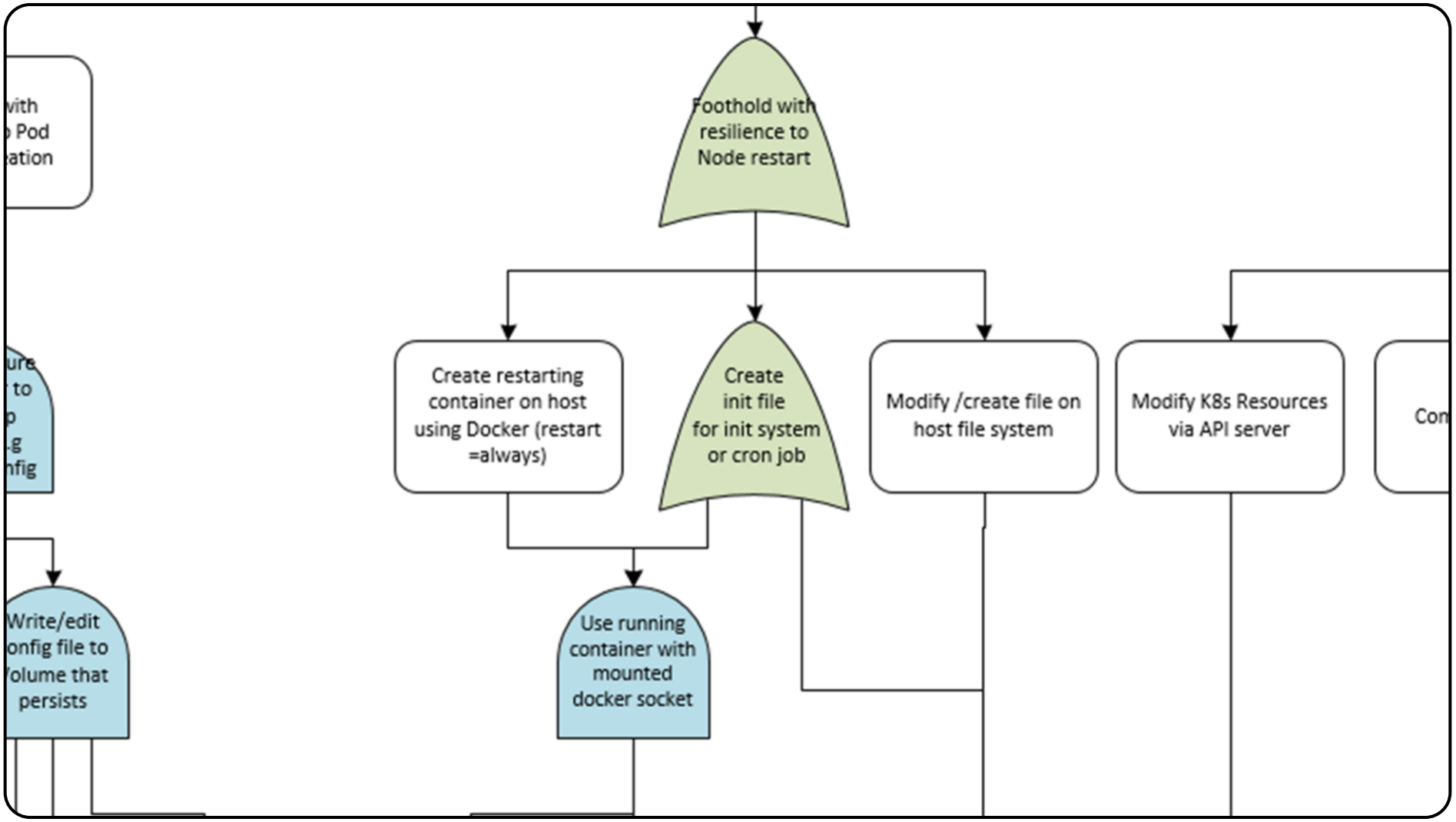
example, they might modify NGINX configuration files to allow unauthorized access to certain routes or disable security headers. By doing so, they can gain more control or bypass security checks within the application.

Write/Edit Config File to Volume that Persists:

•To maintain persistence, the attacker writes or edits a configuration file on a volume that is not dependent on the pod's lifecycle. This volume-based storage ensures that configuration changes will remain effective even if the pod is deleted and recreated. Writing configurations to a persistent volume means that whenever a new pod is created using this volume, it will load the attacker's modified configuration.

Force Container Restart to Apply Config (if required):

•If the attacker's modifications require a container restart to be applied, they may force a restart after editing the config. This step ensures that the modified settings are actively loaded into the application environment. This also means that each time a new pod is recreated and loads the persistent volume, it automatically adopts these configurations.



Foothold with Resilience to Node Restart:

•In this step, the attacker has established a foothold that remains even if the node is restarted. This level of persistence involves modifications at the node or host level, rather than within individual containers or pods. Such persistence ensures that access is maintained even after system-level reboots or reinitializations.

Create Restarting Container on Host Using Docker (restart=always):

•The attacker creates a container directly on the host system (not managed by Kubernetes) with the Docker restart=always policy. This configuration ensures that the container automatically restarts whenever the node or Docker daemon restarts,

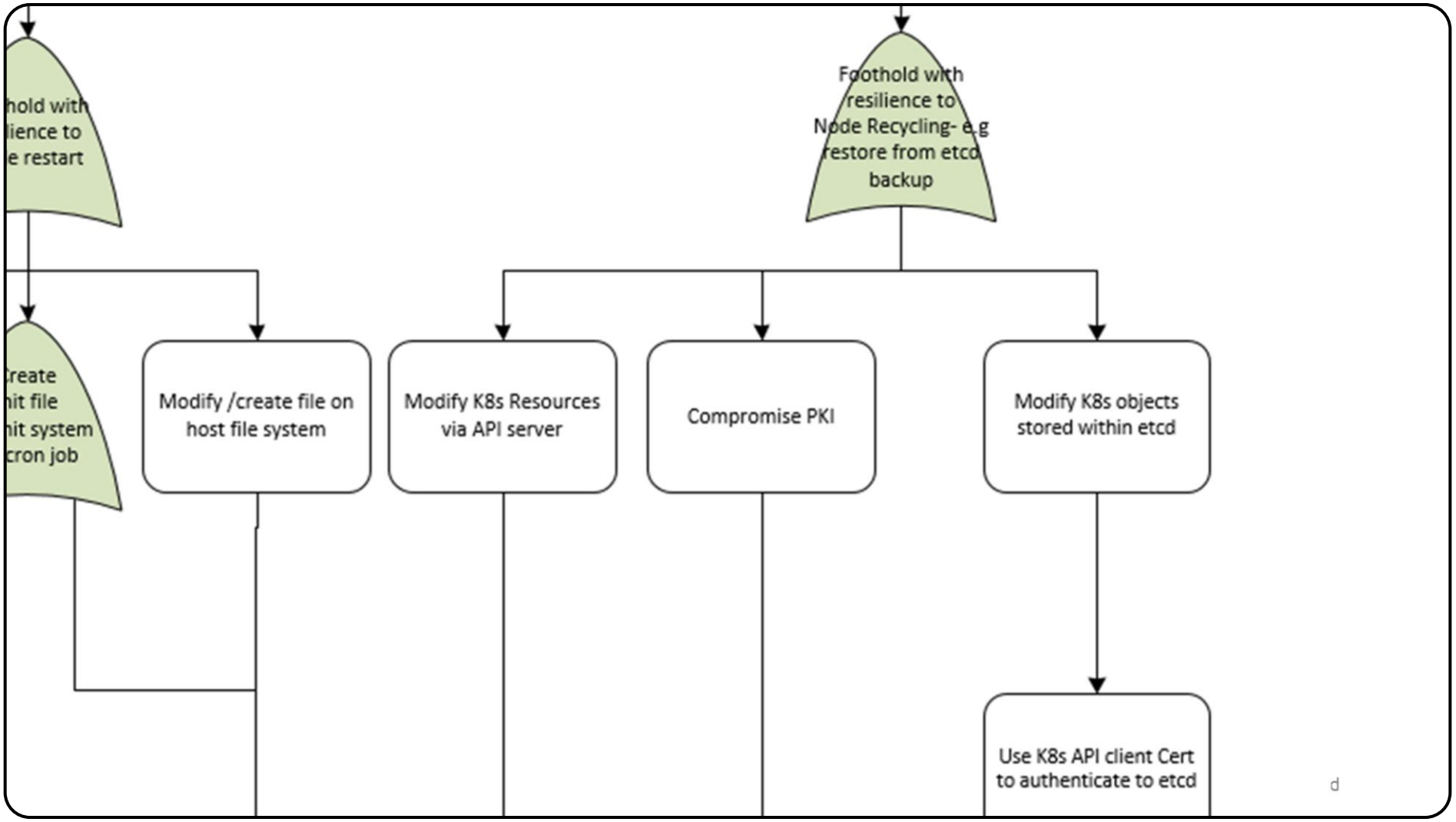
preserving the attacker’s access. This method bypasses Kubernetes’ pod management by directly using Docker on the host.

Create Init File for Init System or Cron Job:

•The attacker creates or modifies an init file (e.g., in /etc/init.d for init-based systems or systemd service files) or sets up a cron job on the host. These files are part of the host's boot process, so they automatically execute on node restart, allowing the attacker’s code to run persistently. This method embeds the attack within the system’s startup processes, making it resilient to node restarts.

Modify/Create File on Host File System:

•As another persistence method, the attacker modifies or creates files directly on the host’s file system. This could involve adding scripts or binaries that will execute automatically upon system startup, granting the attacker an entry point after each restart. Such files might be located in directories typically executed at boot, such as /etc/rc.local or /etc/cron.d, to ensure they run automatically.



Foothold with Resilience to Node Recycling (e.g., Restore from etcd Backup):

•At this stage, the attacker's foothold is resilient to node recycling, meaning they can maintain persistence even when a node is fully recycled and restored from an etcd backup. This indicates a deeper level of compromise, targeting Kubernetes' core infrastructure, such as etcd, which stores the entire cluster's configuration and state data.

Modify Kubernetes Resources via API Server:

•The attacker interacts with the Kubernetes API server to modify resources within the cluster. By gaining access to the API server, they can alter configuration settings, manipulate permissions, and create or delete resources, effectively taking control

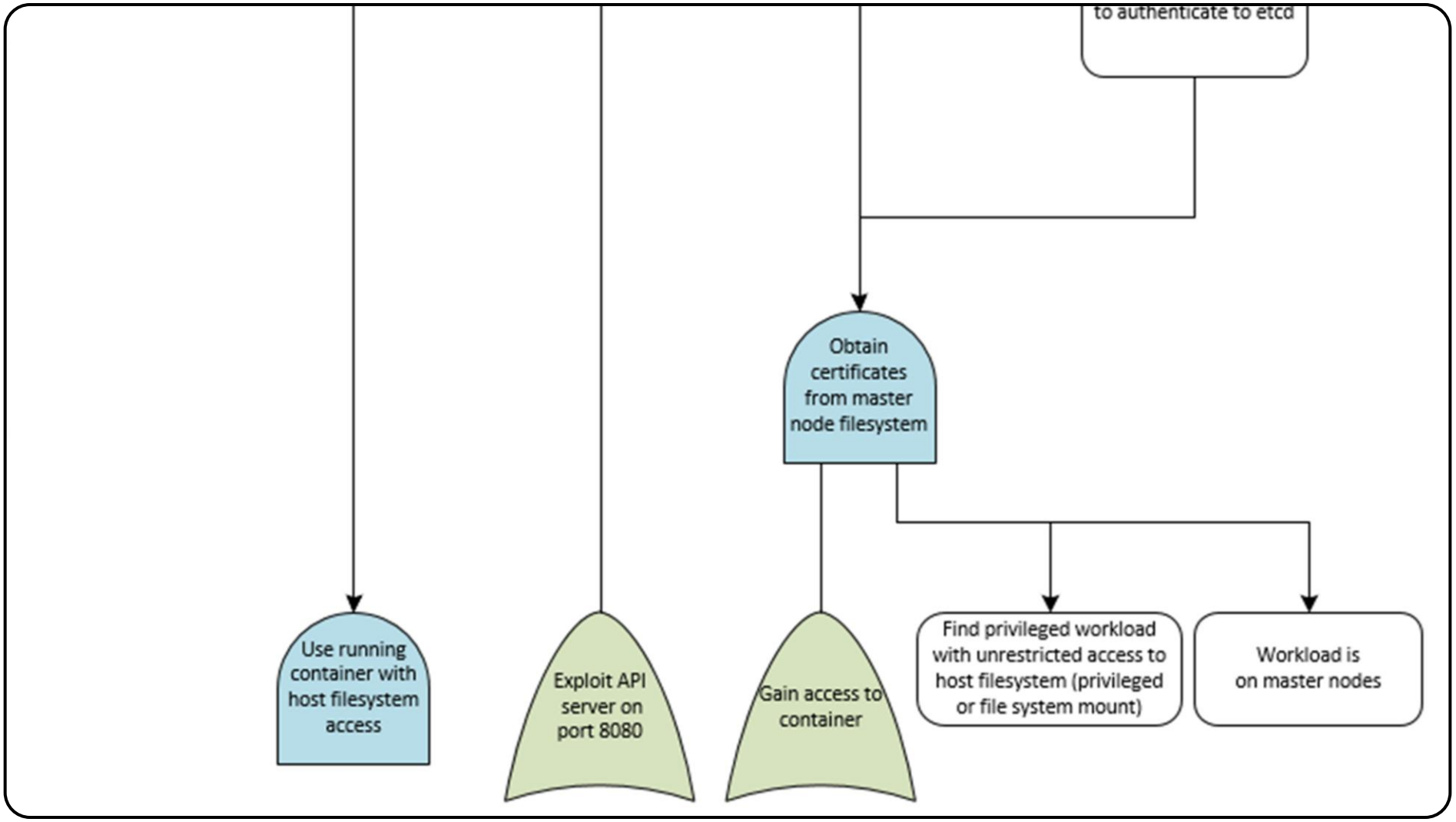
over parts of the cluster. This allows them to make changes that will persist across node recycling.

Compromise PKI (Public Key Infrastructure):

•The attacker targets the Kubernetes PKI, which manages the cluster's authentication and encryption keys. By compromising the PKI, they can create or use existing certificates and private keys to authenticate as trusted entities, enabling them to persistently access the cluster as an authenticated user.

Modify Kubernetes Objects Stored within etcd:

•The attacker directly manipulates Kubernetes objects stored in etcd. This allows them to change cluster configurations at the data storage level, ensuring that their modifications persist even if the cluster is restored from an etcd backup. By tampering with core data, they embed their foothold deeper into the system.



Obtain Certificates from Master Node Filesystem:

•The attacker targets the master node's filesystem to retrieve Kubernetes API client certificates. These certificates, if compromised, grant them authenticated access to critical components like etcd, allowing for deeper control over the cluster's state. With access to these certificates, attackers can authenticate as legitimate entities within the Kubernetes environment, making their actions harder to detect.

Gain Access to Host from Container:

•The attacker leverages their foothold within a container to gain access to the underlying host's filesystem. This typically

involves exploiting vulnerabilities in container configurations or privileges. By moving from the container to the host, they achieve a higher level of control over the system, enabling access to sensitive files, including certificates stored on the master node.

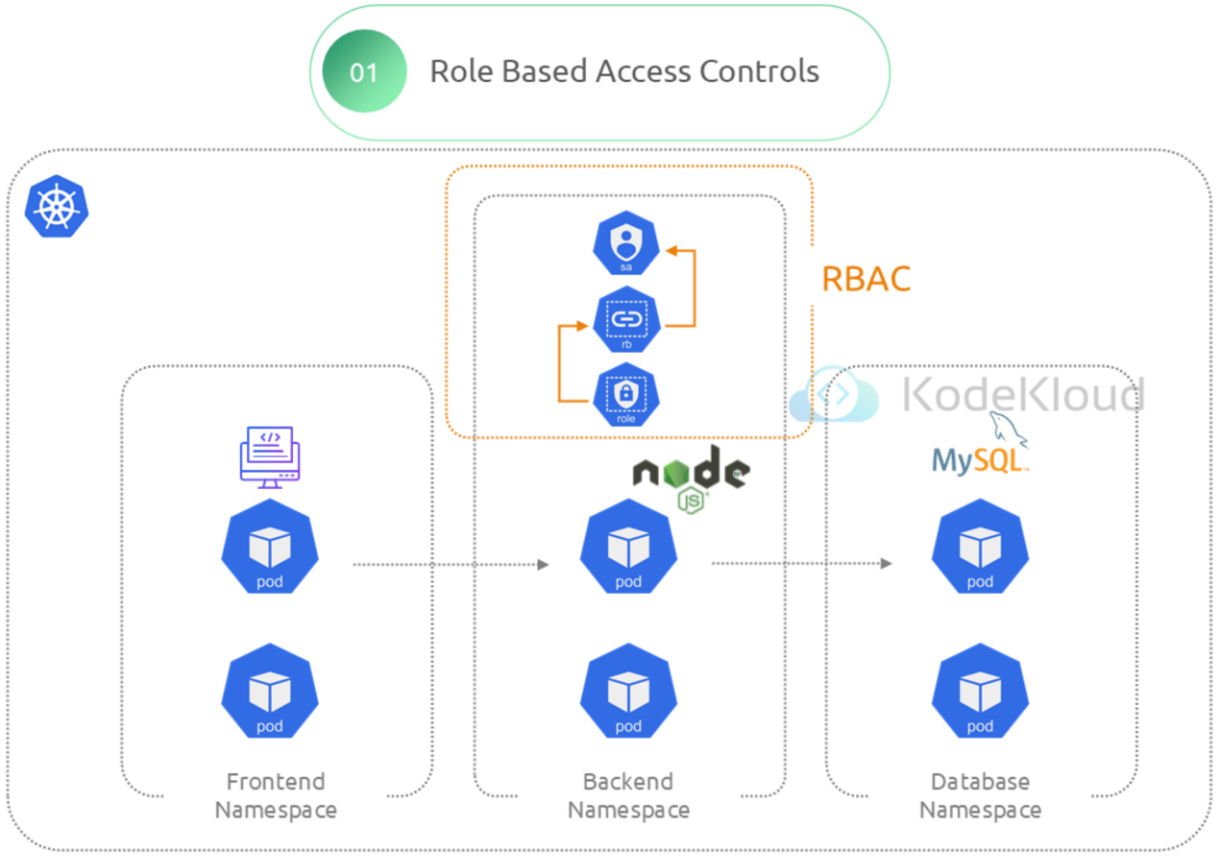
Find Privileged Workload with Unrestricted Access to Host Filesystem (Privileged or File System Mount):

•The attacker searches for privileged workloads with unrestricted access to the host's filesystem. This could be a container running in "privileged" mode or one with specific file system mounts that expose host files to the container. Privileged containers often bypass many security boundaries, making it easier for an attacker to access host resources, including sensitive files on the master node.

Workload is on Master Nodes:

•The attacker identifies and targets workloads running directly on the master nodes. Since master nodes contain Kubernetes control plane components, these workloads often have higher privileges and may have direct or indirect access to critical files, such as the Kubernetes API certificates. By gaining access to workloads on master nodes, attackers position themselves closer to the cluster's core infrastructure, increasing the impact of their control.

Mitigating Persistence Risks

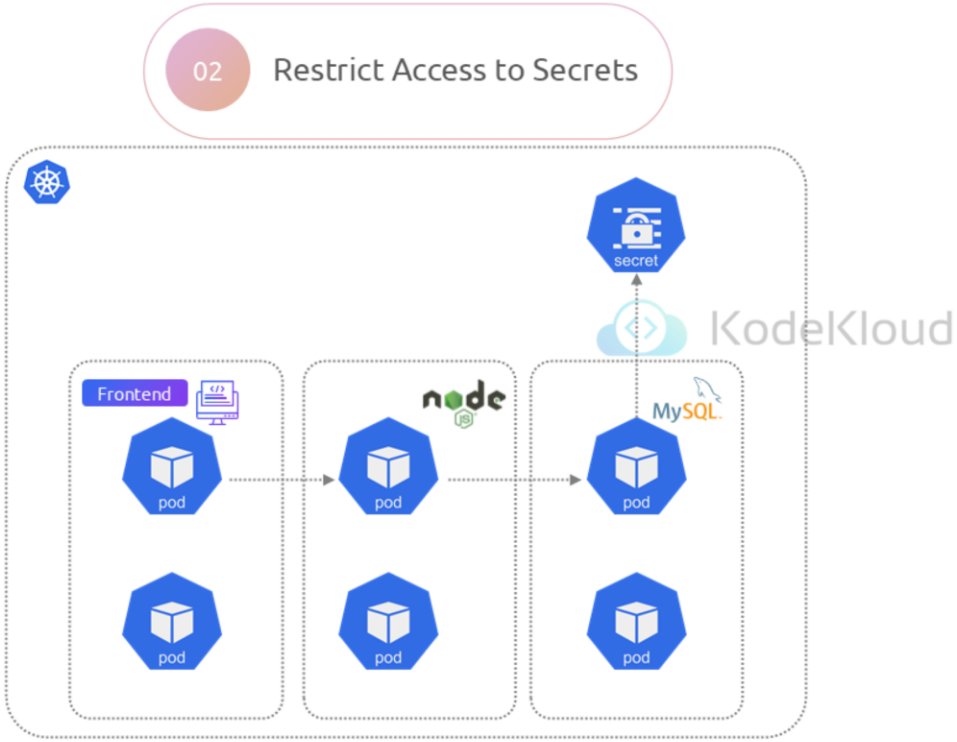


Now let's talk about how to Mitigate Persistence Risks..

To protect our Kubernetes cluster from persistence threats, we need to implement several security controls.

First Restricting RBAC Permissions. In our application, we review and tighten the RBAC policies for service accounts used by our Node.js backend pods. This ensures they cannot read secrets or perform administrative actions, reducing the risk of attackers exploiting excessive permissions.

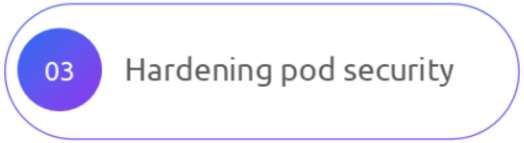
Mitigating Persistence Risks



Next, Securing Secrets Management.

We store database credentials and API keys in Kubernetes secrets and restrict access to these secrets to only the pods that require them, such as the MySQL database pod. This prevents unauthorized access to sensitive information.

