# 100 Methods For Container Attacks

HADESS

WWW.HADESS.IO

RedTeamRecipe

Red Team Recipe for Fun & Profit.

Share

Follow

# 100 Methods for Container Attacks(RTC0010)

## Insecure Container Images

1. Using Trivy:

```
trivy -q -f json <container_name>:<tag> | jq '.[] | select(.Vulnerabilities != null)'
```

This command uses Trivy, a vulnerability scanner for containers, to scan a specific container image (`<container_name>:<tag>`) for vulnerabilities. The `-q` flag suppresses the output, and the `-f json` flag formats the output as JSON. The command then uses `jq` to filter the results and display only the vulnerabilities found.

1. Using Clair-scanner:

```
clair-scanner --ip <container_ip> --report <report_output_file>
```

This command utilizes Clair-scanner, a tool that integrates with the Clair vulnerability database, to scan a running container (`<container_ip>`) and generate a report (`<report_output_file>`) containing the vulnerabilities found.

1. Using kube-hunter:

```
kube-hunter --remote <cluster_address> | grep -B 5 "Critical:"
```

This command employs kube-hunter, a Kubernetes penetration testing tool, to scan a remote Kubernetes cluster (`<cluster_address>`) for security vulnerabilities. The output is then piped to `grep` to filter and display only the critical vulnerabilities found.

## Malicious Images via Aqua

- docker-network-bridge-
- ipv6:0.0.2
- docker-network-bridge-
- ipv6:0.0.1
- docker-network-ipv6:0.0.12
- ubuntu:latest
- ubuntu:latest
- ubuntu:18.04
- busybox:latest
- alpine: latest
- alpine-curl
- xmrig:latest
- alpine: 3.13
- dockgeddon: latest
- tornadorangepwn:latest
- jaganod: latest
- redis: latest
- gin: latest (built on host)
- dockgeddon:latest
- fcminer: latest
- debian:latest
- borg:latest
- docked:latestk8s.gcr.io/pause:0.8
- dockgeddon:latest
- stage2: latest
- dockerlan:latest
- wayren:latest
- basicxmr:latest
- simpledockerxmr:latest
- wscopescan:latest
- small: latest
- app:latest
- Monero-miner: latest
- utnubu:latest
- vbuntu:latest
- swarm-agents:latest
- scope: 1.13.2
- apache:latest
- kimura: 1.0
- xmrig: latest
- sandeep078: latest
- tntbbo:latest
- kuben2

## Other Images

- OfficialImagee

- Ubuntuu
- CentoS
- Alp1ne
- Pythoon

## Privileged Container

1. Using kube-score:

```
kube-score score <cluster_address> --filter-allowed-privilege-escalation=false | grep "Privileged:"
```

This command utilizes kube-score, a Kubernetes security configuration scanner, to score a Kubernetes cluster (`<cluster_address>`) and filter out containers that allow privilege escalation. The output is then filtered using `grep` to display only containers flagged as “Privileged.”

1. Using kubectl and jq:

```
kubectl get pods --all-namespaces -o json | jq '.items[] | select(.spec.containers[].securityContext.privileged==true) | .metadata.name'
```

This command uses kubectl to fetch information about all pods in all namespaces of a Kubernetes cluster. The output is formatted as JSON, which is then processed by `jq`. The command filters out pods that have containers with privileged security context and displays the names of those pods.

1. Using kube-hunter:

```
kube-hunter --remote <cluster_address> | grep -B 5 "Privileged: true"
```

## Container Escape

1. Using Linpeas:

```
docker run --rm -v /:/host -t linpeas -C | grep -E "(Writable to|Capabilities|Capabilities).*"
```

This command uses Linpeas, a Linux privilege escalation checking script, to perform a scan inside a container. The container is run with access to the host’s root filesystem (`-v /:/host`), allowing Linpeas to check for vulnerabilities. The output is then filtered using `grep` to display relevant information related to writable files and capabilities.

1. Using kubeletctl:

```
docker run --rm -it quay.io/kubepwn/kubeletctl containerescape -v
```

This command utilizes kubeletctl, a tool for exploiting Kubernetes kubelet, to check for container escape vulnerabilities. The container is run with the `containerescape` command, and the `-v` flag is used to enable verbose output, providing detailed information about any vulnerabilities found.

1. Using GTFOBins and grep:

```
docker run --rm -v /:/mnt alpine sh -c "wget -qO-
https://raw.githubusercontent.com/GTFOBins/GTFOBins.github.io/master/gtfobins/<
| grep -Eo '(sudo|chmod|chown) \.[^\s]+'"
```

This command uses GTFOBins, a curated list of Unix binaries that can be used for privilege escalation, to search for potential container escape vectors. The container runs an Alpine Linux image, and the specified `<binary>` is fetched from the GTFOBins repository. The output is filtered using `grep` to display any commands that could potentially be used for privilege escalation.