Mitigating Persistence Risks



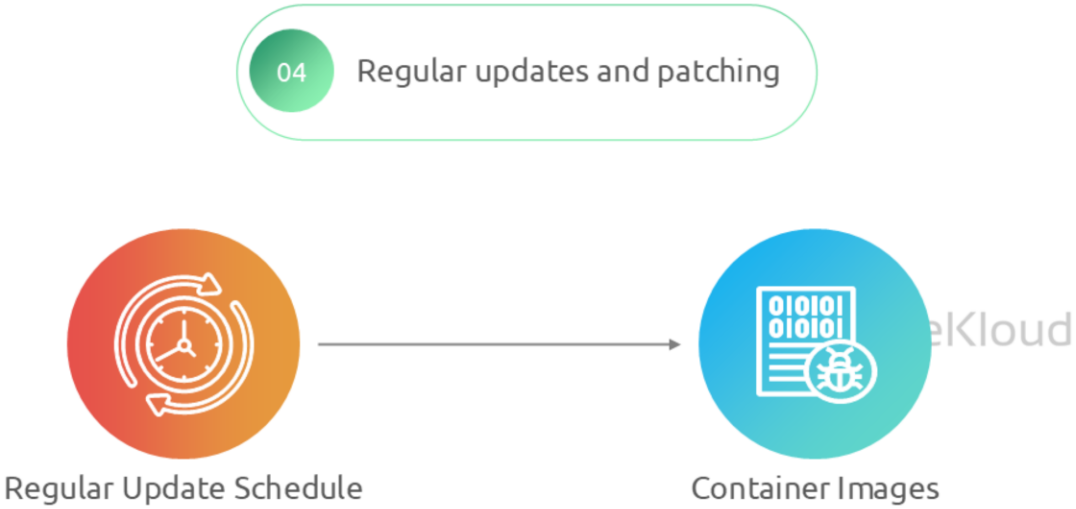
KodeKloud

Prevent the use of privileged containers

Enforce read-only root filesystems

Another important control is Hardening Pod Security. We configure Pod Security Policies to prevent the use of privileged containers and enforce read-only root filesystems for our application pods.
This means that even if an attacker gains access to a pod, they are limited in what they can do, reducing the impact of the attack.

Mitigating Persistence Risks



It is also important to have Regular Updates and Patching. We set up a regular update schedule for all container images used by our application, including Nginx, Node.js, and MySQL, and monitor for any new vulnerabilities.

Keeping systems up to date with the latest security patches helps prevent attackers from exploiting known vulnerabilities.

Mitigating Persistence Risks

05 Monitoring and auditing

Monitor for suspicious activities

Regular audit of Kubernetes events

Finally, Monitoring and Auditing. We implement logging and monitoring to detect suspicious activities and audit Kubernetes events regularly.

We set up monitoring for our Kubernetes cluster to track access to secrets, changes in RBAC policies, and the creation of new pods. Alerting on any suspicious activities helps us quickly respond to potential threats and mitigate the risk of persistence.

Summary

01 Persistence allows attackers to maintain access in clusters

02 Attackers read secrets and leverage misconfigurations for persistence

03 Restrict RBAC permissions to prevent unauthorized access

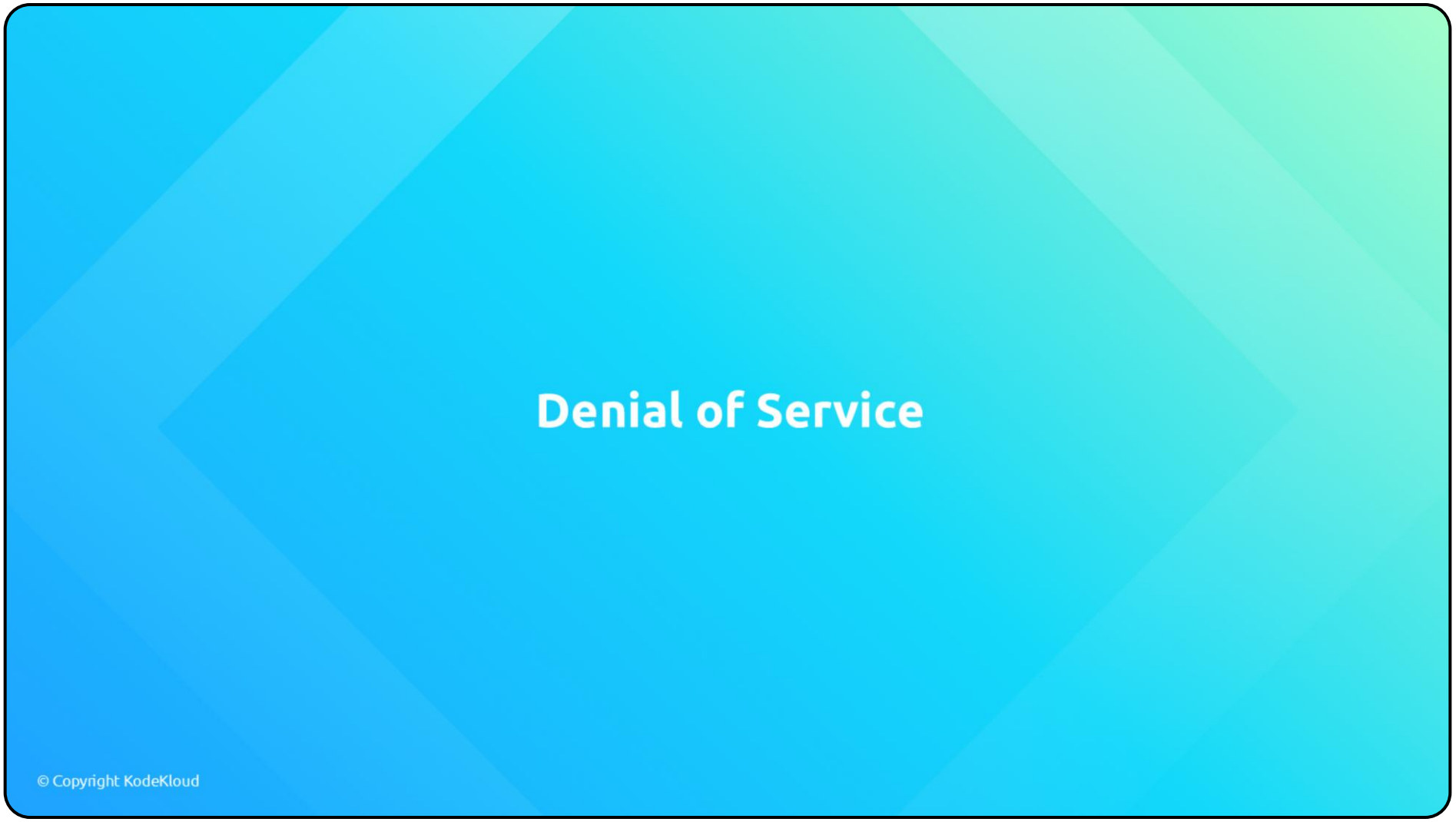
04 Secure secrets management to protect sensitive information

05 Harden pod security to limit attack impact

Summary

06 Regularly update and patch container images

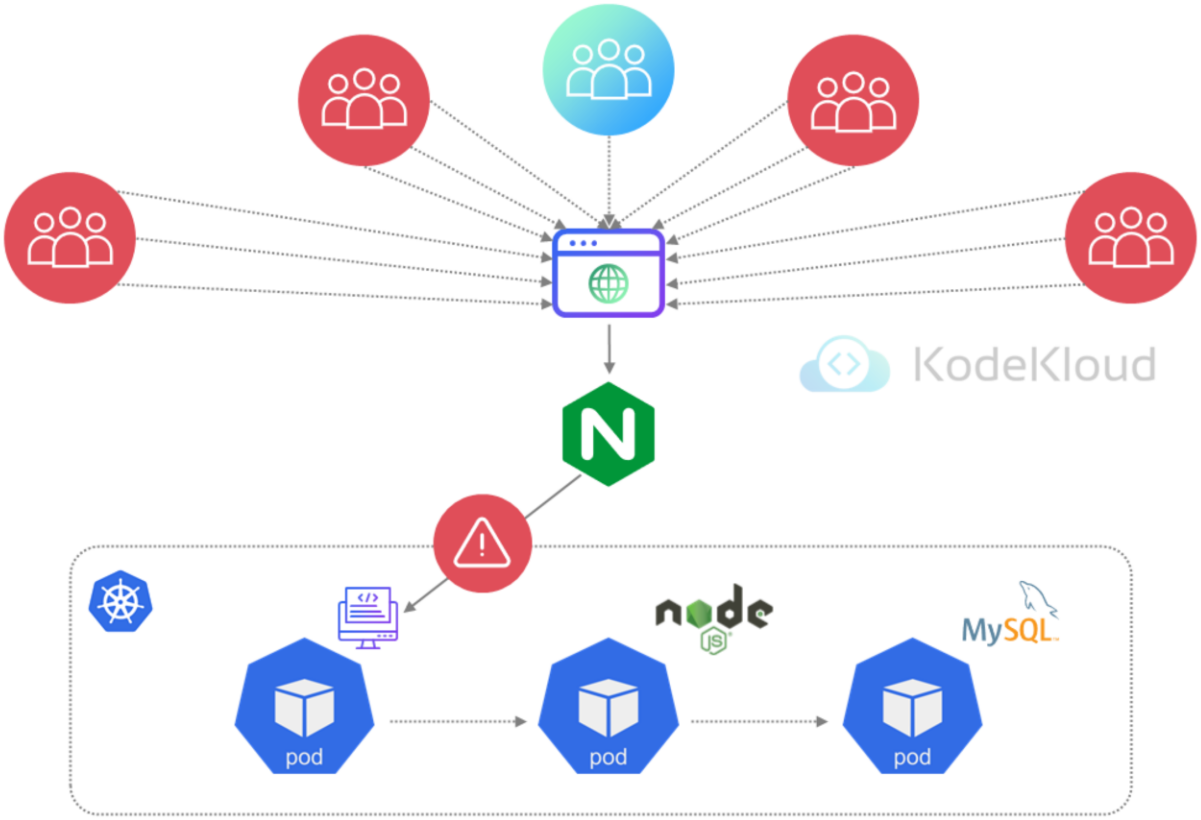
07 Monitor and audit Kubernetes events for suspicious activities



Introduction

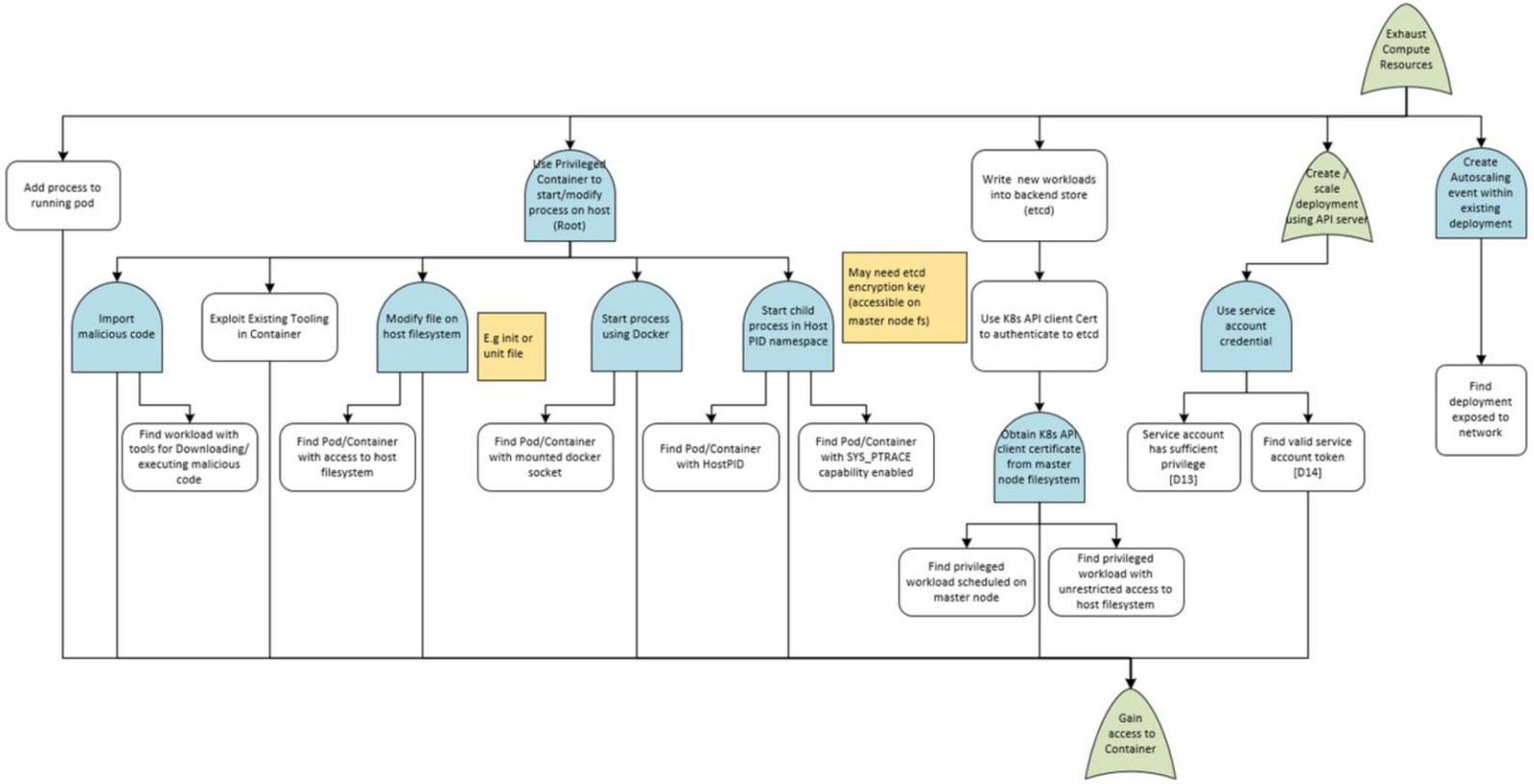
In this lesson, we will explore how attackers can launch Denial of Service (DoS) attacks in a Kubernetes environment, using our multi-tier web application as an example. We'll identify potential vulnerabilities and discuss how to mitigate these risks.

Understanding Denial of Service (DoS)



Denial of Service (DoS) attack is where attackers overwhelm the system with illegitimate requests or exhaust its resources, making it unavailable to legitimate users.

Attack Vectors for DoS

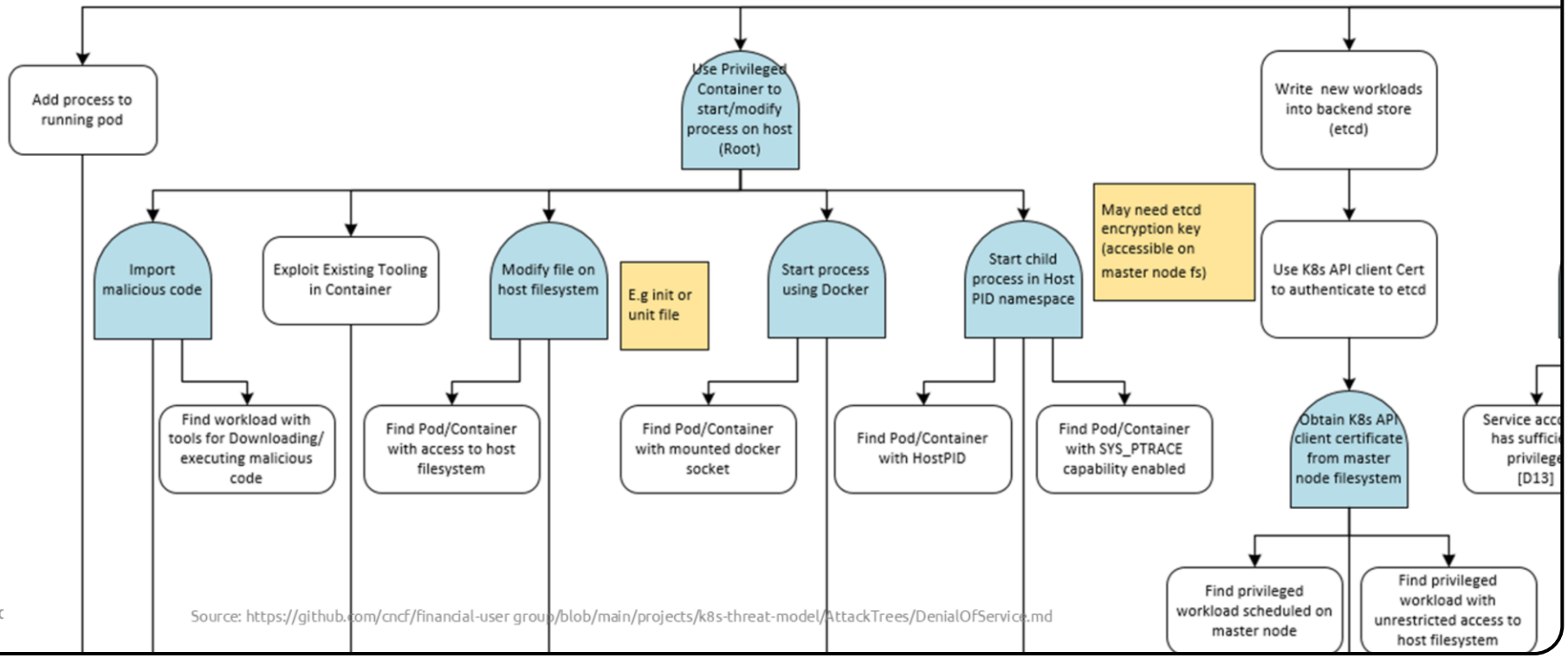


Source: https://github.com/cncf/financial-user group/blob/main/projects/k8s-threat-model/AttackTrees/DenialOfService.md

Attack Vectors for DoS

The CNCF has outlined two main approaches for DoS attacks in Kubernetes.

Attack Vectors for DoS



Add Process to Running Pod:

•The attacker adds new processes to an already running pod, increasing the load on the pod and potentially consuming more memory or CPU resources. By overwhelming the pod with additional processes, they can cause it to slow down, become unresponsive, or crash, disrupting the application's availability.

Use Privileged Container to Start/Modify Process on Host (Root):

•In this step, the attacker leverages a privileged container to start or modify processes directly on the host. Running processes at the host level (especially as the root user) allows them to consume host resources extensively, impacting not only the

targeted pod but also other applications running on the same node. This can lead to a resource bottleneck or even crash the host, causing a large-scale service disruption.

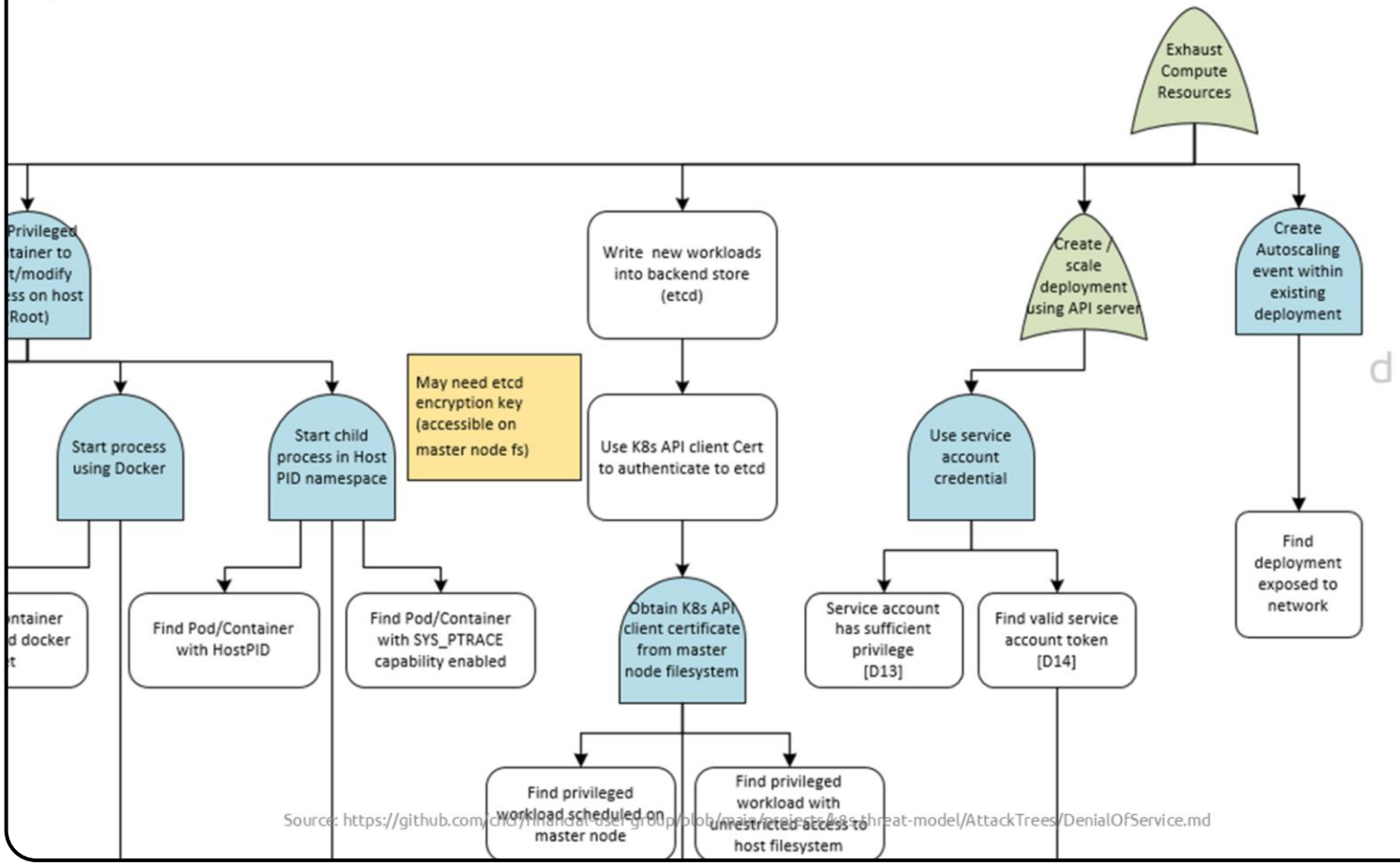
Write New Workloads into Backend Store (etcd):

•By directly writing new workload entries into the backend store (etcd), the attacker can bypass regular Kubernetes API controls and inflate the number of workloads scheduled within the cluster. As etcd is central to Kubernetes' operation, overwhelming it with new entries can degrade performance and cause delays in response times, potentially leading to a partial or full cluster unavailability.

Exhaust Compute Resources:

•The attacker’s goal is to exhaust compute resources across the cluster, such as CPU, memory, or storage, to degrade system performance and availability. This can involve deploying additional workloads, scaling existing ones, or forcing Kubernetes to allocate more resources than it can handle.

Attack Vectors for DoS



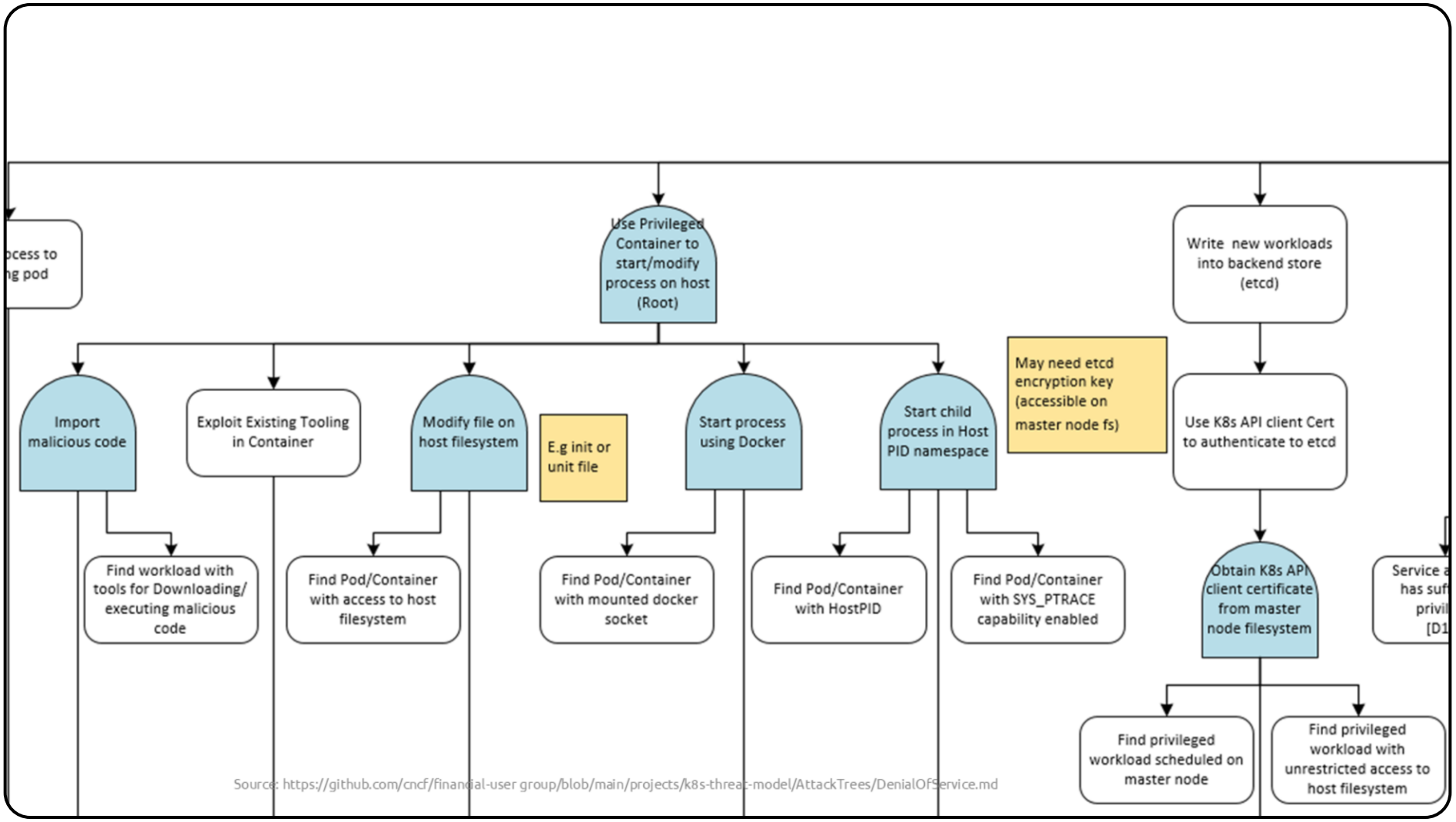
•Create/Scale Deployment Using API Server:

- The attacker creates or scales deployments by interacting with the Kubernetes API server. By requesting more replicas than the system can handle, they place excessive demands on compute resources. Scaling deployments in this way can trigger a resource crunch, slowing down or crashing other critical services due to resource starvation.

•Create Autoscaling Event within Existing Deployment:

- The attacker triggers autoscaling within an existing deployment, potentially by simulating increased load. Autoscaling typically adds more pods to handle high demand, but by manipulating this mechanism, the attacker

forces Kubernetes to continuously allocate new resources. This exhausts available compute capacity, leading to resource depletion and potential service interruptions.



Import Malicious Code:

•The attacker introduces malicious code into the container, which could be achieved by exploiting existing tools or vulnerabilities within the container. This code can then execute within the container environment, potentially with privileges to impact other components or processes.

•Find Workload with Tools for Downloading/Executing Malicious Code:

- The attacker searches for workloads that already have tools, such as curl, wget, or package managers, which allow them to download and execute their malicious code. Containers with these tools present are easier to exploit for

code import and execution.

Exploit Existing Tooling in Container:

•Leveraging tools already present in the container, the attacker can run commands or download additional components without introducing new files, making detection harder. These tools may include interpreters, debuggers, or shell utilities that can facilitate code execution.

•Find Pod/Container with Access to Host Filesystem:

- The attacker seeks out a container that has a mounted host filesystem. This can allow them to read or modify files on the host, expanding their reach beyond the container and potentially affecting the entire node.

Modify File on Host Filesystem (e.g., Init or Unit File):

•By modifying files on the host's filesystem, such as init or unit files, the attacker can ensure their processes start automatically when the system boots. This step grants persistence even across node restarts.

•Find Pod/Container with Access to Host Filesystem or with Mounted Docker Socket:

- The attacker looks for a container that has access to critical areas of the host filesystem or a mounted Docker socket. With access to Docker, they can control other containers or directly impact the host's operation by modifying critical files.

Use Privileged Container to Start/Modify Process on Host (Root):

•A privileged container allows the attacker to run processes as the root user on the host. This gives them the power to start or modify host-level processes, providing broad control over the system and potential disruption or persistence mechanisms.

•Start Process Using Docker:

- The attacker uses Docker to initiate new processes on the host, often by deploying new containers or manipulating existing ones. If the container has a mounted Docker socket, they can issue commands directly to the Docker daemon to start or manage processes on the host.

Start Child Process in Host PID Namespace:

•By starting a child process in the host's PID (process ID) namespace, the attacker can interact with host processes directly, making it easier to influence or manipulate system-level operations from within the container.

•Find Pod/Container with HostPID Enabled:

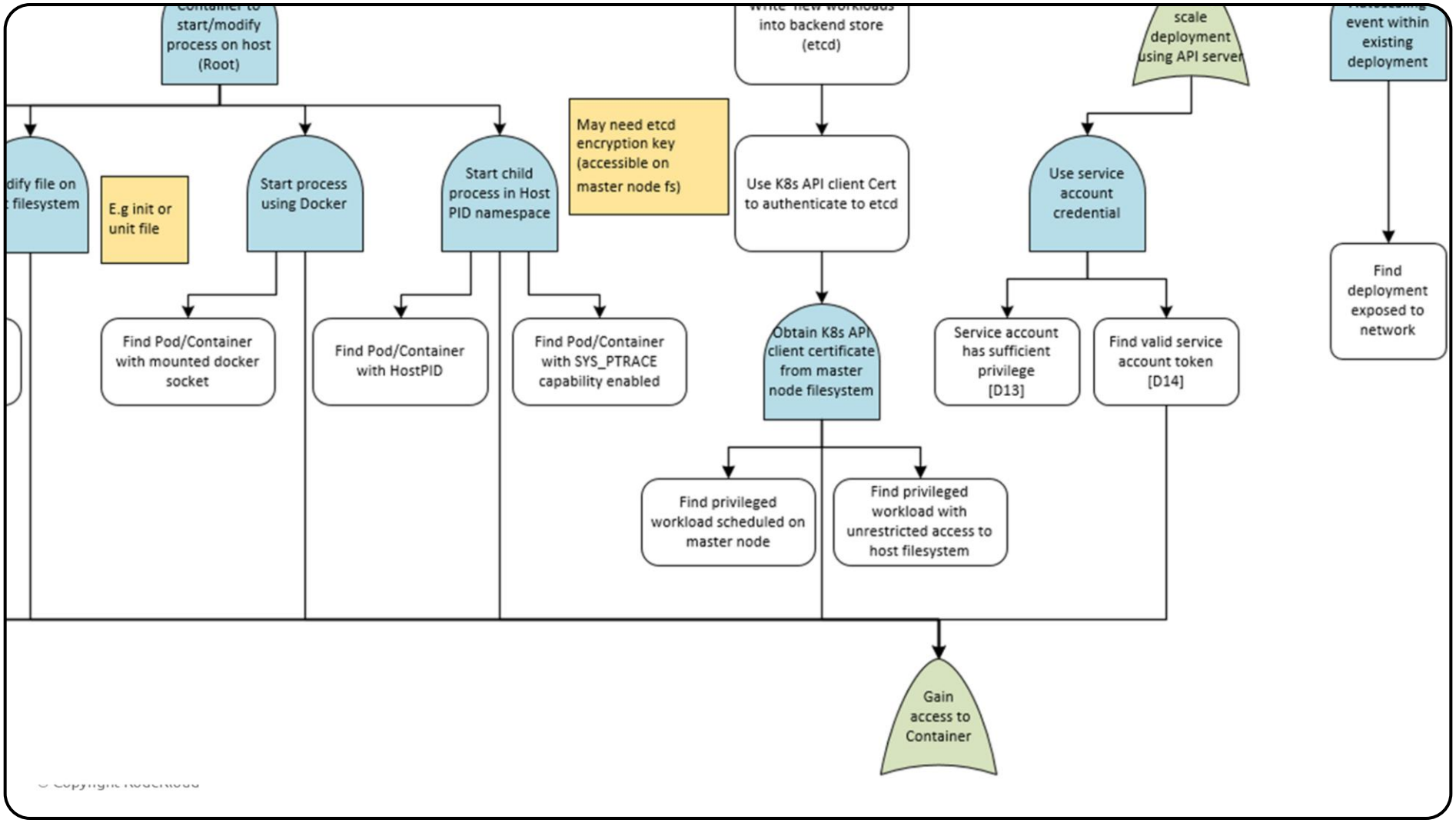
- The attacker searches for a container with the HostPID flag enabled, which allows access to the host's process namespace. This makes it possible to view or interfere with host processes, potentially escalating privileges or manipulating other containers.

Find Pod/Container with SYS_PTRACE Capability Enabled:

•The attacker identifies containers with the SYS_PTRACE capability enabled. This capability allows them to trace or interact with other processes, which can be useful for debugging but can also be exploited to hijack or manipulate host processes, particularly in the context of a privilege escalation attack.

May Need etcd Encryption Key (Accessible on Master Node Filesystem):

•The attacker might require access to the etcd encryption key, which is stored on the master node's filesystem. With this key, they can decrypt sensitive data stored in etcd, gaining access to potentially valuable information like secrets, configurations, and cluster data.



Use Kubernetes API Client Certificate to Authenticate to etcd:

•By leveraging the Kubernetes API client certificate, the attacker gains authenticated access to etcd, the key-value store that holds the cluster's configuration and state. This access allows them to manipulate data within etcd, potentially modifying configurations, adding new resources, or reading sensitive information.

Obtain Kubernetes API Client Certificate from Master Node Filesystem:

•The attacker targets the master node's filesystem to locate and obtain the Kubernetes API client certificate. This certificate is usually stored in specific directories on the master node and grants significant privileges to access and modify cluster

configurations. With this certificate, the attacker can authenticate as a trusted entity within the cluster.

Find Privileged Workload Scheduled on Master Node:

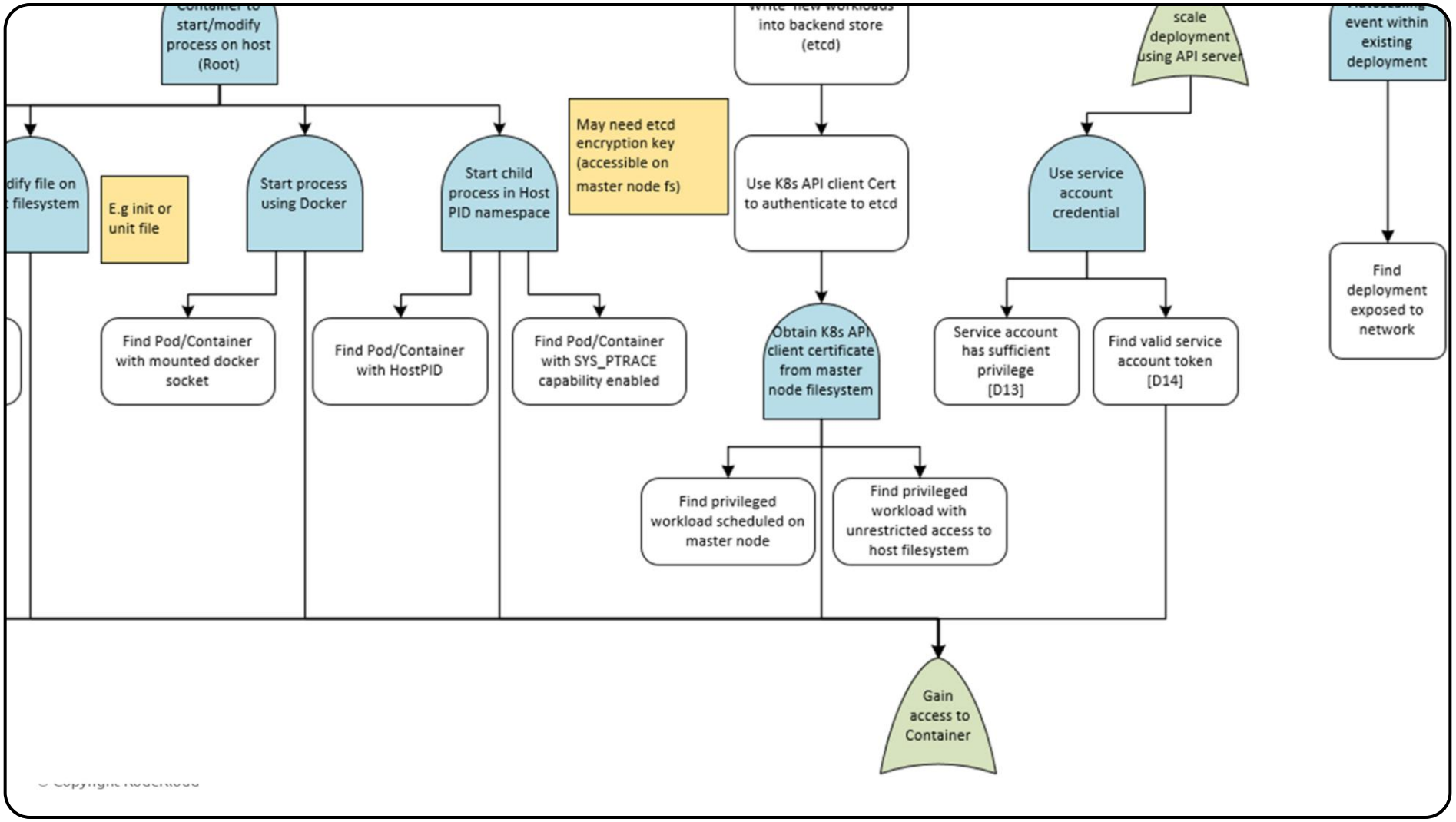
•The attacker searches for a privileged workload that is scheduled directly on the master node. Privileged workloads on the master node often have enhanced permissions and may have access to sensitive components of the cluster. By compromising such a workload, the attacker positions themselves closer to critical data and control mechanisms.

Find Privileged Workload with Unrestricted Access to Node Filesystem:

•Here, the attacker seeks out a privileged workload that has unrestricted access to the node's filesystem. This kind of access allows the attacker to manipulate system files, potentially affecting Kubernetes components, configurations, or other applications on the node. Privileged workloads with full filesystem access are a common target for attackers trying to escalate their control within the environment.

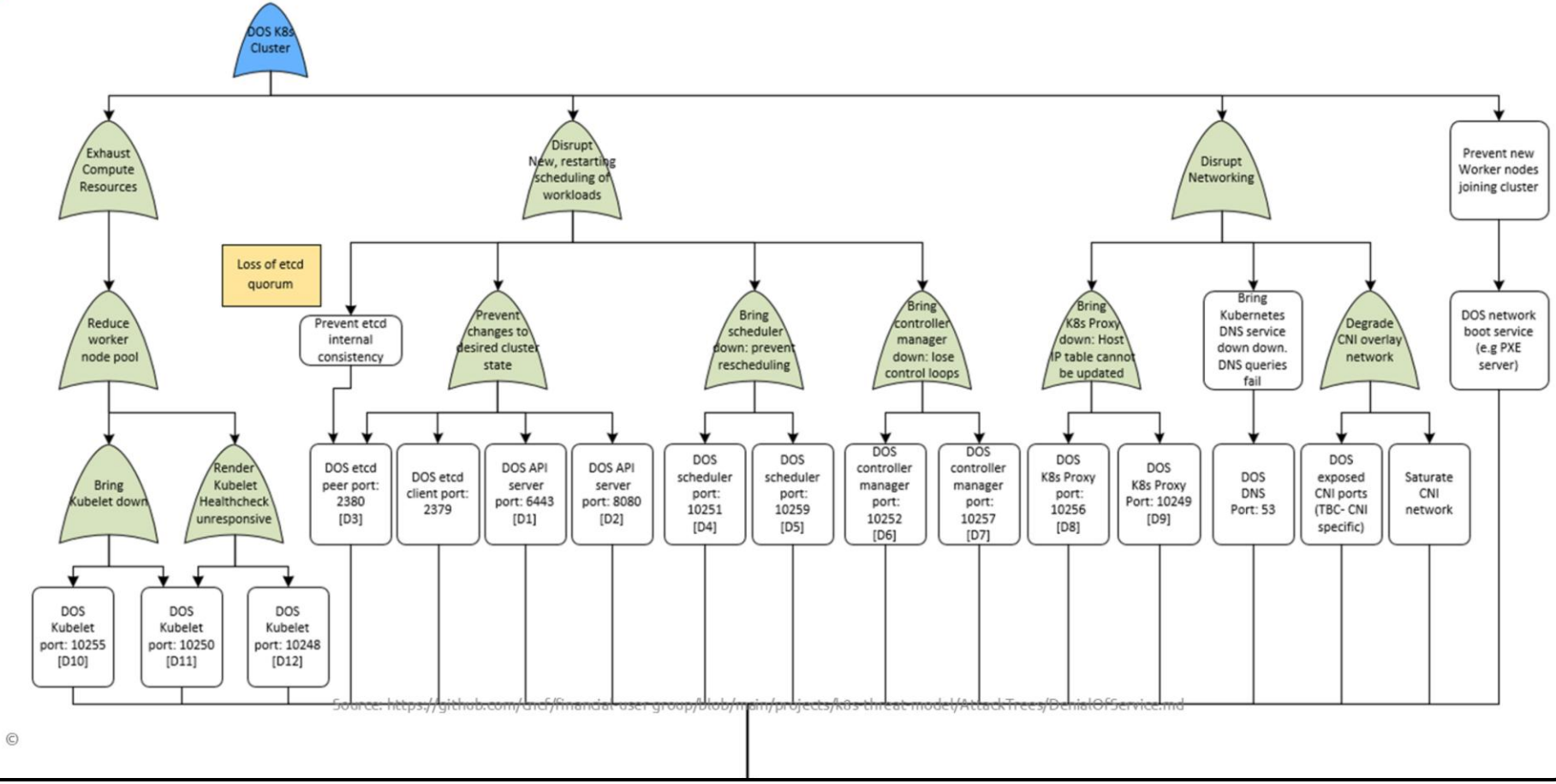
Service Account with Sufficient Privileges (D13):

•The attacker identifies a service account that has the necessary privileges to perform actions within the cluster. By compromising a service account with extensive permissions, they can manipulate resources and configurations within Kubernetes. This service account, if privileged enough, could grant the attacker the ability to interact with critical systems like etcd or the API server.



And then using those techniques to eventually gain access to container.

Attack Vectors for DoS

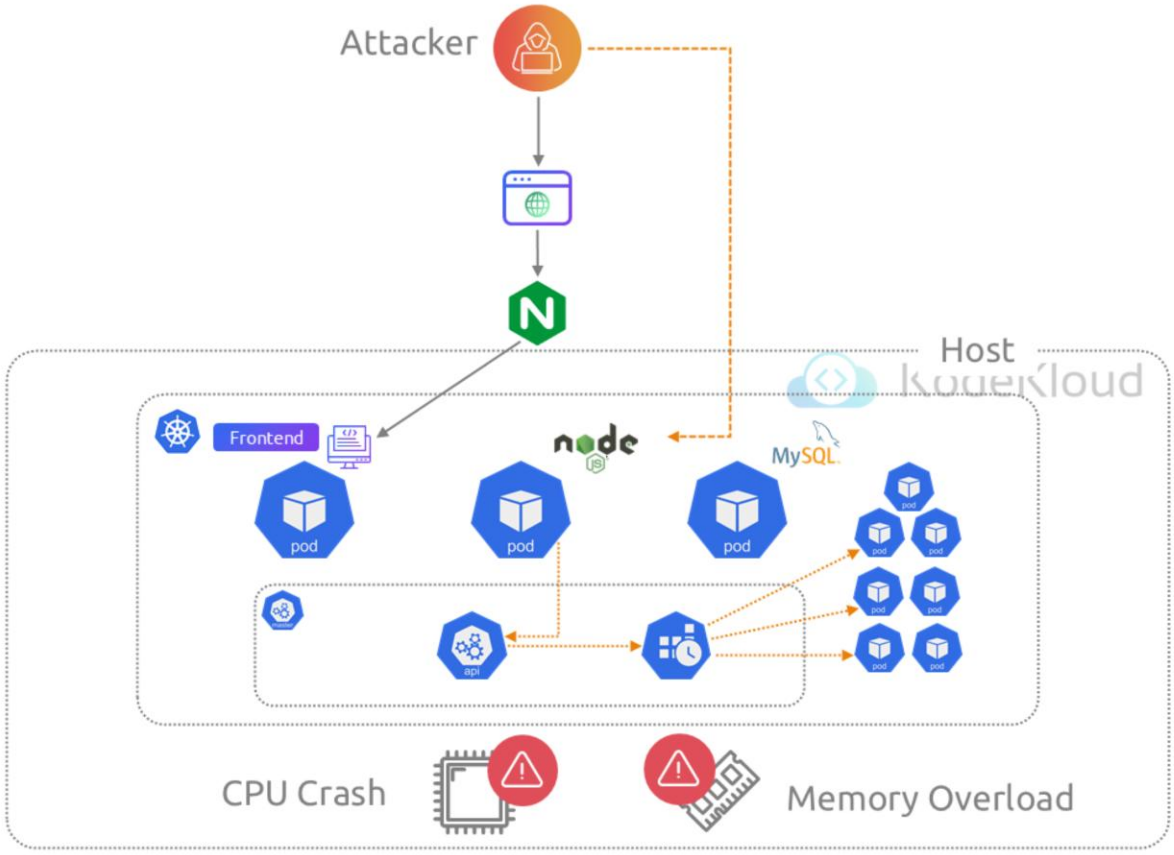


Another approach is to use network attacks targeting the API server.