## Container Image Tampering

1. Using Trivy:

```
trivy -q -f json --exit-code 1 <container_name>:<tag> || echo "Container
image tampered"
```

This command uses Trivy to scan a specific container image (`<container_name>:<tag>`) for vulnerabilities. The `-q` flag suppresses the output, and the `-f json` flag formats the output as JSON. The `--exit-code 1` flag causes Trivy to exit with a non-zero status code if vulnerabilities are found. If the exit code is non-zero, it means that vulnerabilities were detected, indicating a possible container image tampering.

1. Using Docker Content Trust:

```
DOCKER_CONTENT_TRUST=1 docker pull <container_name>:<tag> || echo "Container
image tampered"
```

This command pulls a container image (`<container_name>:<tag>`) with Docker Content Trust (DCT) enabled. DCT ensures the authenticity and integrity of the image by verifying the image signature. If the image has been tampered with and the signature verification fails, the command will output “Container image tampered.”

1. Using Anchore Engine:

```
anchore-cli image vuln <container_name>:<tag> os | grep -i "malware"
```

This command uses Anchore Engine, an open-source container security tool, to scan a container image (`<container_name>:<tag>`) for vulnerabilities. The `image vuln` command checks for vulnerabilities specifically in the operating system layer of the image. The output is then filtered using `grep` to search for any mentions of "malware," indicating potential image tampering.

## Insecure Container Configuration

1. Using kube-score:

```
kube-score score <cluster_address> --filter-allow-privileged=true --filter-security-context=false --filter-capabilities=false --filter-read-only-root-filesystem=false
```

This command utilizes kube-score, a Kubernetes security configuration scanner, to score a Kubernetes cluster (`<cluster_address>`) and filter out containers with insecure configurations. The command checks for containers that are allowed to run as privileged, containers without security context, containers with escalated capabilities, and containers without a read-only root filesystem.

1. Using kube-hunter:

```
kube-hunter --remote <cluster_address> | grep -E "(Unauthenticated Access|Insecure Configuration)"
```

This command employs kube-hunter to scan a remote Kubernetes cluster (`<cluster_address>`) for security vulnerabilities. The output is then filtered using `grep` to display findings related to unauthenticated access and insecure configurations.

1. Using Kritis and kubectl:

```
kubectl get kritisconstraints -A -o json | jq -r '.items[] | select(.spec.requirements[].disallowConfigMapOrSecretAccess==true) | .metadata.name'
```

This command uses Kritis, a Kubernetes admission controller, to fetch the Kritis constraints from all namespaces in a Kubernetes cluster. The output is formatted as JSON and then processed using `jq`. The command filters out constraints that disallow access to ConfigMaps or Secrets and displays the names of the constraints.

## Denial-of-Service (DoS)

1. Using Kubei:

```
kubei scan pod --all-namespaces | grep "Denial of Service"
```

This command uses Kubei, a Kubernetes runtime scanner, to scan all pods in all namespaces for potential Denial-of-Service vulnerabilities. The output is then filtered using `grep` to display any findings related to Denial-of-Service.

1. Using Slowloris:

```
slowloris -dns <target_url> -port <target_port>
```

This command utilizes Slowloris, a popular Denial-of-Service attack tool, to launch a Slowloris attack against a target URL (`<target_url>`) and port (`<target_port>`). Slowloris attempts to exhaust the target's resources, leading to a Denial-of-Service condition.

1. Using GoBuster and curl:

```
gobuster dir -u <target_url> -w <wordlist_file> -c 200 -q | curl -X POST -d @- <target_url>
```

This command combines GoBuster, a directory and file brute-forcing tool, with curl, a command-line HTTP client, to simulate a DoS attack. GoBuster is used to discover directories and files on a target URL (`<target_url>`) using a wordlist file (`<wordlist_file>`). The output of GoBuster is then piped to curl, which sends POST requests to the target URL, potentially overwhelming the server and causing a DoS condition.

1. Using Stress-ng:

```
stress-ng --cpu <num_cpus> --io <num_io_operations> --vm <num_vm_operations> --vm-bytes <vm_memory_allocation>
```

This command utilizes Stress-ng, a tool for generating synthetic workloads and stressing various system components, to exert stress on the CPU, I/O operations, and virtual memory within the container. By adjusting the parameters, such as the number of CPUs (`<num_cpus>`) and the amount of memory allocated for virtual memory operations (`<vm_memory_allocation>`), you can evaluate the container's ability to handle high loads and identify any potential DoS vulnerabilities.