So here, we're looking at a bunch of ways an attacker could throw a Kubernetes cluster into chaos through different DoS (Denial of Service) tactics. There are three main categories here. First up, exhausting resources — this is like hammering the cluster's worker nodes to the point where they just can't keep up, maybe by taking down kubelets or making health checks fail. Next, we've got disrupting scheduling, where they could knock out components like the scheduler or API server, basically

stopping any new workloads from being scheduled or restarted. Finally, there's network disruption, where attackers could mess with DNS or the network plugins (CNI), making it impossible for nodes to communicate or new nodes to join the cluster. All of these tactics aim to leave the cluster paralyzed and unable to operate normally.

From a Compromised Container

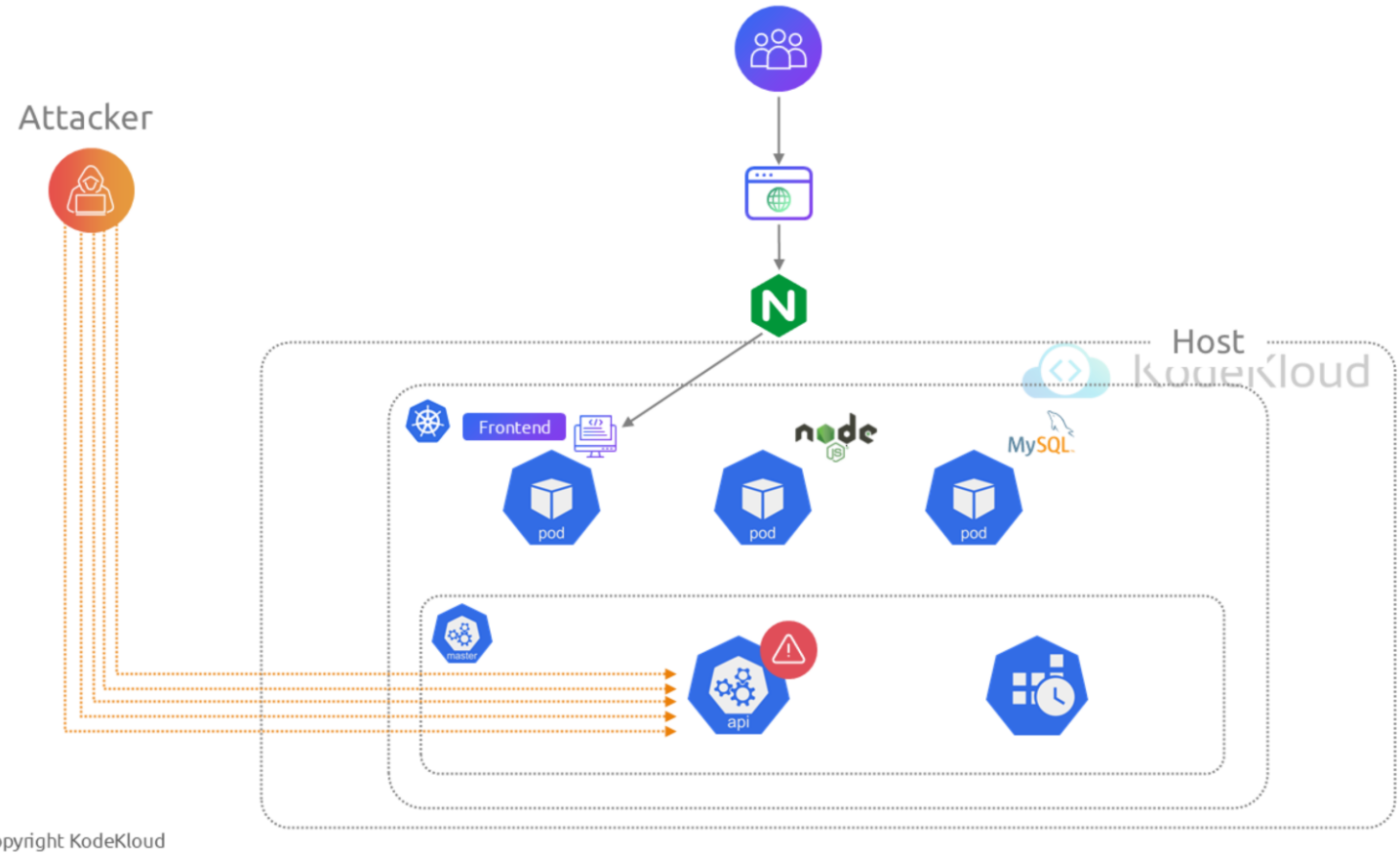


Let's take a look at the first approach - From a Compromised Container.

Suppose an attacker gains access to our Node.js container in the backend. Using the service account token, they could start many new containers via the API server. This action could overwhelm the cluster's CPU and memory resources, leading to resource starvation and causing a DoS for our application.

Without proper quotas and mitigations, the entire cluster's performance could be severely affected.

From a Compromised Container



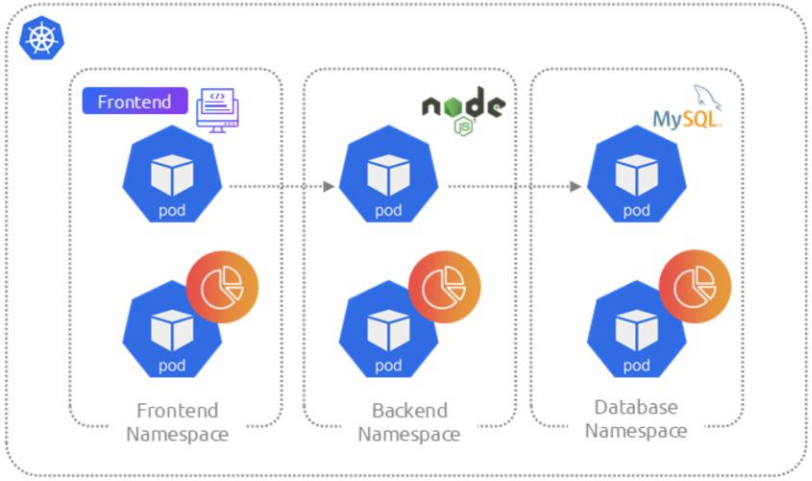
Next let's look at, Network-Based Attacks targeting API server.

Now, imagine an attacker with network access to the Kubernetes control plane. By flooding the API server endpoint with numerous requests, they can consume its resources, making it unresponsive. This disrupts the cluster's operations and causes a DoS for the entire system.

Proper firewall configurations can mitigate many of these attacks, but understanding the potential threats is essential for

comprehensive protection.

Mitigating DoS Risks




```yaml
apiVersion: v1
kind: Pod
metadata:
  name: frontend
  labels:
    app: frontend
spec:
  containers:
  - name: frontend
    image: nginx:latest
    resources:
      limits:
        memory: "512Mi"
        cpu: "500m"
      requests:
        memory: "256Mi"
```

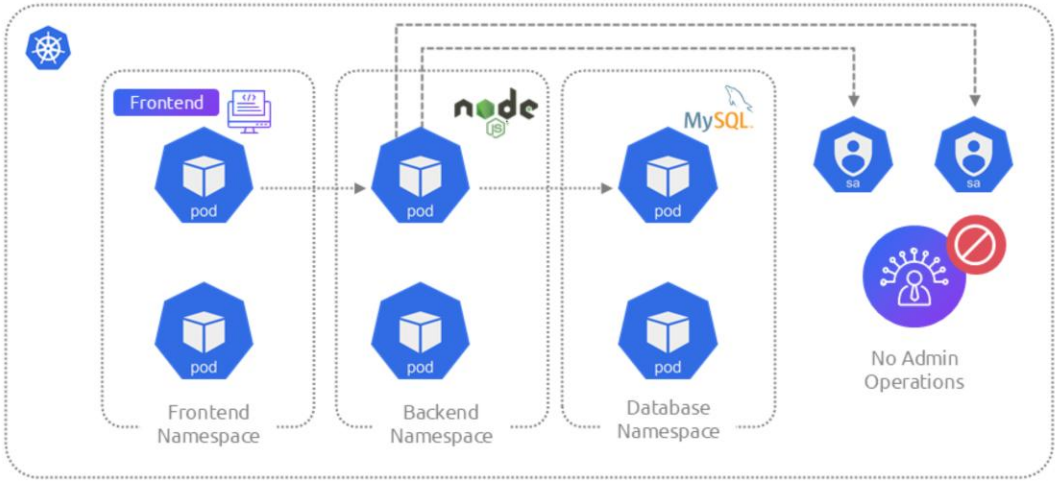

Mitigating DoS Risks

To protect our Kubernetes cluster from DoS attacks, we need to implement several security controls using our application as an example.

First, we need to set resource quotas and limits for each namespace to ensure no single container or group of containers can

consume excessive resources. In our application, we set quotas on the number of pods, CPU, and memory usage for the frontend (Nginx), backend (Node.js microservices), and database (MySQL) namespaces. This ensures that even if an attacker compromises one part of the application, they cannot exhaust the cluster's resources.

Mitigating DoS Risks

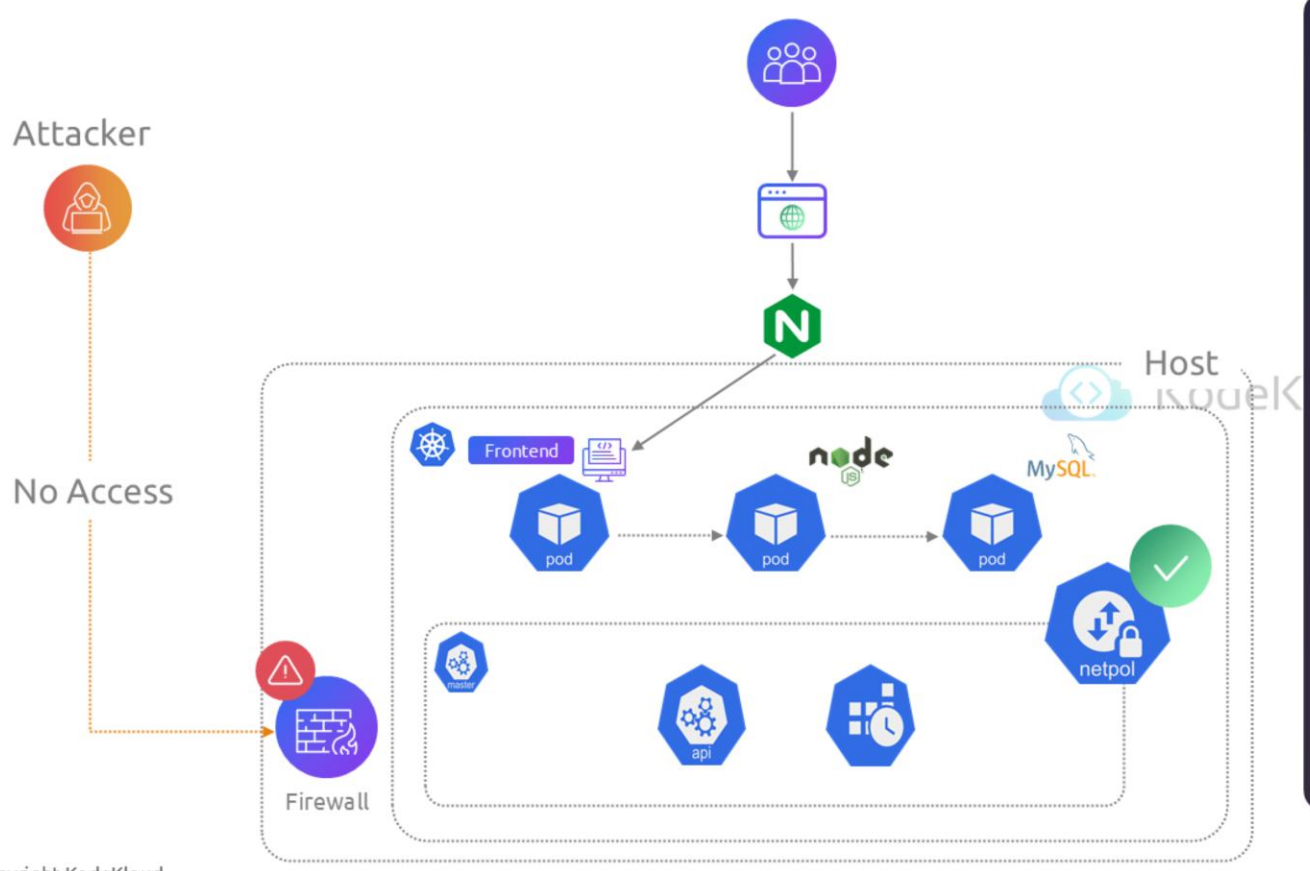



```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: backend-sa
  namespace: default
```

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: backend-read-only
rules:
- apiGroups: [""]
  resources: ["pods", "services", "configmaps"]
  # Only allows read actions; no create, update, or delete permissions
  verbs: ["get", "list"]
```


Next, we must secure service accounts by restricting their permissions to the minimum necessary. In our application, we ensure that service accounts used by the Node.js backend pods cannot create new pods or perform administrative actions. This limits the attacker's ability to use compromised service accounts to launch additional containers.

Mitigating DoS Risks




```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: restrict-api-access
  namespace: kube-system
spec:
  podSelector:
    matchLabels:
      component: kube-apiserver
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          name: trusted-namespace
    - podSelector:
        matchLabels:
          access: api-server
    ports:
    - protocol: TCP
      port: 6443
```


We also need to use Network Policies to control traffic flow within the cluster and configure firewalls to protect the control plane endpoints. For instance, we can limit access to the API server to only trusted IP addresses and internal components. By doing this, we prevent unauthorized network access that could be used to flood the API server with requests.

Mitigating DoS Risks

Monitoring

Detect unusual activities

Set up alerts

Potential DoS attacks