1. Using tc (traffic control):

```
tc qdisc add dev eth0 root netem loss <packet_loss_percentage>%
```

This command uses the tc command, which is part of the Linux Traffic Control suite, to introduce packet loss on the network interface (`eth0`) within the container. By specifying a packet loss percentage (`<packet_loss_percentage>`), you can simulate network congestion and evaluate the container's resilience to network-related DoS attacks.

## Kernel Vulnerabilities

1. Using ksplice-uptrack:

```
uptrack-show --available | grep kernel
```

This command uses ksplice-uptrack, a tool for live patching the Linux kernel, to check for available kernel updates and specifically filter for kernel-related vulnerabilities. The command lists the available kernel updates, and `grep` is used to display only the kernel-related entries.

1. Using Linux Exploit Suggester:

```
wget https://raw.githubusercontent.com/mzet-/linux-exploit-suggester/master/linux-exploit-suggester.sh -O les.sh && chmod +x les.sh && ./les.sh
```

This command downloads the Linux Exploit Suggester script (`les.sh`) from its GitHub repository and executes it. The script performs a scan on the system's kernel and provides a list of potential kernel exploits and vulnerabilities based on the kernel version and configuration.

1. Using KernelCare:

```
kcarectl --check
```

This command uses KernelCare, a live patching solution for the Linux kernel, to check the kernel's vulnerability status. The command verifies if the kernel has the latest patches applied and reports any missing patches or potential vulnerabilities.

## Shared Kernel Exploitation

1. Using Docker Security Scanning (DSS):

```
docker scan <container_name>:<tag> | grep -i "shared kernel"
```

This command uses Docker Security Scanning (DSS) to scan a specific container image (`<container_name>:<tag>`) for known vulnerabilities. The output is then filtered using `grep` to search for any findings related to shared kernel exploitation vulnerabilities.

1. Using Kubernetes Security Context:

```
kubectl get pods --all-namespaces -o=jsonpath='{range .items[*]}{.metadata.name}{"\t"}{.spec.securityContext.hostPID}{"\n"}{end}' | grep -iE "true"
```

This command retrieves information about all pods in all namespaces within a Kubernetes cluster. It uses `jsonpath` to extract the pod name and the `hostPID` value from the pod's security context. The output is then filtered using `grep` to identify any pods where the `hostPID` value is set to `true`. This indicates that the pod has access to the host's process namespace, which could potentially lead to shared kernel exploitation vulnerabilities.

1. Using kube-hunter:

```
kube-hunter --remote <cluster_address> | grep -B 5 "HostPID: true"
```

This command utilizes kube-hunter to scan a remote Kubernetes cluster (`<cluster_address>`) for security vulnerabilities. The output is then piped to `grep` to filter and display only the containers where the `HostPID` value is set to `true`. This indicates that the container has access to the host's process namespace, which may introduce shared kernel exploitation risks.

## Insecure Container Orchestration

1. Using kube-score:

```
kube-score score <cluster_address> --filter-kubernetes-version=false | grep -i "security"
```

This command utilizes kube-score, a Kubernetes security configuration scanner, to score a Kubernetes cluster (`<cluster_address>`) and filter out any security-related issues. The `--filter-kubernetes-version=false` flag disables the check for outdated Kubernetes versions. The output is then filtered using `grep` to display findings related to insecure container orchestration configurations.

1. Using kube-hunter:

```
kube-hunter --remote <cluster_address> | grep -B 5 "Insecure Orchestrator"
```

This command employs kube-hunter to scan a remote Kubernetes cluster (`<cluster_address>`) for security vulnerabilities. The output is then piped to `grep` to filter and display any findings related to insecure container orchestrators.

1. Using Anchore Engine:

```
anchore-cli policy status --detail | grep -iE "notallowed|deny"
```

This command uses Anchore Engine, an open-source container security tool, to check the policy status and details of containers. The output is then filtered using `grep` to display any findings related to policies that do not allow or deny specific container orchestration configurations.

## Dump All Secrets

```
find /path/to/container -type f -exec grep -EHino "secret|key|password" {} \;
```

This powerful command combines the `find` and `grep` tools. It searches for files (`-type f`) within the specified container directory and its subdirectories. For each file found, it uses `grep` to look for patterns like “secret,” “key,” or “password,” displaying the matching lines along with their line number, file name, and the actual content.

## Steal Pod Service Account Token

```
kubectl get serviceaccounts --all-namespaces -o custom-
columns=NAMESPACE:.metadata.namespace,NAME:.metadata.name,SECRET:.secrets[0].na
```

This command uses `kubectl` to retrieve a list of service accounts across all namespaces. By specifying the custom columns, it displays the namespace, service account name, and the associated secret name (where the token might be stored) if available.

## Create Admin ClusterRole

1. Using Kubeletctl and kubectl:

```
kubeletctl clusterroles --all-namespaces | grep admin
kubectl create clusterrolebinding admin-binding --clusterrole=admin --user=
<username>
```

Explanation: The first command uses Kubeletctl to list all ClusterRoles in all namespaces and filters the results to find any ClusterRoles with “admin” in their names. The second command creates a ClusterRoleBinding named “admin-binding” that assigns the “admin” ClusterRole to a specified user.

1. Using kube-score and kubectl:

```
kube-score score /path/to/kubernetes/manifest.yaml | grep -i clusterrole |
grep -i admin
kubectl create clusterrole admin --verb=<allowed-verb> --resource=<allowed-
resource>
```

Explanation: The first command uses kube-score to assess the quality and security of a Kubernetes manifest file, and then filters the results to find any occurrences of “clusterrole” and “admin”. The second command creates a new ClusterRole named “admin” with the specified allowed verbs and resources.

1. Using Kube-hunter and kubectl:

```
kube-hunter --remote <cluster-IP> | grep -i clusterrole | grep -i admin
kubectl create clusterrolebinding admin-binding --clusterrole=admin --
serviceaccount=<namespace>:<service-account>
```

Explanation: The first command uses Kube-hunter to perform a security assessment on a remote Kubernetes cluster by scanning for vulnerabilities and misconfigurations. It filters the results to identify any mentions of “clusterrole” and “admin”. The second command creates a ClusterRoleBinding named “admin-binding” that assigns the “admin” ClusterRole to a specified service account within a specific namespace.

## Create Client Certificate Credential

1. Using Gobuster and OpenSSL:

```
gobuster dir -u https://<target-url> -w /path/to/wordlist.txt -x .pem,.crt
openssl x509 -in /path/to/certificate.pem -text -noout
```

Explanation: The first command uses Gobuster to perform a directory and file enumeration on a target URL, searching for files with “.pem” or “.crt” extensions that may contain client certificate credentials. The second command uses OpenSSL to view the content of a PEM-encoded certificate file and extract relevant information such as the subject and issuer details.

1. Using Nmap and OpenSSL:

```
nmap -p 443 --script ssl-cert <target-ip>
openssl x509 -in /path/to/certificate.pem -text -noout
```

Explanation: The first command uses Nmap with the “ssl-cert” script to perform an SSL certificate enumeration on port 443 of a target IP, searching for certificates that may contain client credentials. The second command, similar to the previous example, uses OpenSSL to view the content of a PEM-encoded certificate file.

1. Using SSLScan and OpenSSL:

```
sslscan <target-ip>
openssl x509 -in /path/to/certificate.pem -text -noout
```

Explanation: The first command uses SSLScan to perform a thorough SSL/TLS vulnerability assessment on a target IP, including the enumeration of certificates. The output may contain information about client certificates. The second command, as before, uses OpenSSL to view the content of a PEM-encoded certificate file.

## Create Long-Lived Token

1. Using truffleHog and grep:

```
trufflehog --regex --entropy=False --rules /path/to/ruleset.json <repository-url> | grep "long-lived token"
```

Explanation: The command uses truffleHog, a tool for finding secrets in source code, to scan a repository specified by `<repository-url>` for long-lived tokens. It uses a ruleset specified by `/path/to/ruleset.json` and filters the results to display only lines containing “long-lived token” using grep.

1. Using GitRob and grep:

```
gitrob -commit-search -github-access-token <access-token> -organisation <org-name> | grep "long-lived token"
```

Explanation: The command uses GitRob, a tool for searching GitHub repositories, to search for long-lived tokens in the specified organization `<org-name>`. It requires a GitHub access token specified by `<access-token>`. The results are filtered to display only lines containing “long-lived token” using grep.

1. Using kube-hunter and jq:

```
kube-hunter --remote <cluster-IP> -f json | jq '. | select(.tests[].name == "Long Lived Tokens")'
```

Explanation: The command uses kube-hunter to perform a security assessment on a remote Kubernetes cluster specified by `<cluster-IP>`. It generates the output in JSON format. The output is then piped to jq, which filters the results to display only the findings related to “Long Lived Tokens”.

## Container breakout via hostPath volume mount

1. Using kube-hunter and grep:

```
kube-hunter --remote <cluster-IP> -f json | grep -A 5 "Container Breakout" | grep "hostPath"
```

Explanation: The command uses kube-hunter to perform a security assessment on a remote Kubernetes cluster specified by `<cluster-IP>`. It generates the output in JSON format. The output is then piped to grep to filter the results for “Container Breakout” findings. Then it further filters for lines that contain “hostPath” to identify potential vulnerabilities related to hostPath volume mount.