```yaml
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: dos-alert-rules
  namespace: default
spec:
  groups:
  - name: dos-alerts
    rules:
    - alert: HighCPUUsage
      expr: sum(rate(container_cpu_usage_seconds_total[5m])) by (namespace) > 0.8
      for: 5m
      labels:
        severity: warning
      annotations:
        summary: "High CPU usage detected"
        description: "CPU usage is above 80% in namespace {{ $labels.namespace }}. This could
indicate a DoS attack."
```


Finally, regular monitoring and alerts are crucial. We implement monitoring to detect unusual activities and set up alerts for potential DoS attacks. Tools like Prometheus and Grafana help us monitor the cluster's resource usage and alert us to any spikes that may indicate a DoS attempt. For example, this This alert fires if the CPU usage across containers in a namespace exceeds 80% for 5 minutes.

Summary

01 DoS attacks overwhelm system resources, causing unresponsiveness

02 Set resource quotas to prevent excessive resource usage

03 Restrict service account permissions to limit potential attacks

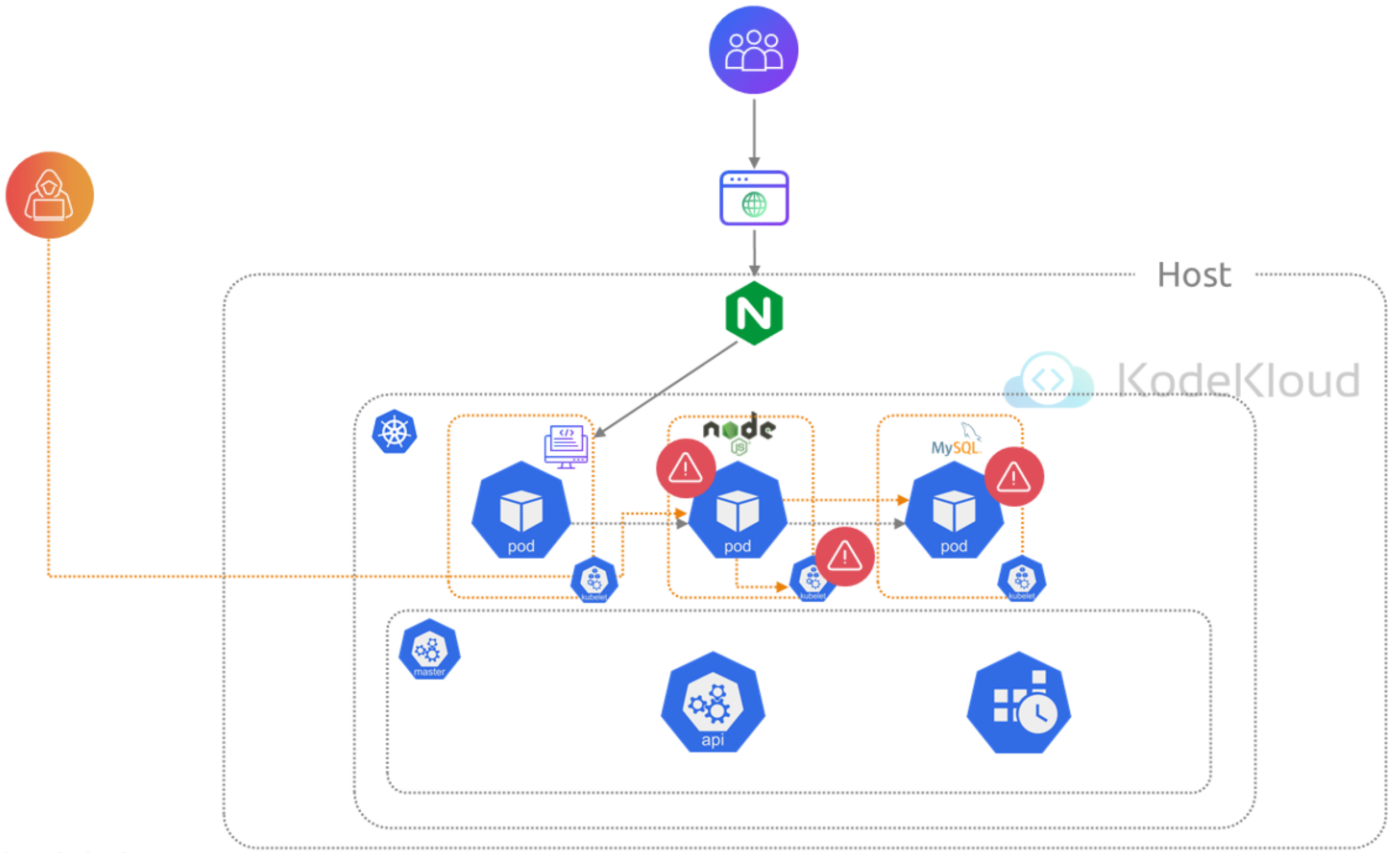
04 Use Network Policies and firewalls to control access

05 Monitor and alert on unusual activity for quick response

INTRODUCTION

In this lesson, we will explore how attackers can execute malicious code in a Kubernetes environment using our multi-tier web application as an example. We'll identify potential vulnerabilities and discuss how to mitigate these risks.

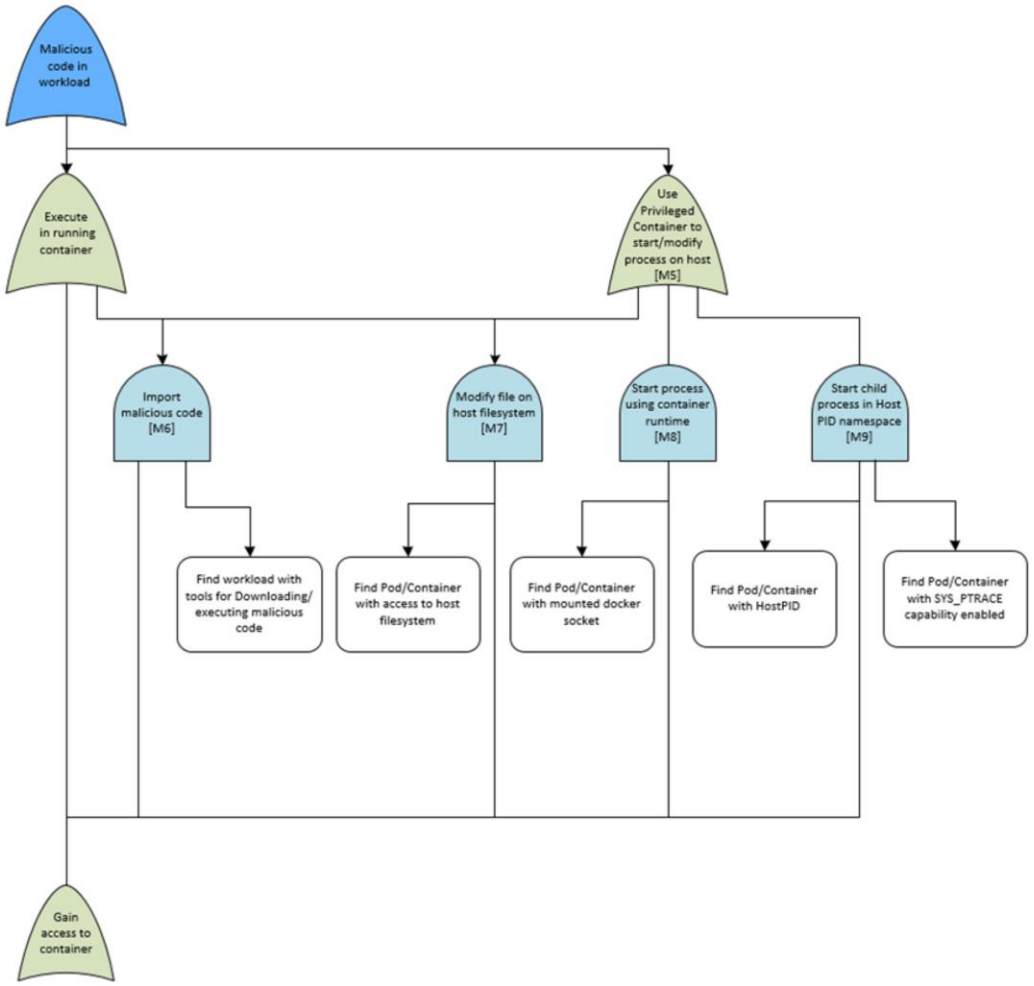
Malicious Code Execution



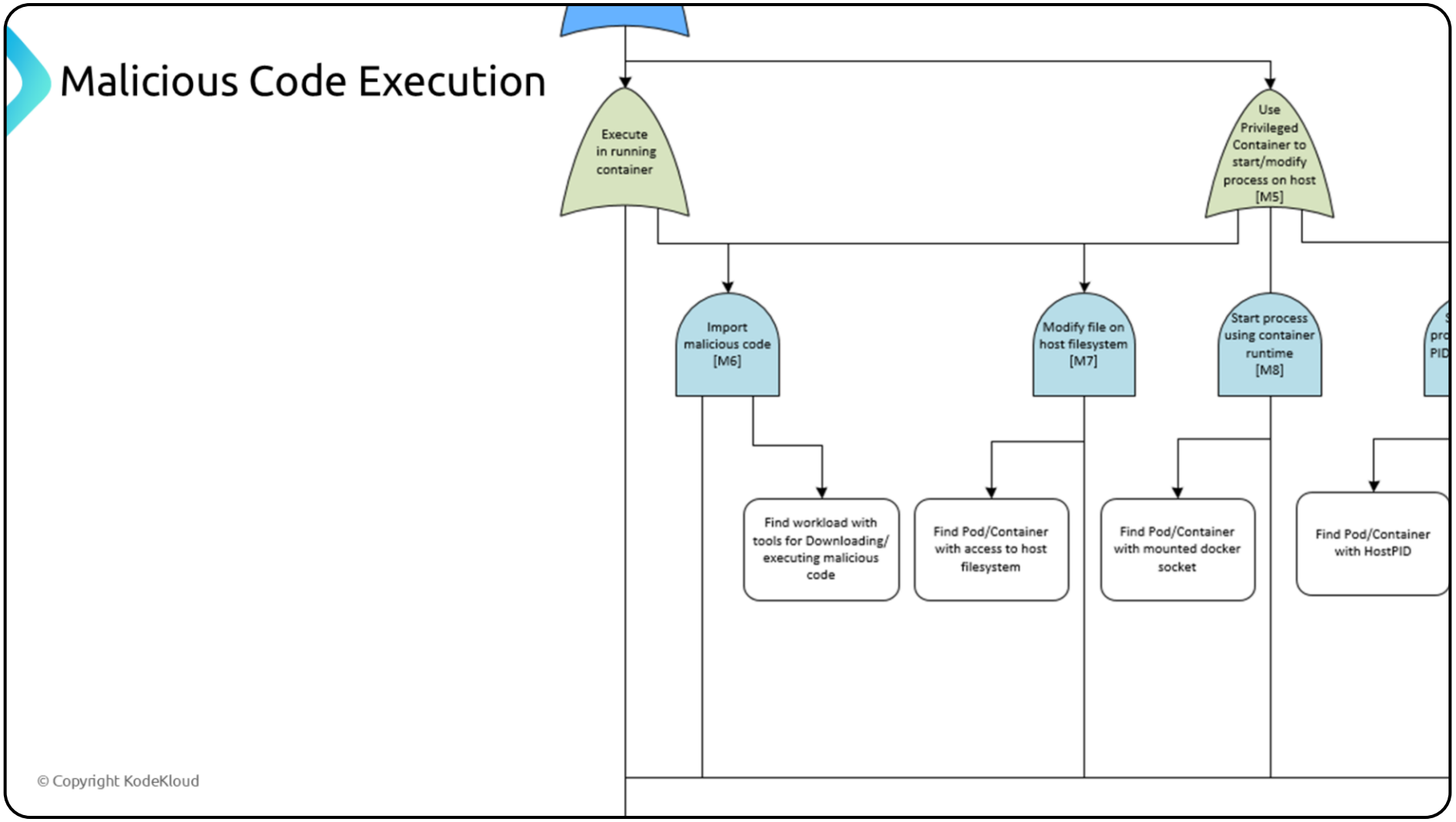
Let's first understand what is Malicious Code Execution

Imagine An attacker finds a vulnerability in our application and uses it to gain access to one of the containers. The attacker's goal is to execute malicious code within our Kubernetes cluster, potentially leading to further exploitation and control over the environment.

Malicious Code Execution

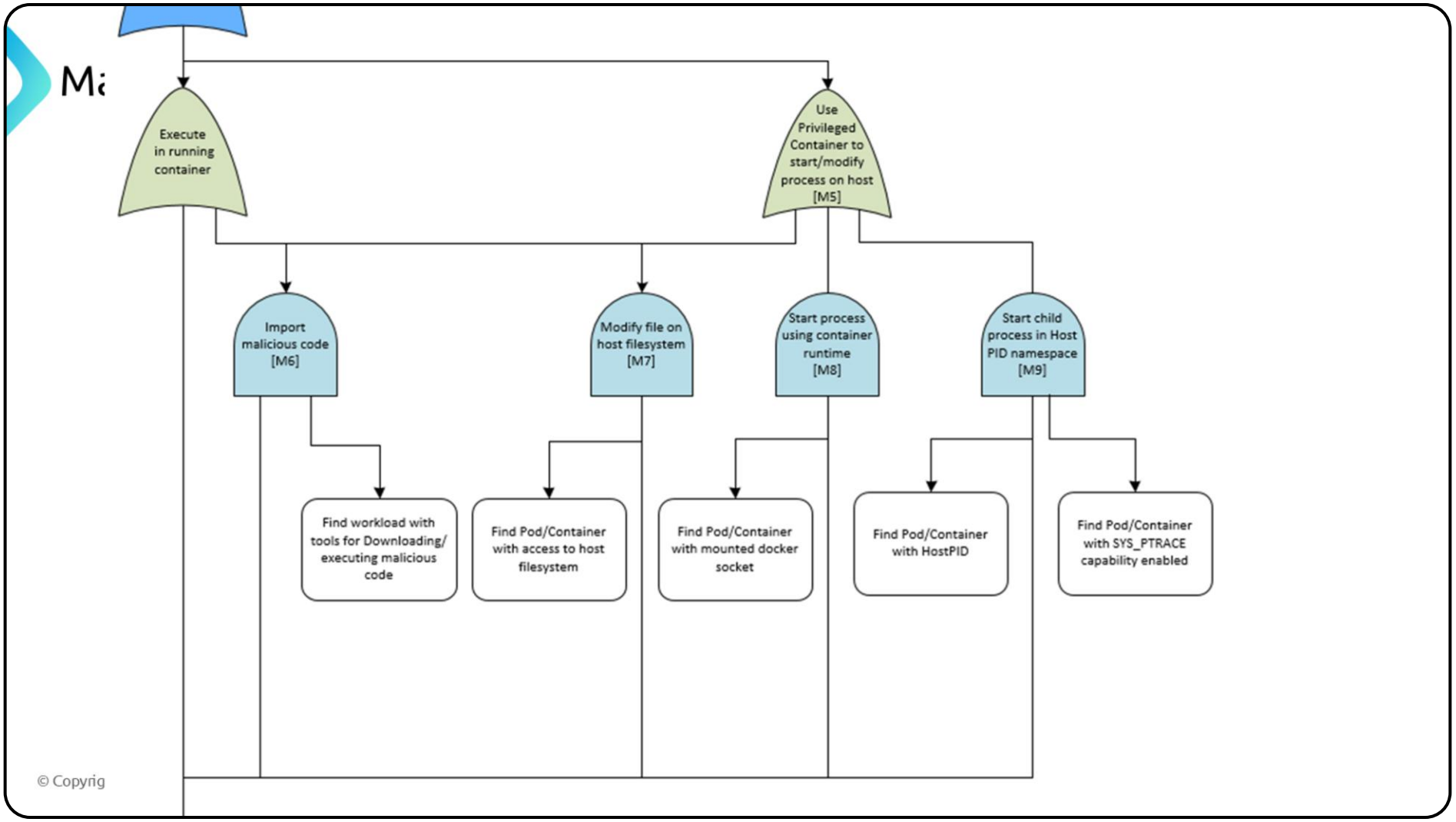


Lets look at the AttackTree for Malicious code in workload.



Import Malicious Code [M6]:
•The attacker might need additional tools or scripts to carry out their goals. They could import more malicious code by finding a workload that has tools like curl or wget to download additional code.
•Find Workload with Tools for Downloading/Executing Malicious Code: The attacker looks for a container with network access and tools for downloading, so they can import extra code without being restricted by the container's default environment.
Modify File on Host Filesystem [M7]:
•If the attacker has access to the host filesystem, they can modify important files directly on the host. This might be

configuration files, startup scripts, or system binaries, allowing them to make their changes persistent.
•Find Pod/Container with Access to Host Filesystem: To do this, they need to find a container that has the host filesystem mounted, giving them the freedom to change files outside their container’s usual scope.



Start Process Using Container Runtime [M8]:

•The attacker may use the container runtime (like Docker) to start new processes on the host. This would let them execute their malicious code directly in the host's context.

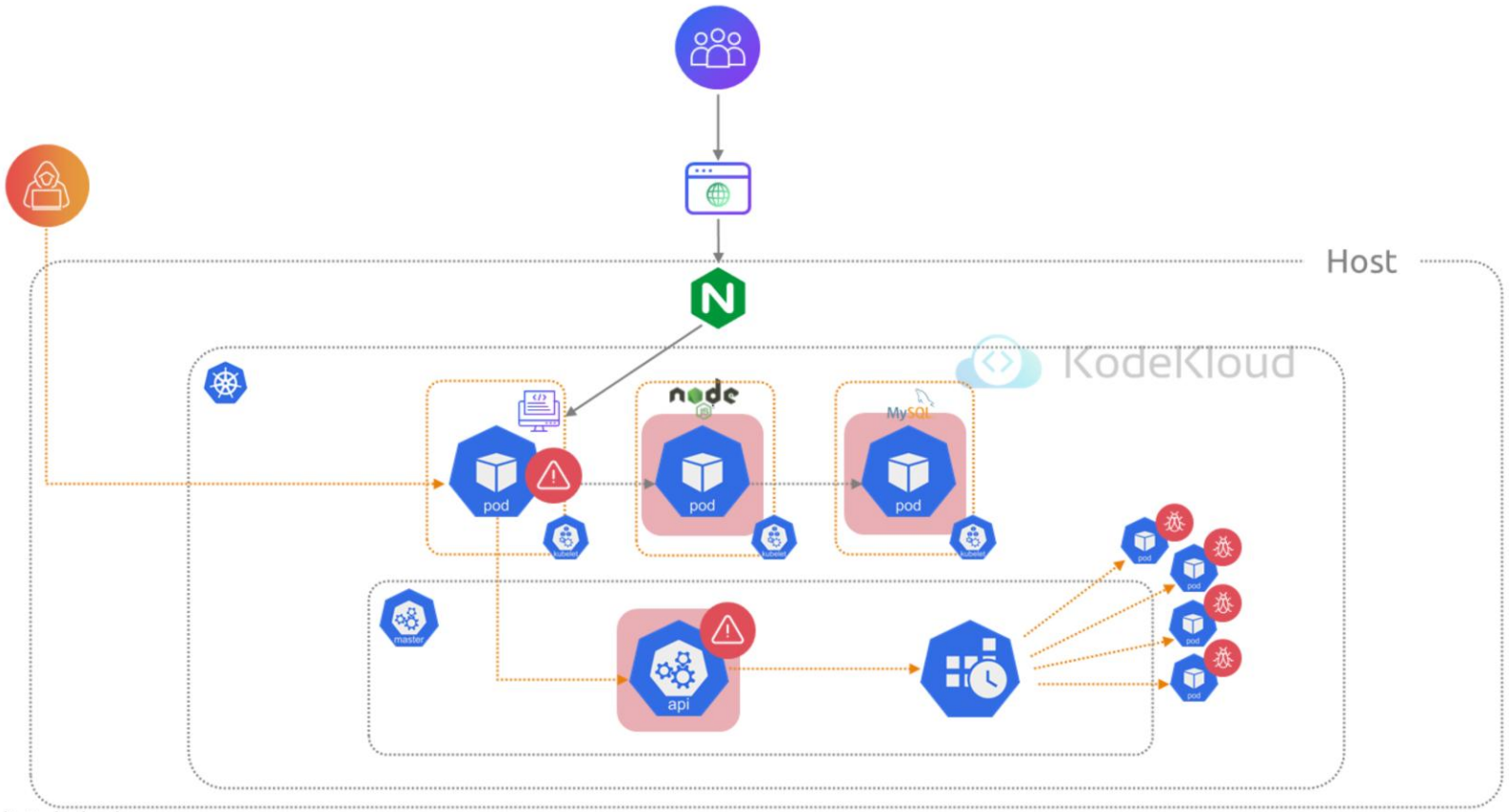
•Find Pod/Container with Mounted Docker Socket: For this, they look for containers with access to the Docker socket (/var/run/docker.sock), which can give them control over the Docker daemon and, by extension, the host.

Start Child Process in Host PID Namespace [M9]:

•If the container is running with HostPID enabled, the attacker can start processes that are part of the host's process

namespace. This means they can see and interact with host processes, potentially escalating their control.
•Find Pod/Container with HostPID: The attacker looks for any pods with HostPID enabled, which would let them interact with host processes and potentially control them.
Find Pod/Container with SYS_PTRACE Capability Enabled:
•Another way to interact with host processes is through the SYS_PTRACE capability, which allows tracing or debugging of processes. If a container has this capability, an attacker could use it to inspect or manipulate host processes.
•By finding a pod or container with SYS_PTRACE enabled, the attacker can effectively snoop on or even control processes running on the host, which could lead to further privilege escalation.

Compromised Application



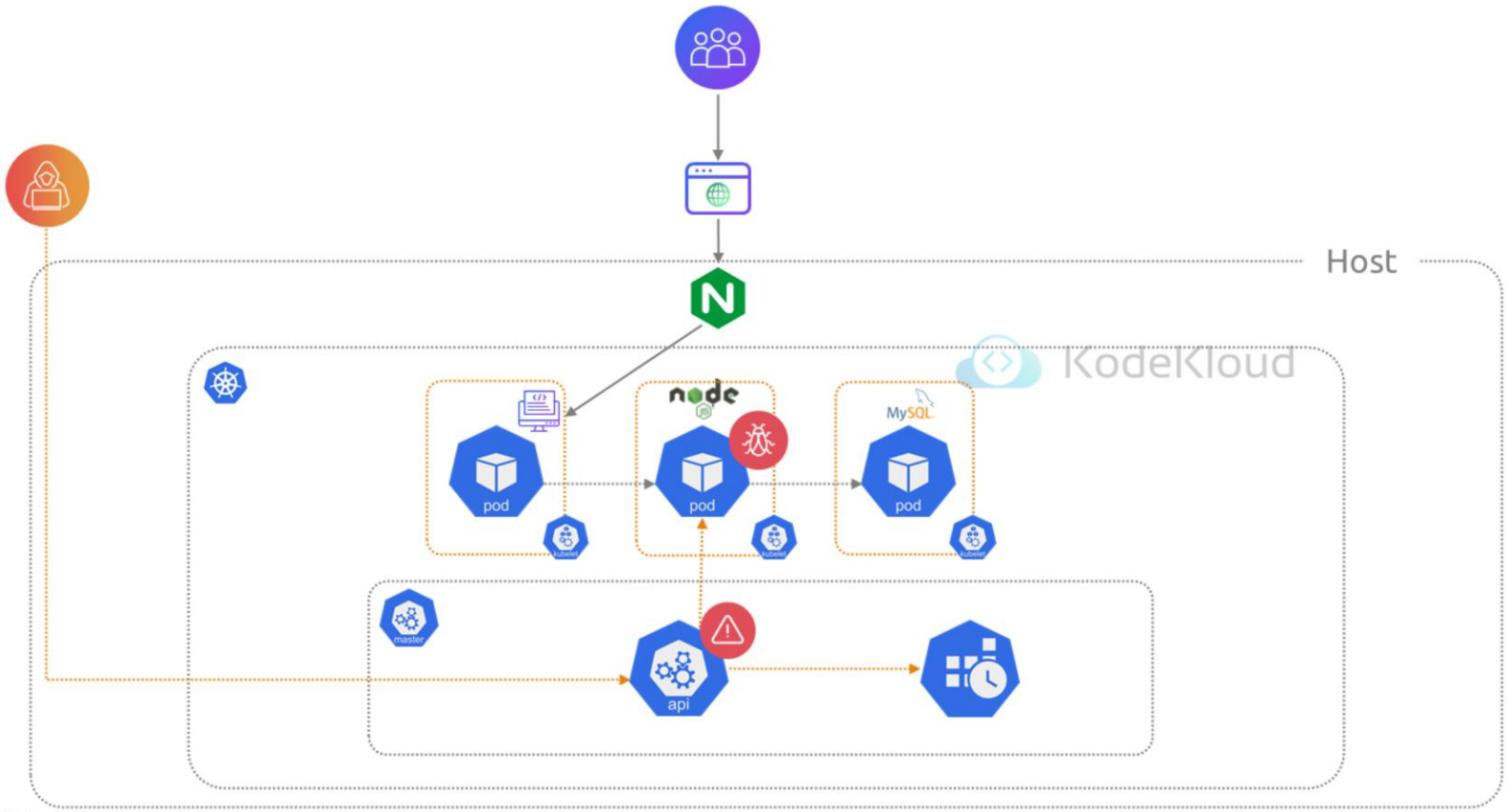
First Attack vector is Compromised Application.

Consider an attacker who exploits a vulnerability in our application's frontend server. By gaining access to the container running this server, the attacker can then attempt to load and execute additional malicious code.

For example, they might download and run scripts that further compromise backend services or the database. If they obtain sufficient privileges, they might even use this initial access to exploit the Kubernetes API server, gaining control over more

containers.

Abusing the API Server

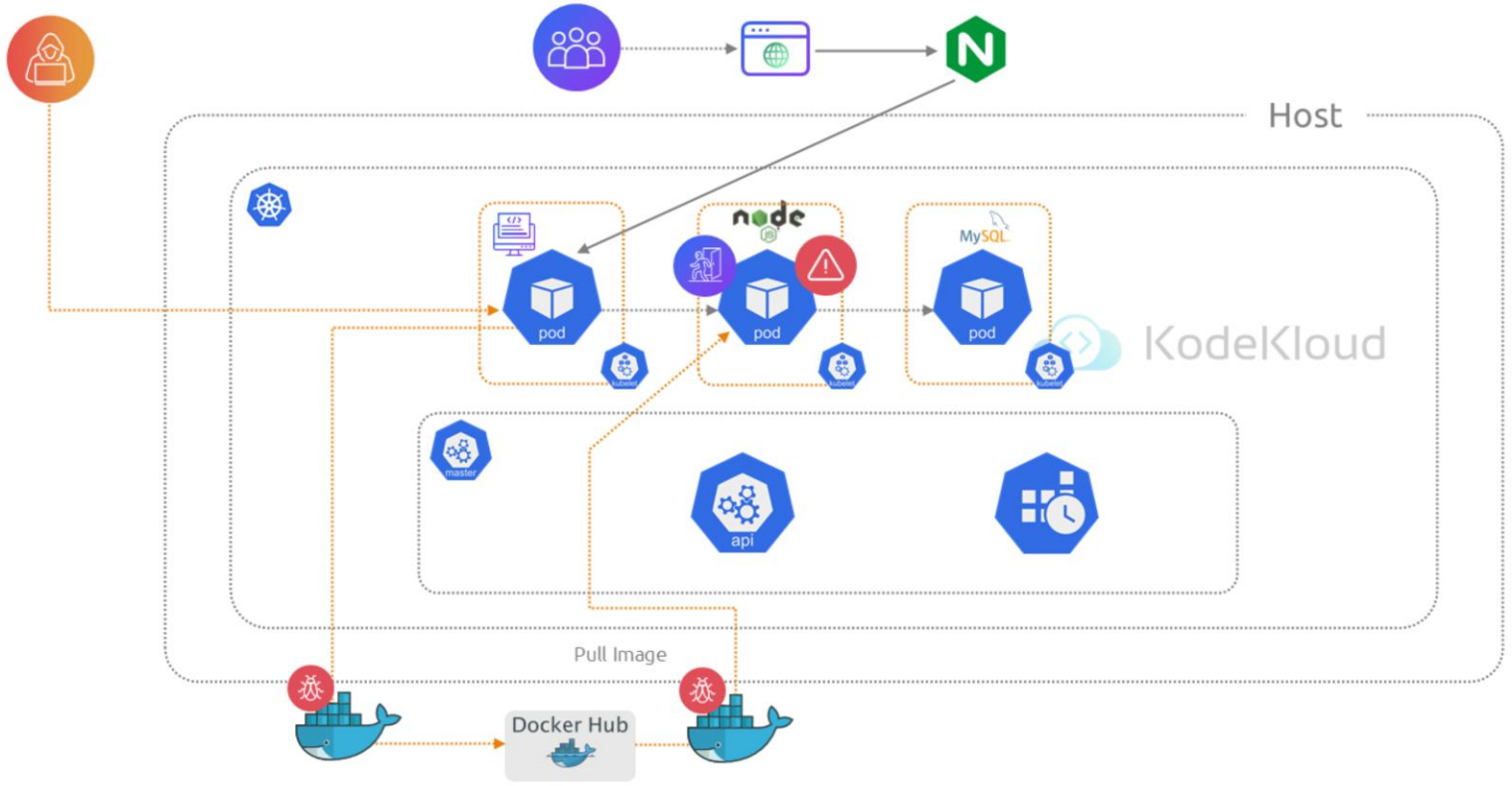


Next we'll move to Abusing the API Server.

Once inside our Kubernetes cluster, an attacker might target the API server to execute commands within running containers. With access to the API server, they can perform actions such as executing code directly in containers.

For example, the attacker could use the API server to execute commands in a backend container, installing malware that compromises data or steals sensitive information.

Poisoning the Image Repository



Then, Poisoning the Image Repository

If the attacker manages to get the Image pull secret, they could poison the container image repository by uploading malicious images. Other parts of the cluster might then pull and run these compromised images, spreading malicious code throughout the environment.

For instance, if a backend service pulls a new image containing a backdoor, the attacker can exploit this backdoor whenever

the container is redeployed.

Malicious Code Execution – Mitigating Risks

Scanning vulnerabilities

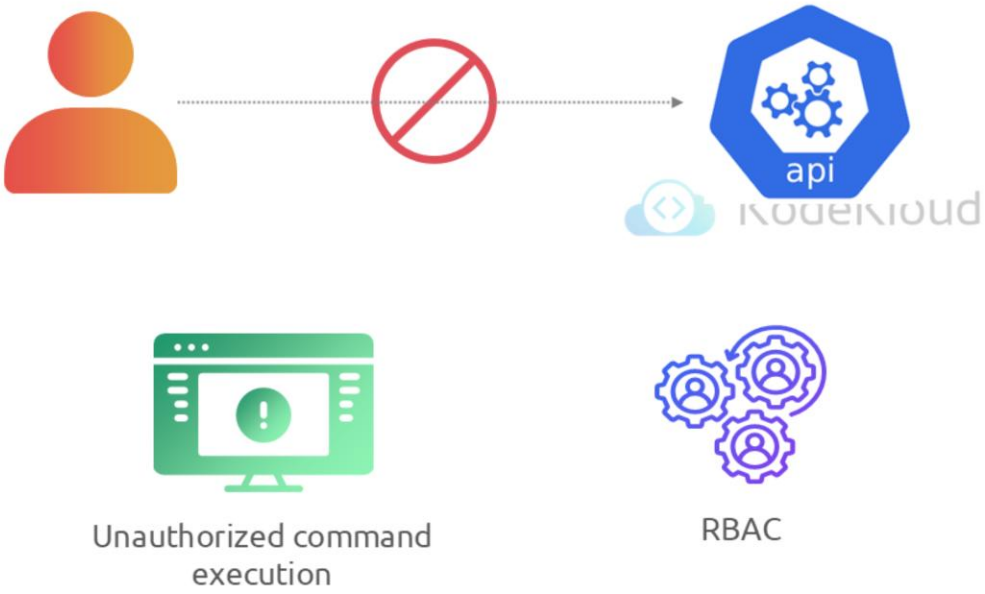
Applying patches

Updating servers and backend with security patches to reduce exploitation risk.

To protect our Kubernetes cluster from malicious code execution, we need to implement several security controls for our application.

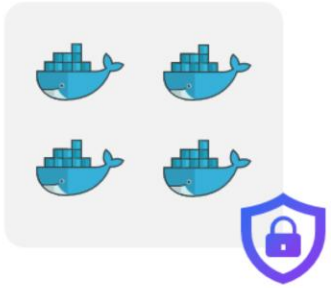
Ensuring our applications are secure by regularly scanning for vulnerabilities and applying patches could be a first step. This includes updating our web server and backend services with the latest security patches to reduce the risk of exploitation.

Malicious Code Execution – Mitigating Risks



Next, we should restrict access to the API server. By ensuring that only authorized users and services can interact with the API server, we can prevent unauthorized command execution. Implementing strong authentication and authorization mechanisms, such as Role-Based Access Control (RBAC), helps limit permissions to only what is necessary.

Malicious Code Execution – Mitigating Risks



Secure image pull secrets and limit access to necessary pods

Signed images verify the integrity and authenticity

```
kubectl create secret docker-registry my-image-
pull-secret \
  --docker-username=<username> \
  --docker-password=<password> \
  --docker-server=<registry-url> \
  --namespace=default
```

```
apiVersion: v1
kind: Pod
metadata:
  name: my-secure-pod
  namespace: default
spec:
  serviceAccountName: specific-service-account
  containers:
  - name: myapp
    image: <registry-url>/myapp:latest
  imagePullSecrets:
  - name: my-image-pull-secret
```


After that, we need to protect our container image repositories. Securing the Image pull secrets and ensuring that only trusted sources can upload images is essential. We should store Image pull secrets securely and restrict their access to only necessary pods. Using signed images helps verify the integrity and authenticity of the images we pull from the repository.

Malicious Code Execution – Mitigating Risks

Prometheus

Grafana


```yaml
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: security-monitoring-rules
  namespace: default
spec:
  groups:
  - name: security-alerts
    rules:
    - alert: SecretChangeDetected
      expr: kube_secret_info
      for: 1m
      labels:
        severity: critical
      annotations:
        summary: "Secret Change Detected"
        description: "A change was detected in a Kubernetes Secret, which
could indicate unauthorized access or tampering."