1. Using trivy and awk:

```
trivy image --no-progress <image-name> | awk '/^=== [C|c]ontainer [B|b]reakout ===/,/^---/'
```

Explanation: The command uses trivy, a vulnerability scanner for container images, to scan a container image specified by `<image-name>`. The output is piped to awk, which selects the lines between the patterns `"=== Container Breakout ==="` and `"---"` to highlight any vulnerabilities related to container breakout.

1. Using kubectl and jq:

```
kubectl get pod --all-namespaces -o json | jq '.items[].spec |
select(.volumes[].hostPath != null) | {namespace: .namespace, pod: .nodeName,
volume: .volumes[].hostPath}'
```

Explanation: The command uses kubectl to retrieve information about pods in all namespaces in JSON format. The output is piped to jq, which filters and formats the results to display the namespace, pod name, and any hostPath volumes that are being used. This can help identify pods that potentially mount hostPath volumes, which can be a security concern for container breakout vulnerabilities.

## Privilege escalation through node/proxy permissions

1. Using kube-hunter and grep:

```
kube-hunter --remote <cluster-IP> -f json | grep -A 5 "Privilege Escalation"
| grep -E "node|proxy"
```

Explanation: The command uses kube-hunter to perform a security assessment on a remote Kubernetes cluster specified by `<cluster-IP>`. It generates the output in JSON format. The output is then piped to grep to filter the results for “Privilege Escalation” findings. Then it further filters for lines that contain “node” or “proxy” to identify potential vulnerabilities related to privilege escalation through node or proxy permissions.

1. Using kubectl and jq:

```
kubectl get clusterrolebindings --all-namespaces -o json | jq '.items[] |
select(.roleRef.name == "cluster-admin") | .metadata.namespace + "/" +
.metadata.name'
```

Explanation: The command uses kubectl to retrieve information about cluster role bindings in all namespaces in JSON format. The output is piped to jq, which filters the results to select cluster role bindings with the name “cluster-admin”. It then displays the namespace and name of those bindings, which can indicate potential privilege escalation vulnerabilities.

1. Using trivy and grep:

```
trivy image --no-progress <image-name> | grep -i
"privileged:true\|hostnetwork:true\|hostpid:true\|hostipc:true"
```

Explanation: The command uses trivy, a vulnerability scanner for container images, to scan a container image specified by `<image-name>`. The output is piped to grep to search for lines containing keywords related to privilege escalation, such as “privileged:true”, “hostnetwork:true”, “hostpid:true”, and “hostipc:true”. This can help identify potential vulnerabilities related to node or proxy permissions.

## Run a Privileged Pod

1. Using kubectl and grep:

```
kubectl get pods --all-namespaces -o json | grep -i "privileged: true"
```

Explanation: The command uses kubectl to retrieve information about pods in all namespaces in JSON format. The output is piped to grep to search for lines containing “privileged: true”, which indicates that a pod is running with privileged access.

1. Using kube-hunter and grep:

```
kube-hunter --remote <cluster-IP> -f json | grep -A 5 "Privileged Containers"
```

Explanation: The command uses kube-hunter to perform a security assessment on a remote Kubernetes cluster specified by `<cluster-IP>`. It generates the output in JSON format. The output is then piped to grep to filter the results for “Privileged Containers” findings. This will identify any pods running with privileged access.

1. Using kubectl and jq:

```
kubectl get pods --all-namespaces -o json | jq '.items[] |
select(.spec.containers[].securityContext.privileged == true) |
.metadata.namespace + "/" + .metadata.name'
```

Explanation: The command uses kubectl to retrieve information about pods in all namespaces in JSON format. The output is piped to jq, which filters the results to select pods where at least one container has the “privileged” field set to true in its securityContext. It then displays the namespace and name of those pods, indicating the presence of privileged pods.

## Restrict Kubernetes API access to specific IP ranges

1. Using kubectl and jq:

```
kubectl get service -n kube-system kube-apiserver -o json | jq '.spec.ports[]
| select(.name=="https").nodePort'
```

Explanation: The command uses kubectl to retrieve information about the kube-apiserver service in the kube-system namespace. The output is piped to jq, which selects the port configuration for HTTPS and displays the corresponding nodePort. This nodePort can be used to access the Kubernetes API.

1. Using nmap and grep:

```
nmap -p <nodePort> <cluster-IP> | grep "open"
```

Explanation: The command uses nmap to scan a specific nodePort, which is obtained from the previous command. It scans the specified cluster IP for the given nodePort and filters the output to display only lines that indicate the port is “open”. This verifies if the Kubernetes API is accessible from the specified IP range.