    - alert: CommandExecutionInContainer
      expr: rate(container_cpu_usage_seconds_total{image!="",
name=~"exec.*"}[5m]) > 0
      for: 1m
      labels:
        severity: warning
      annotations:
        summary: "Command execution detected in container"
        description: "A command execution was detected in container {{
$labels.container }}. This might indicate suspicious activity."
```

Additionally, setting up robust monitoring and logging helps detect and respond to malicious activities quickly. Monitoring access to the API server, changes in Image pull secrets, and the execution of commands within containers can alert us to suspicious activities. Tools like Prometheus and Grafana are useful for this purpose.

For example here we have a SecretChangeDetected alert: This Alerts when a Kubernetes Secret is changed, which could indicate unauthorized modifications. And another alert CommandExecutionInContainer: Detects command executions within

containers by observing CPU usage spikes associated with specific command executions.

Malicious Code Execution – Mitigating Risks

Auditing and Reviewing

- Permissions granted to service accounts
- Security of the image repositories
- Access controls for API server

Finally, regularly auditing and reviewing our security configurations and practices ensures they remain effective. We should periodically check the permissions granted to service accounts, the security of our image repositories, and the access controls for the API server.

Summary

01 Attackers exploit vulnerabilities in containers to execute malicious code

02 Restrict API server access to authorized users and services only

03 Secure image repositories and use signed images for verification

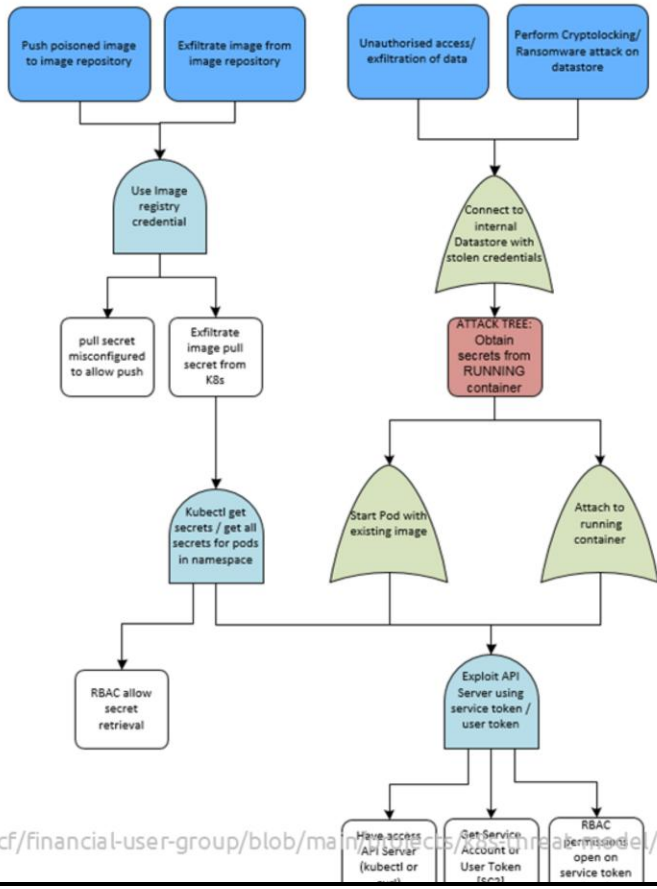
04 Monitor and log activities to detect and respond to threats

05 Regularly update and patch applications to prevent security exploits

As a start, Let's understand how a Compromised Application Leading wat to Container Access

Suppose an attacker exploits a vulnerability in our Node.js backend service. Then, They gain access to the container running this service. With access to the container, the attacker can explore further avenues of attack within the Kubernetes cluster.

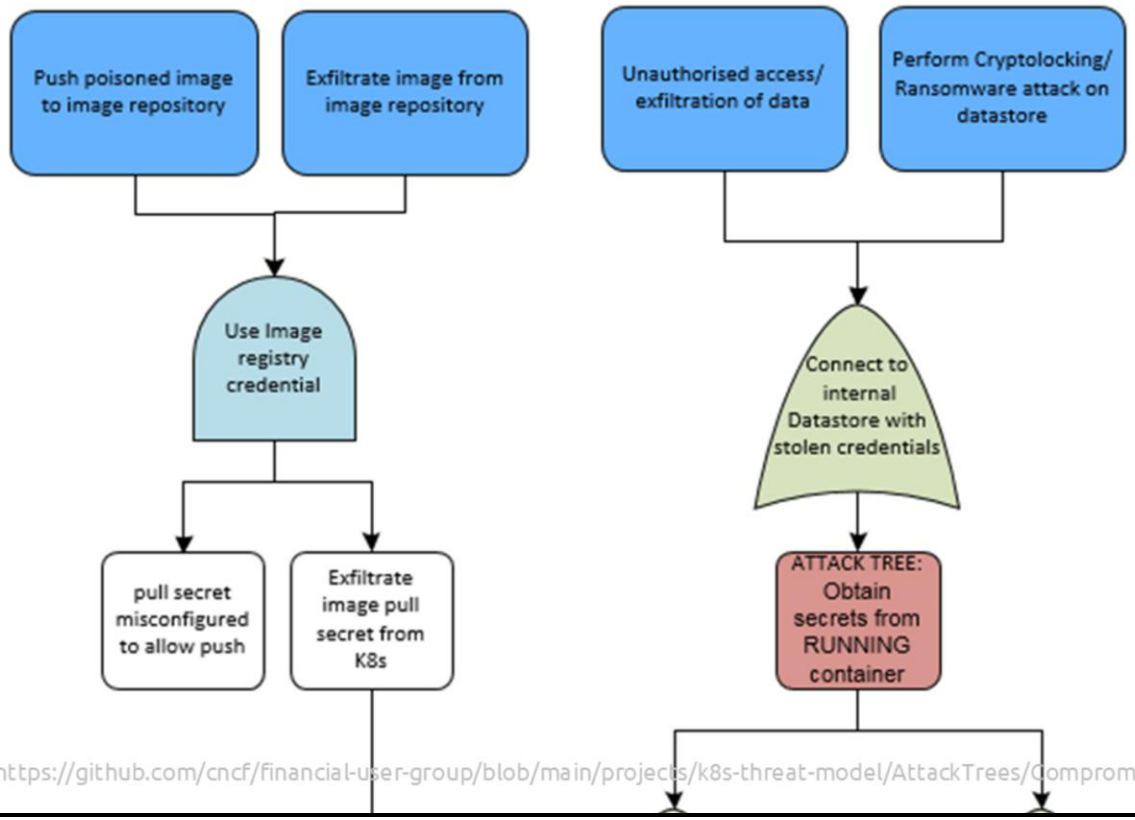
Compromised Container AttackTree



https://github.com/cncf/financial-user-group/blob/main/topics/AttackThreatModel/AttackTrees/CompromisedContainer.md

Let's look at the Compromised Container Attack Tree.

Compromised Container AttackTree



https://github.com/cncf/financial-user-group/blob/main/projects/k8s-threat-model/AttackTrees/CompromisedContainer.md

WE'll look at how it breaks down into 4 areas –

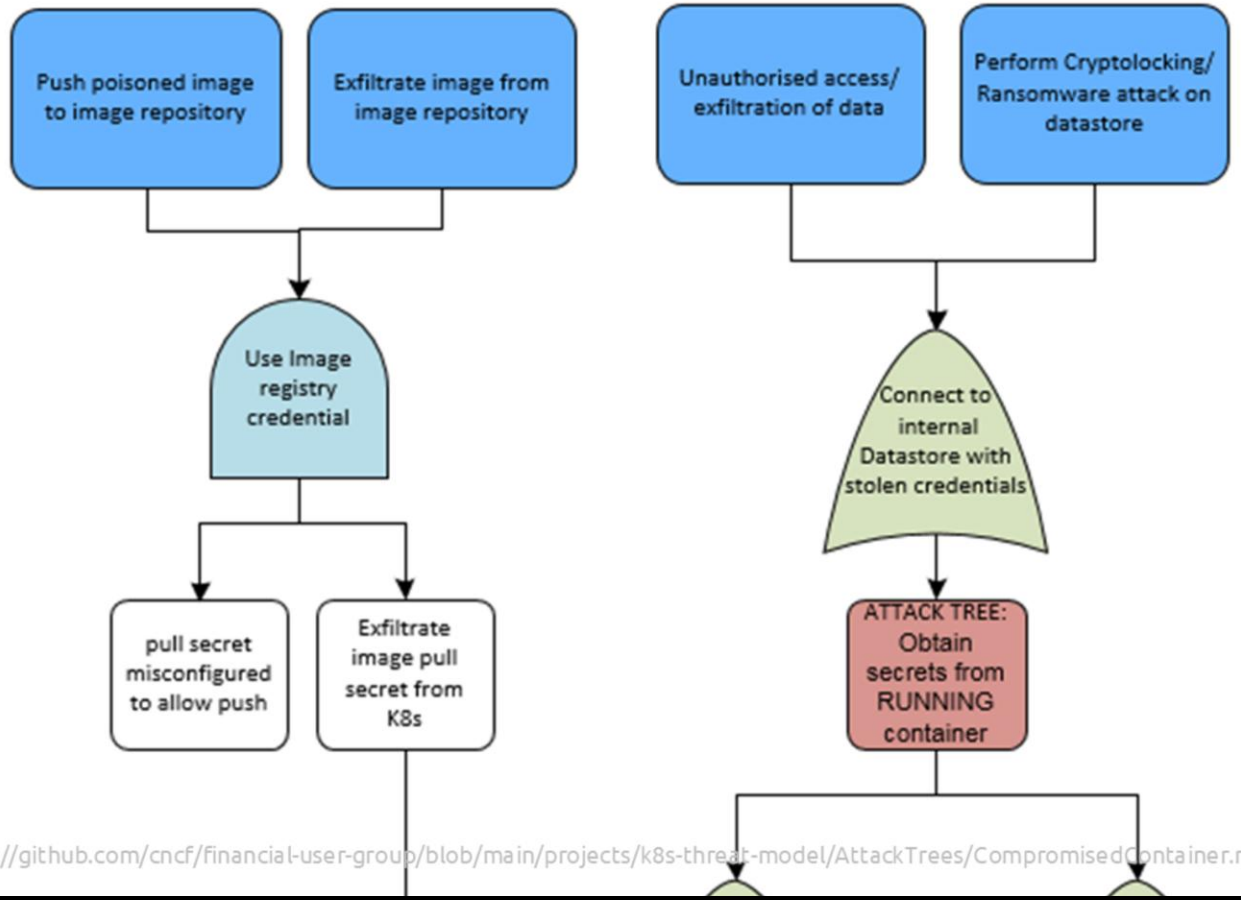
•An attacker could inject malicious code into a container image and then push this "poisoned" image to your image repository.

If an attacker gains unauthorized access to your image repository, they could pull and examine your images, potentially finding sensitive information or security flaws.

Once they compromise a container, attackers may try to access or exfiltrate sensitive data.

With control over a compromised container, an attacker might launch a ransomware attack by encrypting or locking up data in connected datastores.

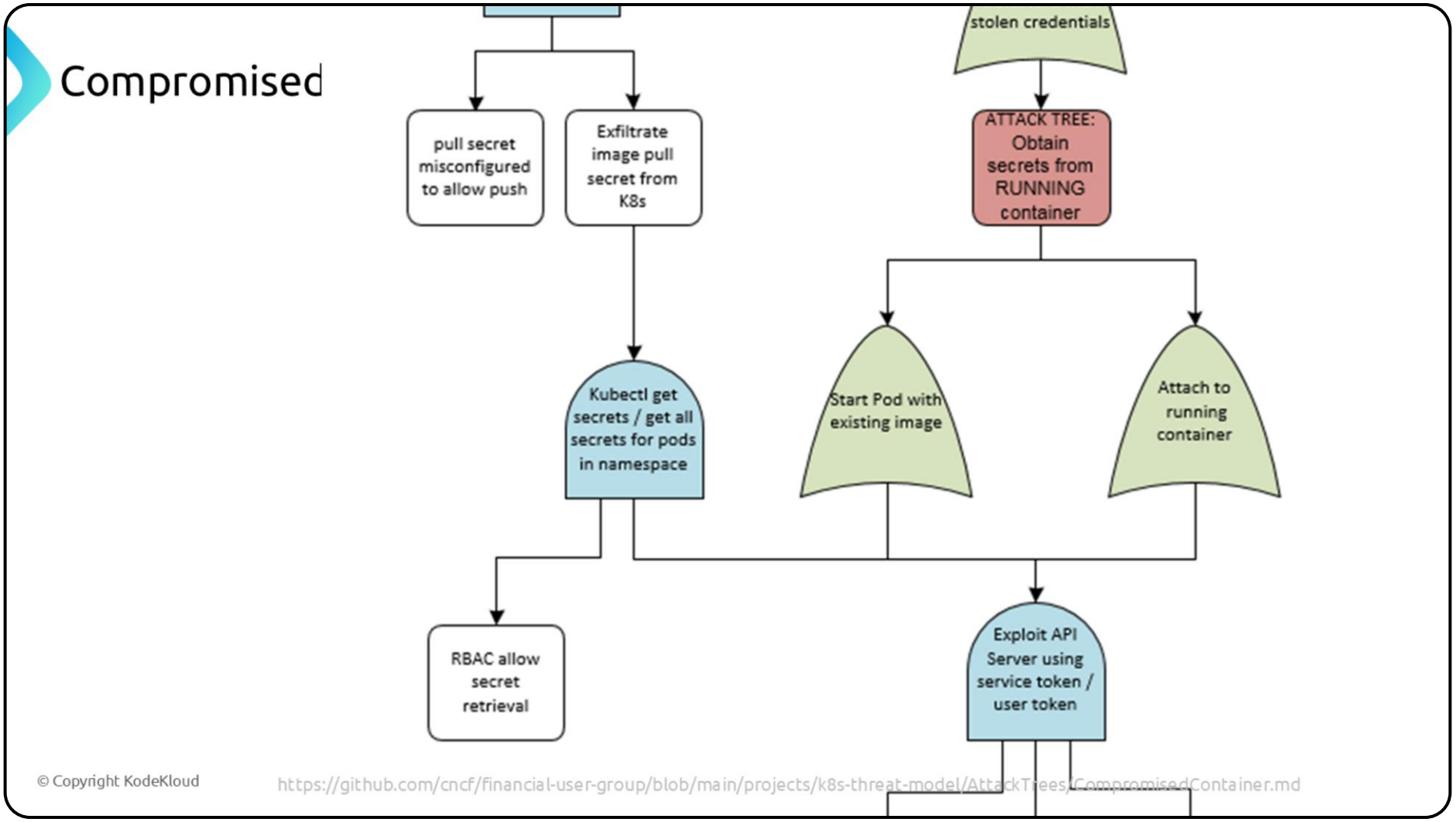
Compromised Container AttackTree



https://github.com/cncf/financial-user-group/blob/main/projects/k8s-threat-model/AttackTrees/CompromisedContainer.md

In both the first two cases The attacker leverages credentials (Image Pull Secrets) intended for accessing the image repository. If they manage to get their hands on these credentials, they can either push malicious images to the repository or pull existing images out of it. Access to these credentials gives the attacker a lot of power over your image repository.

Sometimes, an Image Pull Secret may be misconfigured to allow both pulling and pushing images. This misconfiguration could allow an attacker not only to download images but also to upload their own malicious versions.
In this scenario, the attacker finds a way to extract an Image Pull Secret from a Kubernetes environment.



Objective: Obtain Secrets from Running Container:

•The attacker's primary goal here is to access sensitive information (like environment variables, API tokens, or database credentials) from within a running container.

Path 1: Start Pod with Existing Image:

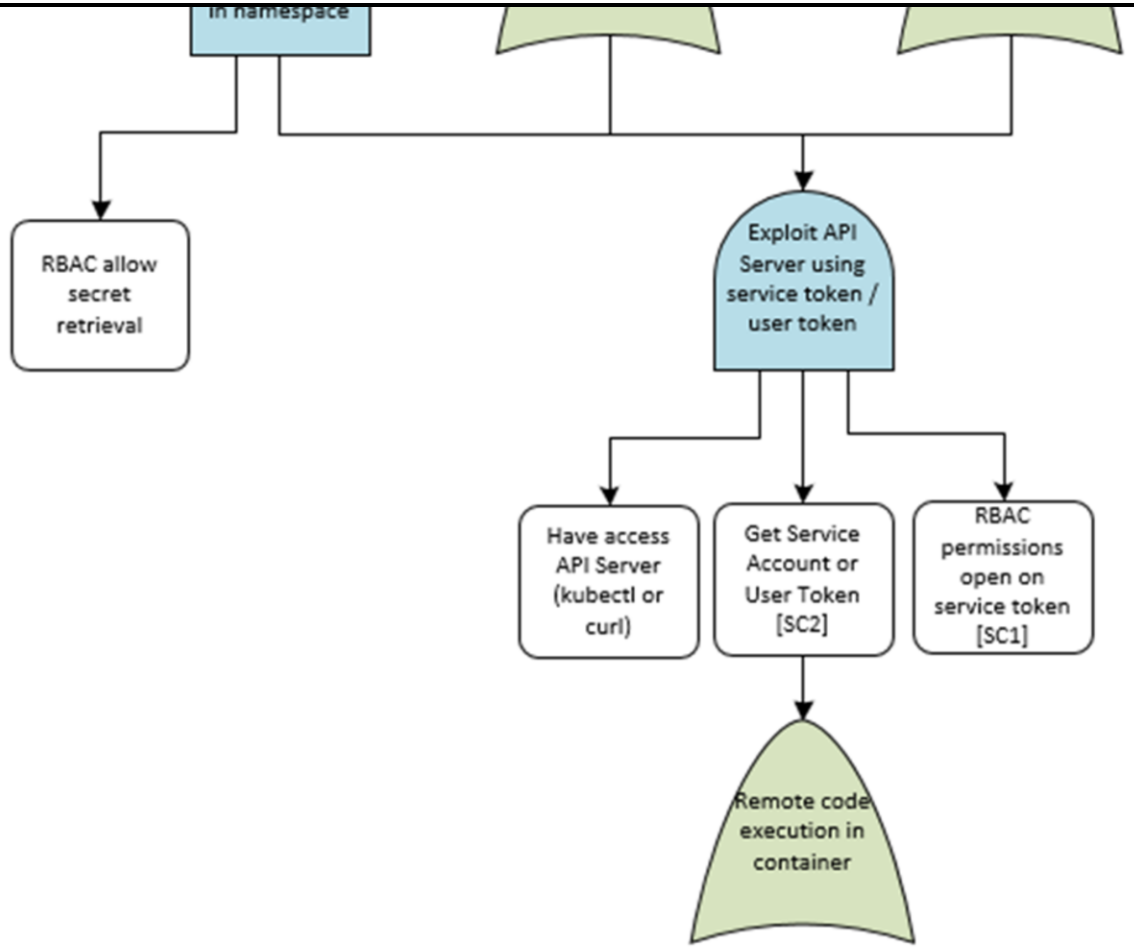
•The attacker might start a new pod using an existing image that's already configured with secrets. By doing this, they can create a replica of a sensitive environment, allowing them to access the same secrets without needing to break into an already running pod.

•Another approach is to directly attach to an already running container. If the attacker gains access to a running container, they can inspect its environment and extract secrets directly.

Exploit API Server Using Service Token/User Token:

•After gaining access to the container, the attacker could further exploit the Kubernetes API server using the container's service account token or a user token available within the container. By leveraging these tokens, they can escalate their privileges within the cluster, potentially accessing more secrets or sensitive configurations

Compromised



https://github.com/cncf/financial-user-group/blob/main/projects/k8s-threat-model/AttackTrees/CompromisedContainer.md

•Example Use Case: By accessing the API server with a token, they could list secrets, inspect configurations, or even manipulate other resources within the cluster.

Summary

01 Attackers exploit vulnerabilities to access and control containers

02 Avoid mounting service account tokens unless absolutely necessary

03 Limit service account permissions to minimize potential damage

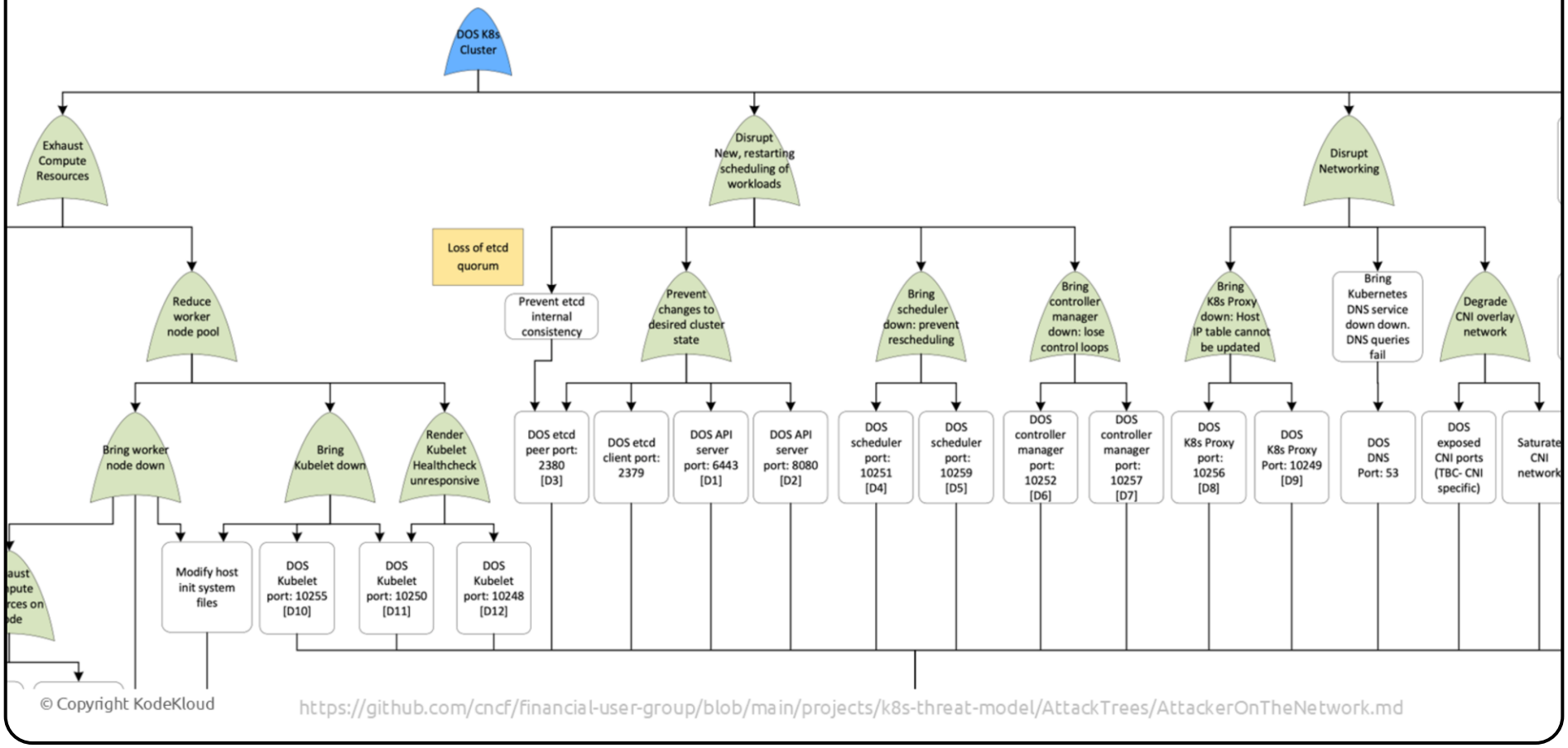
04 Implement network policies to restrict container communication paths

05 Use RBAC to enforce strict access controls in Kubernetes

06 Monitor and log activities to detect and respond to threats

Attacker on the Network

Compromised Network AttackTree



This Network Attack Tree shows us different ways an attacker could cause chaos in a Kubernetes cluster by launching a Denial of Service (DoS) attack on various network components and control plane services. Let's go through the main attack paths:

1.Exhaust Compute Resources:

1. Here, the attacker's goal is to overload the cluster's compute power, especially the worker nodes and kubelets.
2. They could reduce the worker node pool by knocking out worker nodes one by one. This might be done by overwhelming kubelets or messing with critical files on the nodes, making the nodes unresponsive or unhealthy.
3. By targeting kubelets specifically, they could prevent nodes from reporting their health status, effectively taking

them out of the pool and reducing the cluster's ability to handle workloads.

2. Mess with etcd (Cluster Brain):
    1. etcd is like the brain of Kubernetes—it’s the place where all the important state information is stored. If etcd goes down or loses consistency, the whole cluster gets confused.
    2. The attacker could cause a loss of etcd quorum by overloading its ports (2380 and 2379) or by blocking it from syncing up with other components. This would prevent the cluster from understanding its own state.
    3. They could also go after the API server ports (6443 and 8080). The API server is essential for cluster operations, so overloading these ports stops legitimate access, making it hard to manage anything in the cluster.

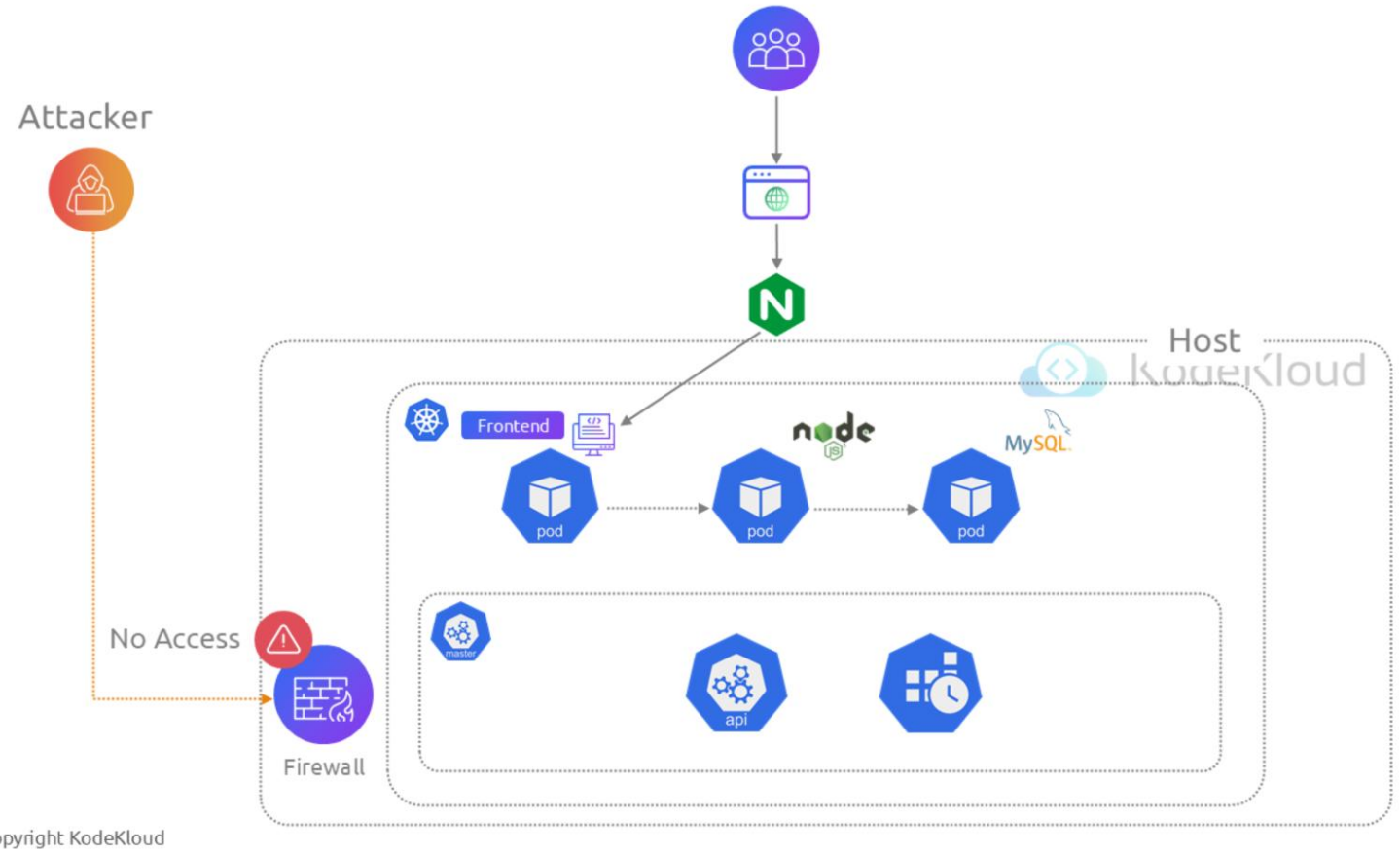
3.Stop Scheduling of New or Restarted Workloads:
    1. By taking down the scheduler and controller manager, the attacker can halt new or restarted workloads from being scheduled in the cluster.
    2. Attacking the scheduler ports (10251 and 10259) prevents Kubernetes from assigning pods to nodes, stopping new tasks from starting.
    3. Going after the controller manager ports (10252 and 10257) disrupts critical control loops, which impacts scaling, updates, and replication. In other words, Kubernetes won’t be able to manage the cluster’s state effectively.

4.Disrupt Networking:
    1. This path is all about messing with internal and external network communications in the cluster.
    2. K8s Proxy is the service that manages network rules on each node. If the attacker takes down the K8s Proxy or overloads its ports (10256 and 10249), it stops the flow of traffic between services and pods, essentially freezing communication.
    3. Targeting Kubernetes DNS (port 53) would break name resolution, meaning services can’t find each other by name, causing connectivity issues all over.
    4. Finally, they could degrade the CNI (Container Network Interface) overlay network by flooding it. This slows down or cuts off pod-to-pod communication, causing distributed services to fail.

5.Prevent New Worker Nodes from Joining:
    1. By targeting network boot services (like PXE servers), the attacker could prevent new nodes from

joining the cluster. This keeps the cluster from scaling up or replacing failed nodes, locking it into a degraded state.

Configuring Firewalls



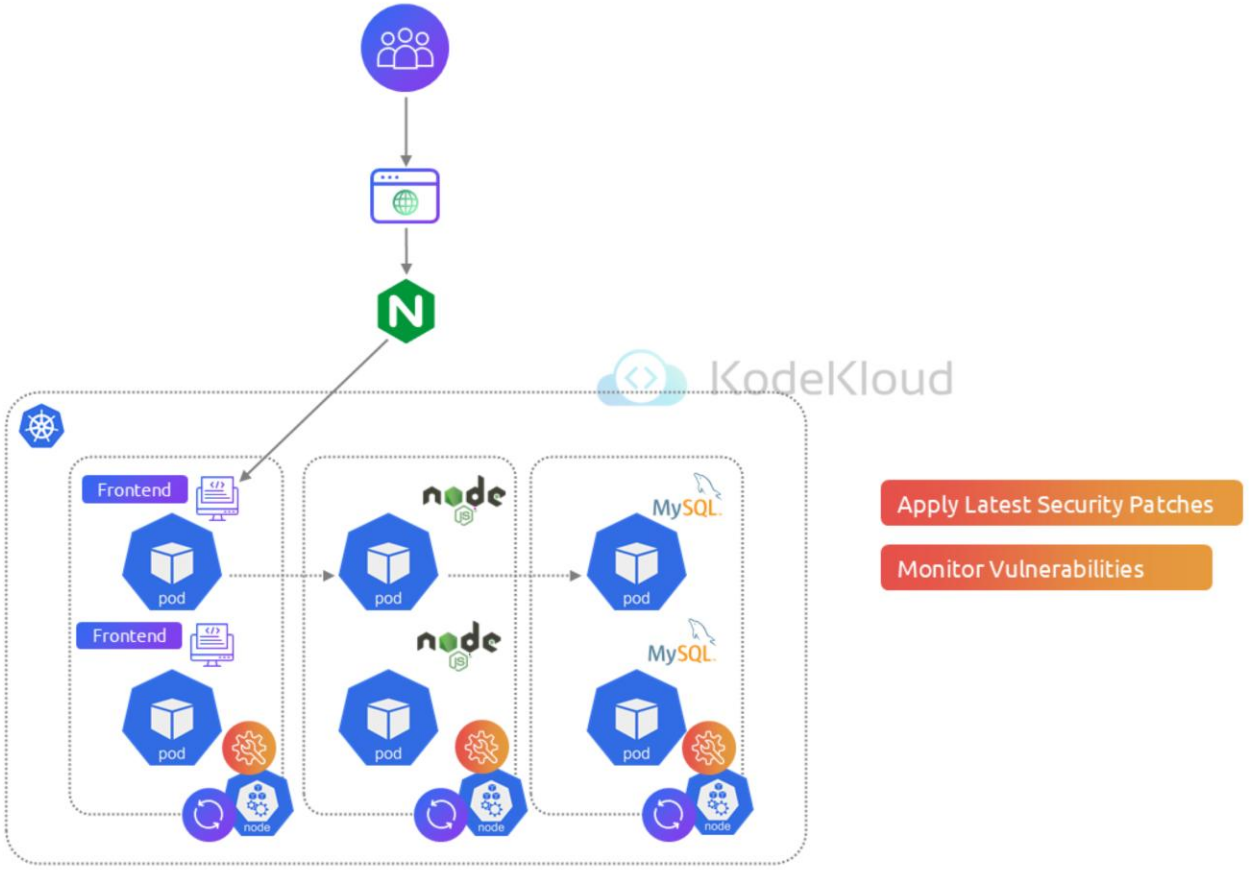
How to Mitigate these type of network-based attack risks then?

One solution is Configuring Firewalls.

We configure firewalls to limit network access to the control plane and nodes. By setting up firewall rules to restrict access to

the API server and allowing only trusted IP addresses to communicate with it, we prevent unauthorized access and reduce the risk of denial-of-service attacks.

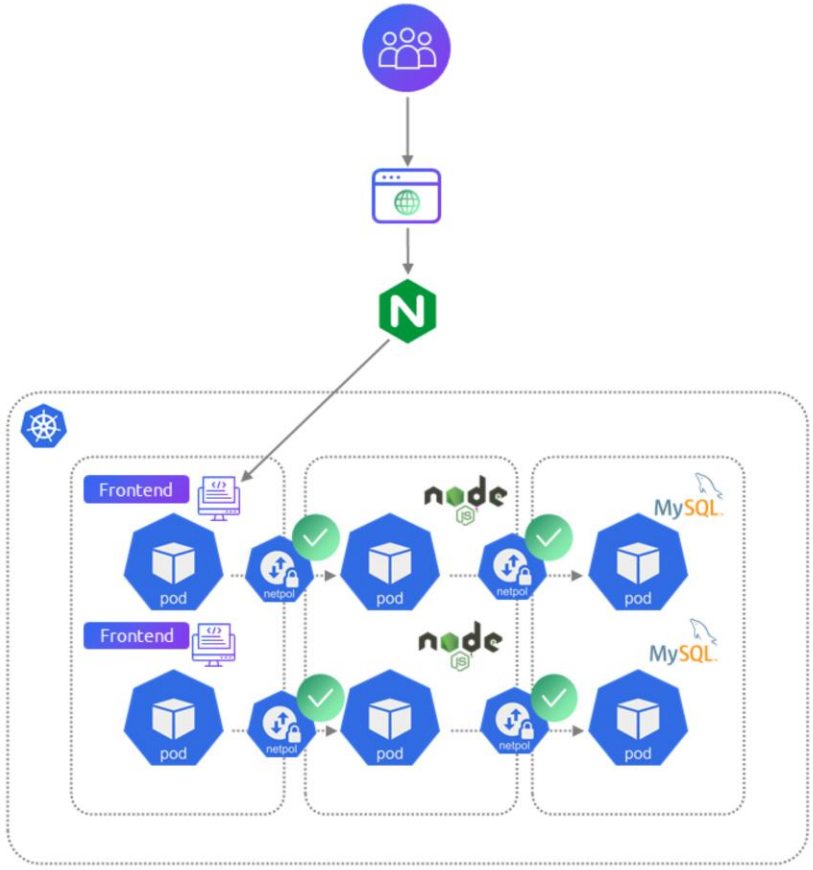
Securing Nodes



Another way is Securing Nodes.

Keeping the operating systems and Kubernetes components on our nodes up to date with the latest security patches is crucial. Regularly updating and patching the systems helps mitigate vulnerabilities that attackers might exploit. Ensuring that the nodes running our Node.js backend service are regularly patched and monitored for vulnerabilities is essential.

Implementing Network Policies

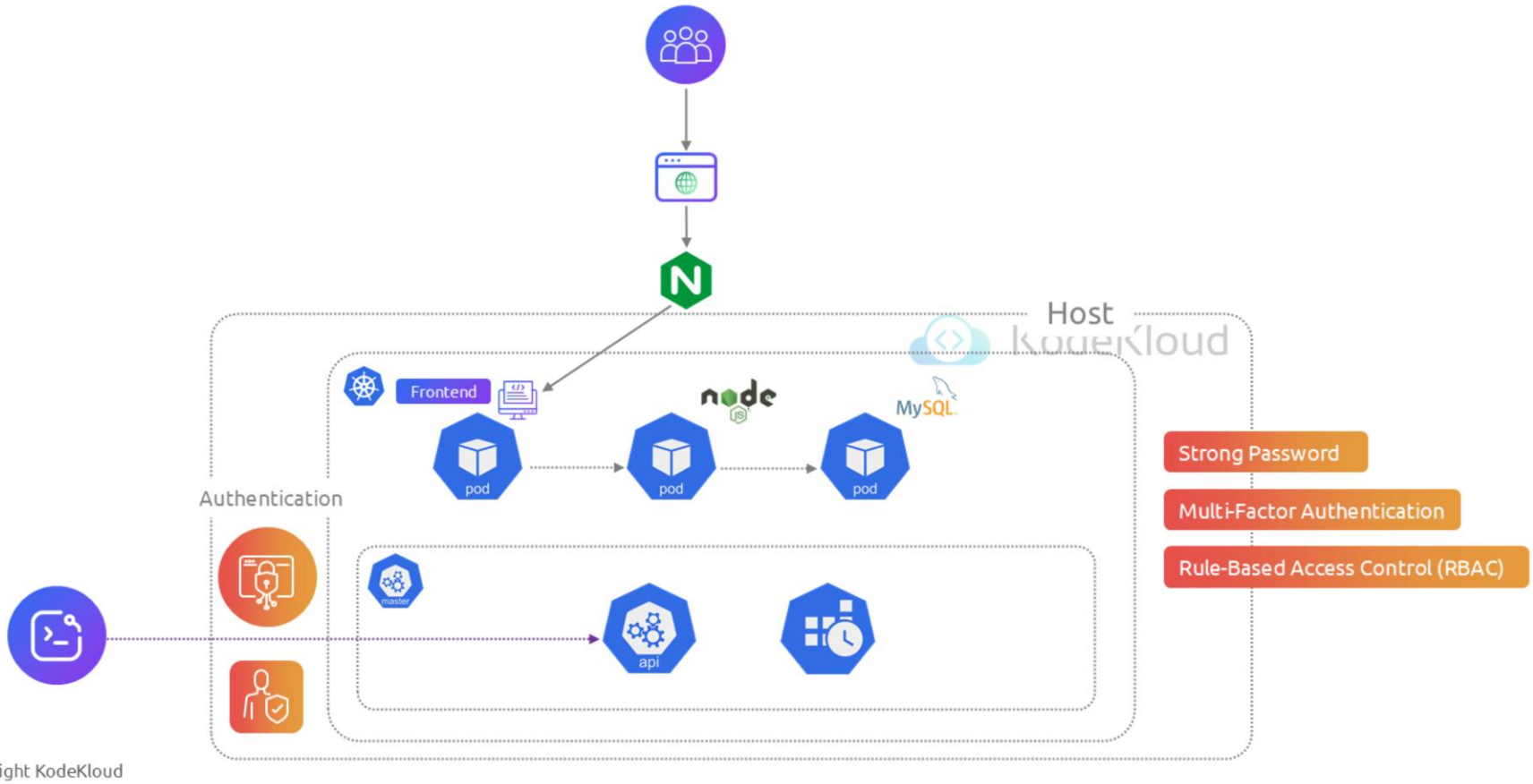



```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-frontend-to-backend
  namespace: frontend
spec:
  podSelector: {}
  policyTypes:
    - Egress
  egress:
    - to:
        - namespaceSelector:
            matchLabels:
              name: backend
          podSelector: {}
```

As always, Implementing Network Policies is crucial here as well.

With Network policies in place, We can restrict network communication between the frontend, backend, and database components, limiting the attacker's ability to move from one compromised node to another.

Strong Authentication and Authorization



Another important approach is having, Strong Authentication and Authorization.

Enforcing strong authentication and authorization mechanisms is vital to protect access to the API server and other critical components. Using strong passwords, multi-factor authentication, and Role-Based Access Control (RBAC) limits access to only necessary operations, ensuring that only authorized users can perform sensitive actions.

Monitoring and Logging

Monitoring

Helps detect unusual access patterns

Logging

Ensures a quick response to potential threats


```yaml
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: network-alert-rules
  namespace: default
spec:
  groups:
  - name: network-alerts
    rules:
    - alert: HighAPIRequests
      expr: sum(rate(apiserver_request_total[5m])) by (client) > 100
      for: 5m
      labels:
        severity: warning
      annotations:
        summary: "High rate of API server requests detected"
        description: "There is a high rate of API server requests from {{
$labels.client }}. This could indicate unusual activity."