1. Using kube-hunter and grep:

```
kube-hunter --remote <cluster-IP> -f json | grep -A 5 "Open Kubernetes API Server Proxy"
```

Explanation: The command uses kube-hunter to perform a security assessment on a remote Kubernetes cluster specified by `<cluster-IP>`. It generates the output in JSON format. The output is then piped to grep to filter the results for “Open Kubernetes API Server Proxy” findings. This identifies if the Kubernetes API server proxy is accessible without restrictions.

## Use Role-Based Access Control (RBAC)

1. Using kubectl and jq:

```
kubectl get clusterrolebindings --all-namespaces -o json | jq '.items[] | select(.roleRef.name != null) | .metadata.namespace + "/" + .metadata.name'
```

Explanation: The command uses kubectl to retrieve information about cluster role bindings in all namespaces in JSON format. The output is piped to jq, which filters the results to select cluster role bindings that have a non-null “roleRef.name”. It then displays the namespace and name of those bindings, indicating the use of RBAC.

1. Using kube-hunter and grep:

```
kube-hunter --remote <cluster-IP> -f json | grep -A 5 "RBAC"
```

Explanation: The command uses kube-hunter to perform a security assessment on a remote Kubernetes cluster specified by `<cluster-IP>`. It generates the output in JSON format. The output is then piped to grep to filter the results for “RBAC” findings. This identifies if RBAC is being used within the cluster.

1. Using kubectl and grep:

```
kubectl get rolebindings --all-namespaces -o custom-columns='NAMESPACE:.metadata.namespace, NAME:.metadata.name' | grep -v "NAMESPACE NAME"
```

Explanation: The command uses kubectl to retrieve information about role bindings in all namespaces. It formats the output to display the namespace and name of each role binding. The output is then piped to grep to filter out the header line (NAMESPACE NAME) and display only the role bindings, indicating the use of RBAC.

## Enable PodSecurityPolicy (PSP)

1. Using kubectl and jq:

```
kubectl get podsecuritypolicies.policy --all-namespaces -o json | jq '.items[].metadata.name'
```

Explanation: The command uses kubectl to retrieve information about PodSecurityPolicies in all namespaces in JSON format. The output is piped to jq, which extracts and displays the name of each PodSecurityPolicy. If any PSPs are listed, it indicates that PodSecurityPolicy is enabled.

1. Using kube-hunter and grep:

```
kube-hunter --remote <cluster-IP> -f json | grep -A 5 "Pod Security Policy"
```

Explanation: The command uses kube-hunter to perform a security assessment on a remote Kubernetes cluster specified by `<cluster-IP>`. It generates the output in JSON format. The output is then piped to grep to filter the results for “Pod Security Policy” findings. This identifies if PodSecurityPolicy is being used within the cluster.

1. Using kubectl and grep:

```
kubectl get clusterrolebindings --all-namespaces -o custom-columns='NAMESPACE:.metadata.namespace, NAME:.metadata.name, ROLE:.roleRef.name' | grep "podsecuritypolicy"
```

Explanation: The command uses kubectl to retrieve information about cluster role bindings in all namespaces. It formats the output to display the namespace, name, and roleRef name of each cluster role binding. The output is then piped to grep to filter out lines that do not mention “podsecuritypolicy”. If any matching lines are found, it indicates that PodSecurityPolicy is being utilized.

## Use Network Policies

1. Using kubectl and jq:

```
kubectl get networkpolicies --all-namespaces -o json | jq '.items[] |
select(.spec.podSelector != null) | .metadata.namespace + "/" +
.metadata.name'
```

Explanation: The command uses kubectl to retrieve information about Network Policies in all namespaces in JSON format. The output is piped to jq, which filters the results to select network policies that have a non-null “podSelector”. It then displays the namespace and name of those network policies, indicating the use of Network Policies.

1. Using kube-hunter and grep:

```
kube-hunter --remote <cluster-IP> -f json | grep -A 5 "Network Policy"
```

Explanation: The command uses kube-hunter to perform a security assessment on a remote Kubernetes cluster specified by `<cluster-IP>`. It generates the output in JSON format. The output is then piped to grep to filter the results for “Network Policy” findings. This identifies if Network Policies are being used within the cluster.

1. Using kubectl and grep:

```
kubectl get pods --all-namespaces -o json | grep -i "networkpolicies"
```

Explanation: The command uses kubectl to retrieve information about pods in all namespaces in JSON format. The output is piped to grep to search for lines containing “networkpolicies”. If any matching lines are found, it indicates that Network Policies are being utilized.

## Enable Audit Logging

1. Using kubectl and jq:

```
kubectl get auditpolicies --all-namespaces -o json | jq
'.items[].metadata.name'
```

Explanation: The command uses kubectl to retrieve information about Audit Policies in all namespaces in JSON format. The output is piped to jq, which extracts and displays the name of each Audit Policy. If any Audit Policies are listed, it indicates that audit logging is enabled.

1. Using kube-hunter and grep:

```
kube-hunter --remote <cluster-IP> -f json | grep -A 5 "Audit Logging Enabled"
```

Explanation: The command uses kube-hunter to perform a security assessment on a remote Kubernetes cluster specified by `<cluster-IP>`. It generates the output in JSON format. The output is then piped to grep to filter the results for "Audit Logging Enabled" findings. This identifies if audit logging is enabled within the cluster.

1. Using kubectl and grep:

```
kubectl get pods --all-namespaces -o json | grep -i "audit-log"
```

Explanation: The command uses kubectl to retrieve information about pods in all namespaces in JSON format. The output is piped to grep to search for lines containing "audit-log". If any matching lines are found, it indicates that audit logging is enabled, as pods may have log mounts for audit logs.

## Use Secure Service Endpoints

1. Using kubectl and jq:

```
kubectl get services --all-namespaces -o json | jq '.items[] |
select(.spec.publishNotReadyAddresses==true) | .metadata.namespace + "/" +
.metadata.name'
```

Explanation: The command uses kubectl to retrieve information about services in all namespaces in JSON format. The output is piped to jq, which filters the results to select services that have the "publishNotReadyAddresses" field set to true. It then displays the namespace and name of those services, indicating the use of secure service endpoints.

1. Using kube-hunter and grep:

```
kube-hunter --remote <cluster-IP> -f json | grep -A 5 "Secure Service
Endpoints"
```

Explanation: The command uses kube-hunter to perform a security assessment on a remote Kubernetes cluster specified by `<cluster-IP>`. It generates the output in JSON format. The output is then piped to grep to filter the results for "Secure Service Endpoints" findings. This identifies if secure service endpoints are being used within the cluster.

1. Using kubectl and grep:

```
kubectl get endpoints --all-namespaces -o json | grep -i "subset"
```

Explanation: The command uses kubectl to retrieve information about endpoints in all namespaces in JSON format. The output is piped to grep to search for lines containing “subset”. If any matching lines are found, it indicates that subsets, which are commonly used for secure service endpoints, are defined for the endpoints.

## Use Pod Security Context

1. Using kubectl and jq:

```
kubectl get pods --all-namespaces -o json | jq '.items[] |
select(.spec.securityContext != null) | .metadata.namespace + "/" +
.metadata.name'
```

Explanation: The command uses kubectl to retrieve information about pods in all namespaces in JSON format. The output is piped to jq, which filters the results to select pods that have a non-null “securityContext” defined in their spec. It then displays the namespace and name of those pods, indicating the use of Pod Security Context.

1. Using kube-hunter and grep:

```
kube-hunter --remote <cluster-IP> -f json | grep -A 5 "Pod Security Context"
```

Explanation: The command uses kube-hunter to perform a security assessment on a remote Kubernetes cluster specified by `<cluster-IP>`. It generates the output in JSON format. The output is then piped to grep to filter the results for “Pod Security Context” findings. This identifies if Pod Security Context is being used within the cluster.

1. Using kubectl and grep:

```
kubectl get pods --all-namespaces -o json | grep -i "securityContext"
```

Explanation: The command uses kubectl to retrieve information about pods in all namespaces in JSON format. The output is piped to grep to search for lines containing “securityContext”. If any matching lines are found, it indicates that Pod Security Context is being utilized.

## Use Kubernetes Secrets

1. Using kubectl and jq:

```
kubectl get secrets --all-namespaces -o json | jq '.items[] |
select(.type!="Opaque") | .metadata.namespace + "/" + .metadata.name'
```

Explanation: The command uses kubectl to retrieve information about secrets in all namespaces in JSON format. The output is piped to jq, which filters the results to select secrets that have a type other than “Opaque”. It then displays the namespace and name of those secrets, indicating the use of Kubernetes Secrets.

1. Using kube-hunter and grep:

```
kube-hunter --remote <cluster-IP> -f json | grep -A 5 "Kubernetes Secrets"
```

Explanation: The command uses kube-hunter to perform a security assessment on a remote Kubernetes cluster specified by `<cluster-IP>`. It generates the output in JSON format. The output is then piped to grep to filter the results for “Kubernetes Secrets” findings. This identifies if Kubernetes Secrets are being used within the cluster.

1. Using kubectl and grep:

```
kubectl get pods --all-namespaces -o json | grep -i "secrets"
```

Explanation: The command uses kubectl to retrieve information about pods in all namespaces in JSON format. The output is piped to grep to search for lines containing “secrets”. If any matching lines are found, it indicates that Kubernetes Secrets are being utilized.