    - alert: HighNetworkTraffic
      expr: sum(rate(container_network_receive_bytes_total[5m]) +
rate(container_network_transmit_bytes_total[5m])) by (pod) > 10000000
      for: 5m
      labels:
        severity: critical
      annotations:
        summary: "High network traffic detected in pod"
        description: "The pod {{ $labels.pod }} is generating unusually
high network traffic, which may indicate an attack or data exfiltration."
```

Finally as always, Monitoring and Logging important.

Monitoring and logging are essential for detecting unusual network activities and responding quickly to potential attacks. Tools like Prometheus and Grafana help monitor network traffic and alert us to any suspicious activities. Setting up monitoring to detect unusual access patterns and logging all access attempts ensures a quick response to potential threats.

In the example shown here HighAPIRequests: alert fires if a client is making more than 100 API requests per second on average over 5 minutes, which might indicate a DoS attack or suspicious automation. And the HighNetworkTraffic: alert triggers if a pod is receiving or transmitting more than 10MB of network traffic per second, indicating potential data exfiltration or network abuse.

Summary

01 Attackers can target Kubernetes control plane and nodes for breaches

02 Configure firewalls to limit network access to trusted IP addresses

03 Keep node operating systems and components updated and patched

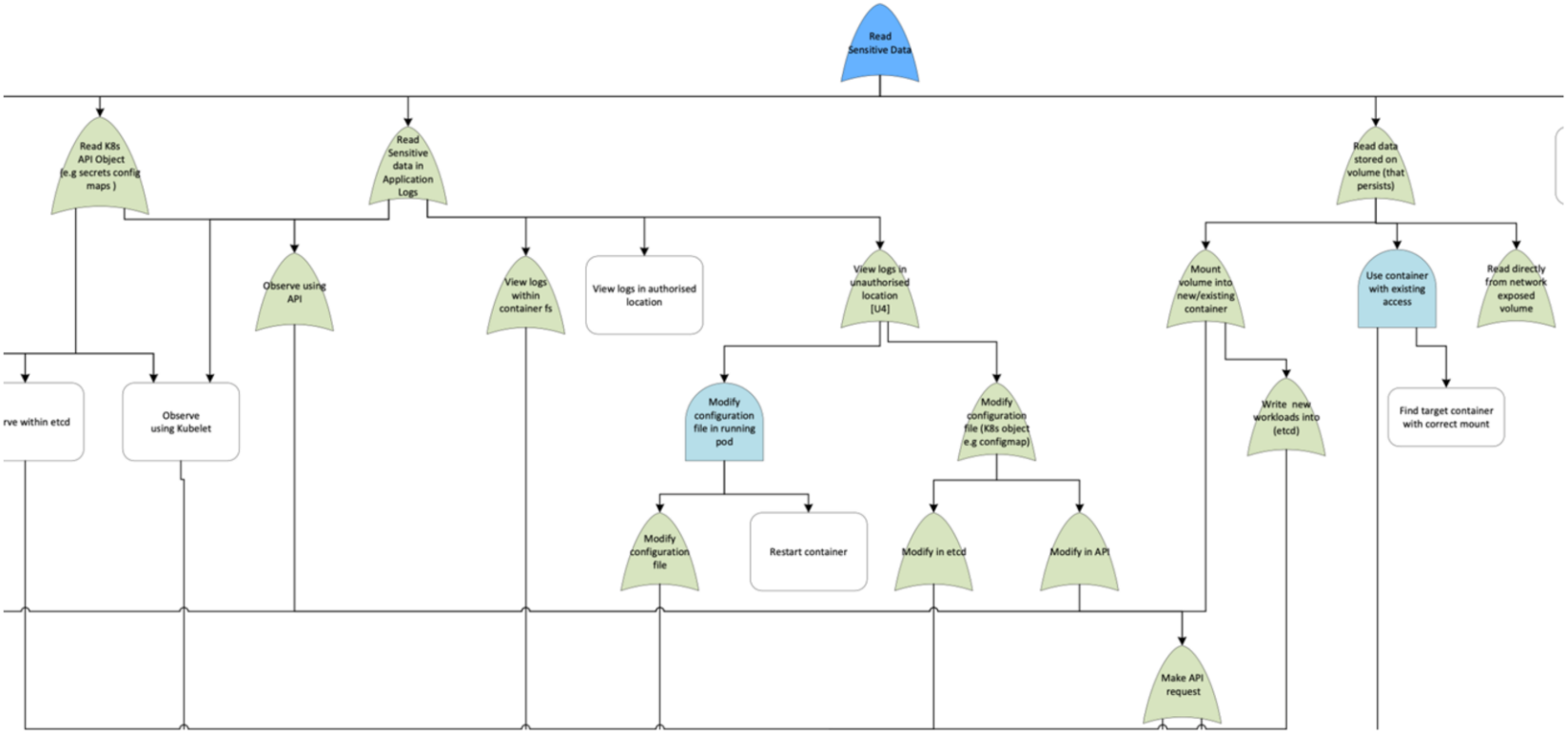
04 Implement network policies to control traffic and prevent lateral movement

05 Use strong authentication, multi-factor, and RBAC for secure access

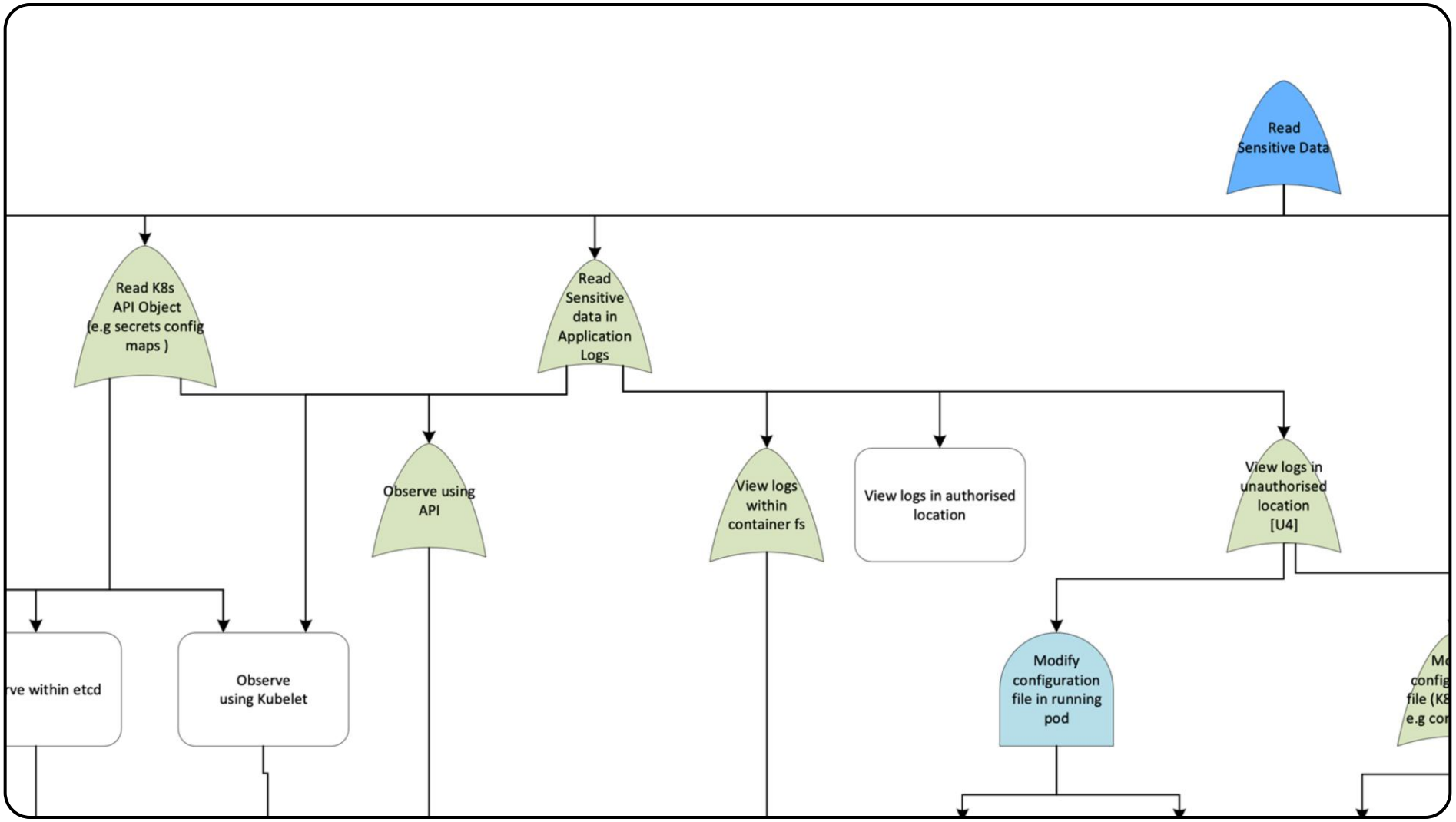
06 Monitor and log activities to detect and respond to threats

Access to Sensitive Data

Access Sensitive Data Attack Tree



Let's take a look at the Access to Sensitive Data Attack Tree.



This is all about the different tricks an attacker might use to get their hands on sensitive information in a Kubernetes setup.

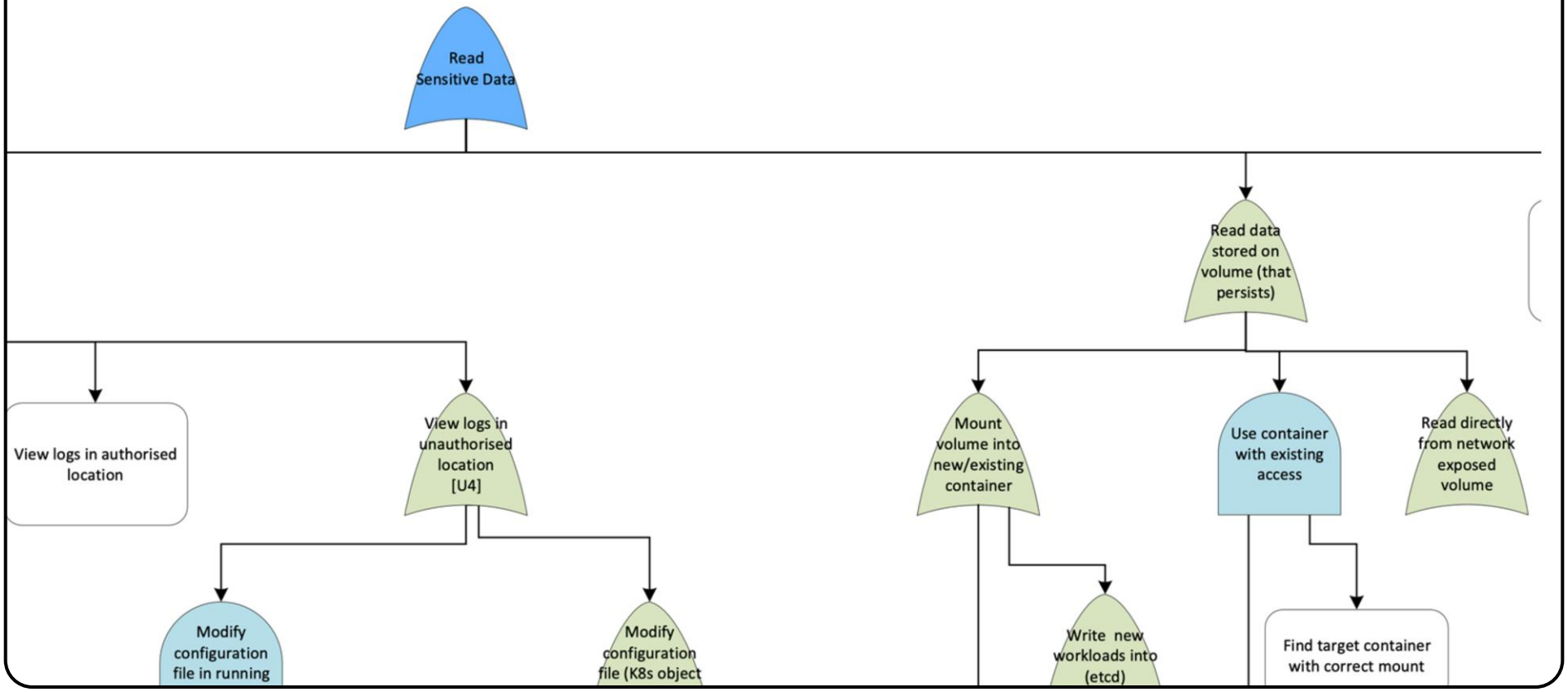
Getting to Kubernetes Secrets:
•The attacker could try to peek at secrets or config maps stored by Kubernetes, which might include passwords, API tokens, or other sensitive bits.
•They can go straight to etcd (the cluster's main data store) or try accessing the kubelet (which manages containers on each node). Both can potentially reveal secrets if not locked down.

Diving into Application Logs:

•Apps sometimes log way more info than they should, and attackers know this. If they can get inside a container, they might find logs that accidentally reveal sensitive data.

•Sometimes logs get stored in places they shouldn’t, or they can mess with config files to change where logs get saved, making them easier to access.

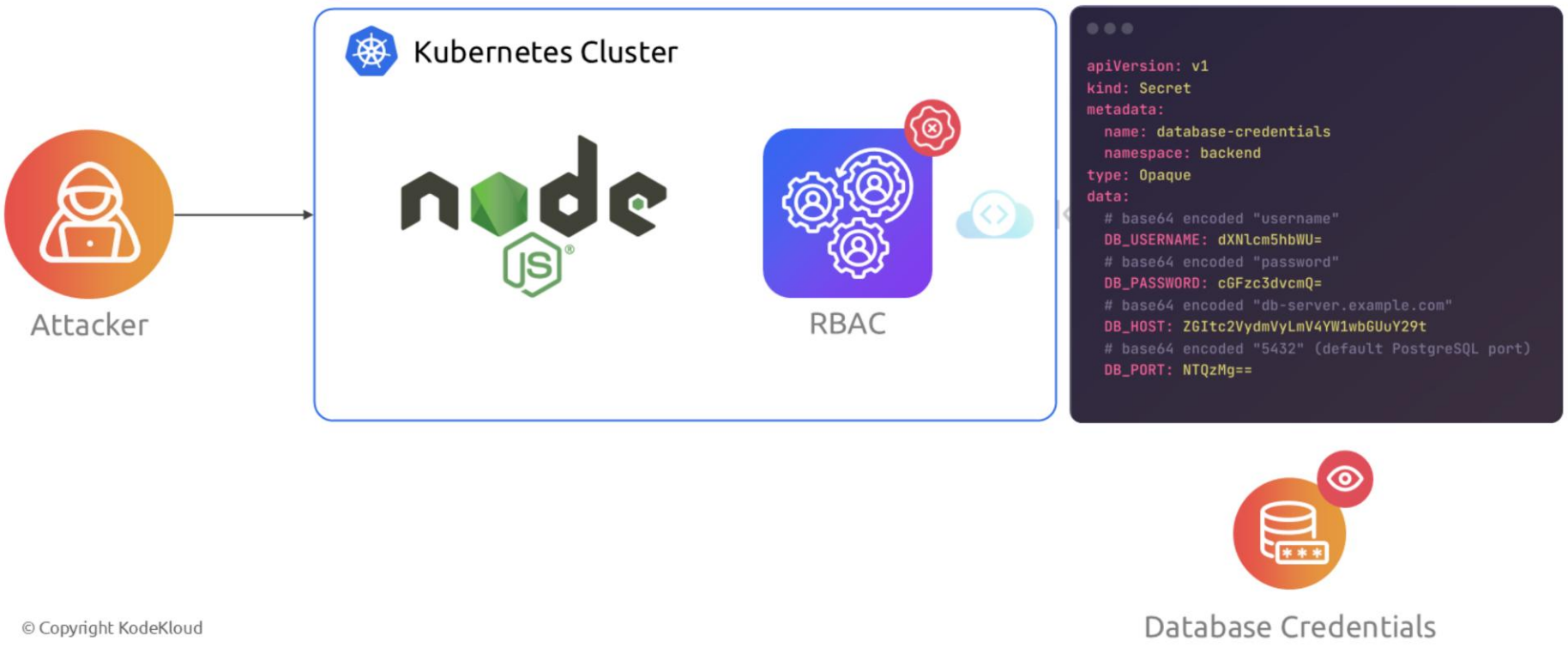
Access Sensitive Data Attack Tree



Reading Persistent Data:
•Some data sticks around in persistent volumes, which can be a goldmine for attackers.
•They could mount the volume to a new container they control or find a container that already has access to the volume and read it from there.
Accessing Exposed Volumes:
•If a volume is exposed over the network, they could skip all the container hassle and just read the data directly.
Going for the Cluster's PKI:

•If they manage to compromise the cluster’s keys, they basically have free access to anything encrypted in the cluster.

Exploiting Misconfigured RBAC Permissions



Starting with, Exploiting Misconfigured RBAC Permissions.

In our application, let's say we have a Node.js backend pod. If this pod has excessive permissions due to misconfigured Role-Based Access Control (RBAC), an attacker could exploit these permissions to read secrets stored in the cluster.

These secrets might include database credentials or API keys, allowing the attacker to access sensitive information directly from the cluster.

Mitigating Risks of Accessing Sensitive Data


```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: backend-read-only
  namespace: backend
rules:
  - apiGroups: [""]
    resources: ["configmaps", "secrets"]
    verbs: ["get", "list"]
  - apiGroups: [""]
    resources: ["pods"]
    verbs: ["get", "list", "watch"]
```

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: backend-rolebinding
  namespace: backend
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: backend-read-only
subjects:
  - kind: ServiceAccount
    name: backend-sa
    namespace: backend
```


To protect our Kubernetes cluster from unauthorized access to sensitive data

we must ensure RBAC permissions are correctly configured. In our application, we review and tighten the RBAC policies for service accounts used by our backend Node.js pods. By ensuring these service accounts only have the permissions they need, we reduce the risk of attackers exploiting excessive permissions to access sensitive data.

Viewing Sensitive Data in Logs

```
[2024-11-12T10:15:25.345Z] [INFO] Connected to the database at db-
server.example.com:5432
[2024-11-12T10:15:26.456Z] [INFO] Executing query: SELECT * FROM users WHERE email =
'user@example.com' AND password = 'superSecretPassword123'
[2024-11-12T10:15:27.567Z] [ERROR] Database error: Connection timed out while accessing
db-server.example.com with username db_user and password dbPass123!
[2024-11-12T10:15:28.678Z] [DEBUG] API request payload: {"creditCardNumber":
"4111111111111111", "expiration": "12/25", "cvv": "123"}
```

Configure our logging practices

```
[2024-11-12T10:15:25.345Z] [INFO] Connected to the database at db-
server.example.com:5432
[2024-11-12T10:15:26.456Z] [INFO] Executing query: SELECT * FROM users WHERE email =
'**********' AND password = '**********'
[2024-11-12T10:15:27.567Z] [ERROR] Database error: Connection timed out while accessing
db-server.example.com with username db_user and password '**********'
[2024-11-12T10:15:28.678Z] [DEBUG] API request payload: {"creditCardNumber":
"************", "expiration": "**/**", "cvv": "***"}
```

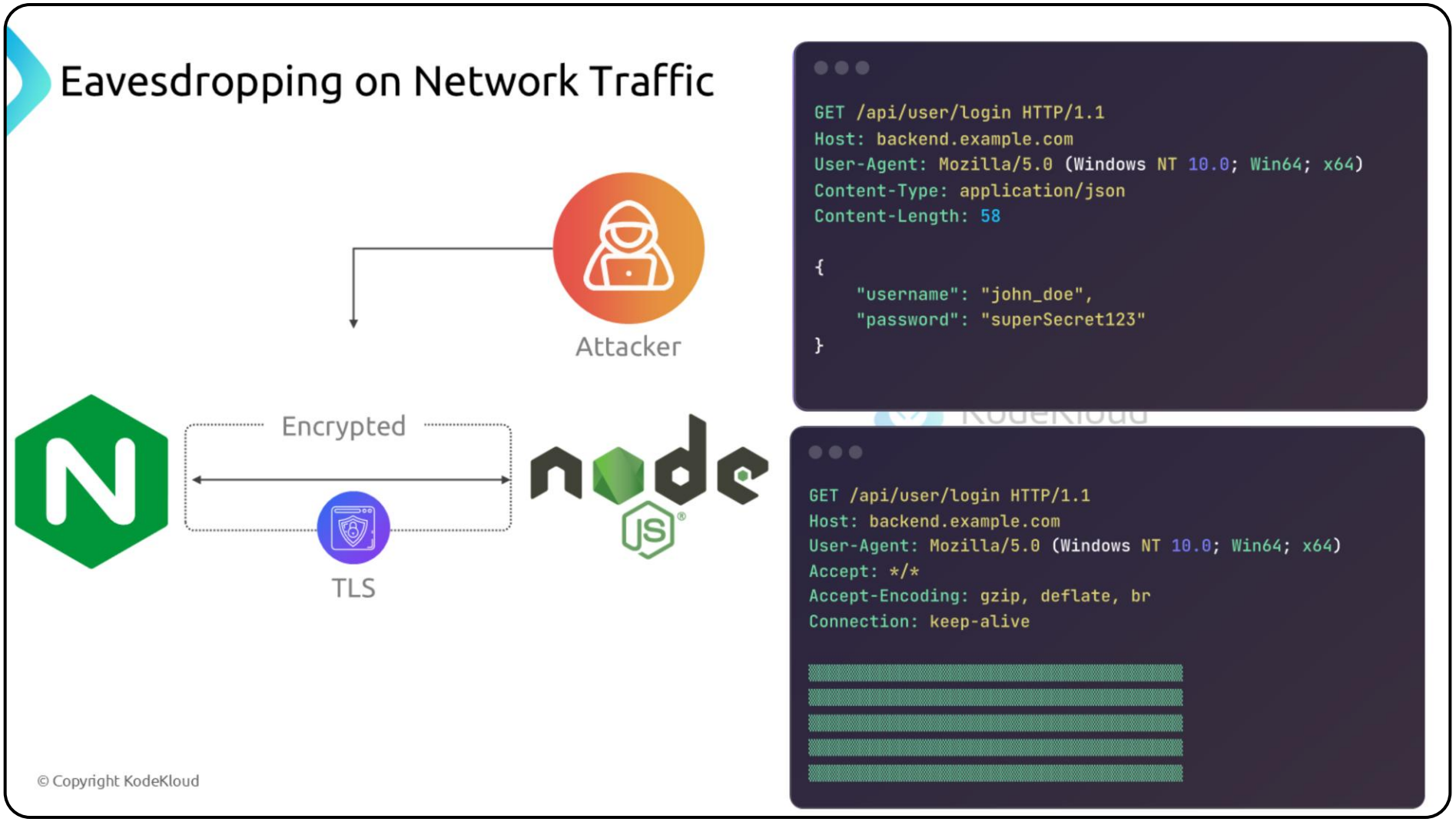
Restrict access to logs

Next, Viewing Sensitive Data in Logs

Our application generates logs that might inadvertently contain sensitive information.

For instance, if our logs include database queries or API request details, an attacker who gains access to these logs could extract valuable data. This could provide the attacker with critical insights into the system's operations and potential vulnerabilities.

We must secure our logs to ensure they do not contain sensitive information. This involves configuring our logging practices to avoid logging sensitive data like database credentials or API keys. We should also restrict access to logs so that only authorized personnel can view them. In our application, centralized logging solutions that provide fine-grained access controls and monitor log access for suspicious activities are essential.



Finally, Eavesdropping on Network Traffic.

If the communication between the frontend Nginx server and the backend Node.js services is not encrypted, an attacker with network access could intercept this traffic. By capturing unencrypted data transfers, the attacker could gain access to sensitive information such as user credentials or internal API calls. This could lead to unauthorized access and data breaches.

encrypting network traffic is crucial to prevent eavesdropping. In our application, we use TLS (Transport Layer Security) to

encrypt communication between the frontend Nginx server and the backend Node.js services, as well as between all other components. This ensures that even if an attacker intercepts the traffic, they cannot read the sensitive information being transmitted.

Summary

01 Ensure RBAC permissions are correctly configured to avoid excessive access

02 Secure logs to prevent storing and exposing sensitive information

03 Encrypt network traffic using TLS to prevent eavesdropping attacks

Visit www.kodekloud.com to learn more.