## Enable Container Runtime Protection

1. Using kube-hunter and grep:

```
kube-hunter --remote <cluster-IP> -f json | grep -A 5 "Container Runtime Protection"
```

Explanation: The command uses kube-hunter to perform a security assessment on a remote Kubernetes cluster specified by `<cluster-IP>`. It generates the output in JSON format. The output is then piped to grep to filter the results for “Container Runtime Protection” findings. This identifies if container runtime protection is enabled within the cluster.

1. Using trivy and grep:

```
trivy image --no-progress <image-name> | grep -i "vulnerabilities" | grep -i "container runtime"
```

Explanation: The command uses trivy, a vulnerability scanner for container images, to scan a container image specified by `<image-name>`. The output is piped to grep to search for lines containing “vulnerabilities” and “container runtime”. This can indicate if container runtime protection is being used to detect and mitigate vulnerabilities in the container runtime environment.

1. Using kubectl and jq:

```
kubectl get podsecuritypolicies.policy --all-namespaces -o json | jq '.items[] | select(.spec.runtimeClass != null) | .metadata.namespace + "/" + .metadata.name'
```

Explanation: The command uses kubectl to retrieve information about PodSecurityPolicies in all namespaces in JSON format. The output is piped to jq, which filters the results to select pod security policies that have a non-null "runtimeClass" defined. It then displays the namespace and name of those policies, indicating the use of container runtime protection.

## Enable Admission Controllers

1. Using kubectl and jq:

```
kubectl get mutatingwebhookconfigurations.admissionregistration.k8s.io --all-namespaces -o json | jq '.items[].metadata.name'
```

Explanation: The command uses kubectl to retrieve information about MutatingWebhookConfigurations in all namespaces in JSON format. The output is piped to jq, which extracts and displays the name of each MutatingWebhookConfiguration. If any MutatingWebhookConfigurations are listed, it indicates that admission controllers are enabled.

1. Using kube-hunter and grep:

```
kube-hunter --remote <cluster-IP> -f json | grep -A 5 "Admission Controllers"
```

Explanation: The command uses kube-hunter to perform a security assessment on a remote Kubernetes cluster specified by `<cluster-IP>`. It generates the output in JSON format. The output is then piped to grep to filter the results for "Admission Controllers" findings. This identifies if admission controllers are being used within the cluster.

1. Using kubectl and grep:

```
kubectl get validatingwebhookconfigurations.admissionregistration.k8s.io --all-namespaces -o custom-columns='NAMESPACE:.metadata.namespace,NAME:.metadata.name' | grep -v "NAMESPACE NAME"
```

Explanation: The command uses kubectl to retrieve information about ValidatingWebhookConfigurations in all namespaces. It formats the output to display the namespace and name of each ValidatingWebhookConfiguration. The output is then piped to grep to filter out the header line (NAMESPACE NAME) and display only the validating webhook configurations, indicating the use of admission controllers.

## Enable Docker Content Trust

1. Using Docker CLI:

```
DOCKER_CONTENT_TRUST=1 docker pull <image-name>
```

Explanation: The command sets the `DOCKER_CONTENT_TRUST` environment variable to `1` to enable Docker Content Trust. It then pulls the specified container image `<image-name>`. If the pull is successful, it indicates that Docker Content Trust is enabled.

1. Using kube-hunter and grep:

```
kube-hunter --remote <cluster-IP> -f json | grep -A 5 "Docker Content Trust"
```

Explanation: The command uses kube-hunter to perform a security assessment on a remote Kubernetes cluster specified by `<cluster-IP>`. It generates the output in JSON format. The output is then piped to grep to filter the results for "Docker Content Trust" findings. This identifies if Docker Content Trust is enabled within the cluster.

1. Using Trivy and grep:

```
TRIVY_INSECURE_SKIP=1 trivy --no-progress <image-name> | grep -i "content trust"
```

Explanation: The command sets the `TRIVY_INSECURE_SKIP` environment variable to `1` to bypass security checks. It then uses Trivy, a vulnerability scanner for container images, to scan the specified container image `<image-name>`. The output is piped to grep to search for lines containing "content trust". If any matching lines are found, it indicates that Docker Content Trust is enabled.

## Restrict communication with Docker daemon to local socket

1. Using kube-hunter and grep:

```
kube-hunter --remote <cluster-IP> -f json | grep -A 5 "Docker Daemon - Local Socket"
```

Explanation: The command uses kube-hunter to perform a security assessment on a remote Kubernetes cluster specified by `<cluster-IP>`. It generates the output in JSON format. The output is then piped to grep to filter the results for "Docker Daemon - Local Socket" findings. This identifies if the communication with the Docker daemon is restricted to the local socket.

1. Using Docker CLI: