09. – 12.12.2019
Frankfurt am Main

Johannes Schnatterer, Cloudogu GmbH

Kubernetes-Security: 3 Dinge, die jeder Entwickler wissen sollte

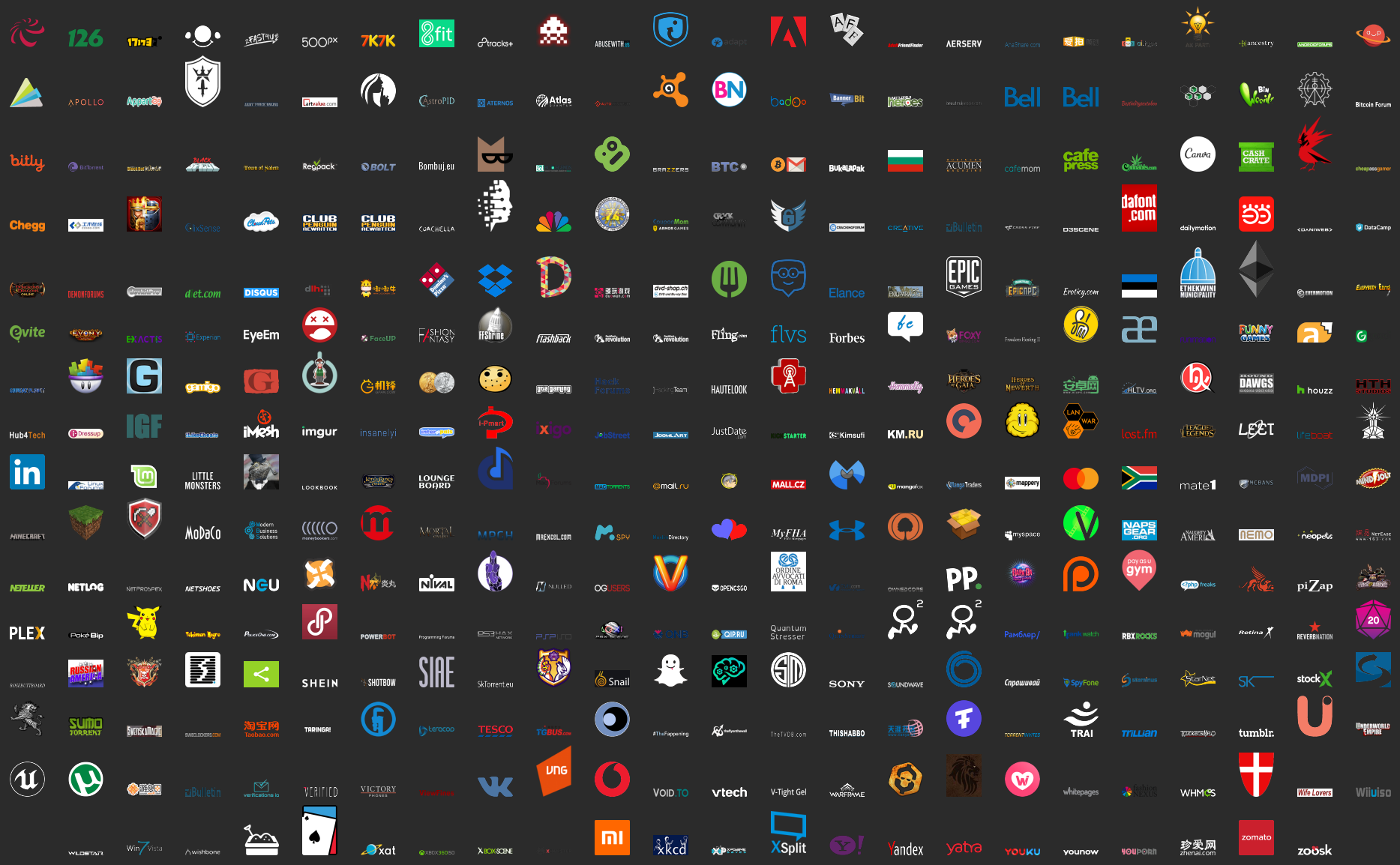
#ittage
schnatterer

https://haveibeenpwned.com/PwnedWebsites

@haveibeenpwned

https://haveibeenpwned.com/PwnedWebsites

@haveibeenpwned



https://de.wikipedia.org/wiki/Elfenbeinturm#/media/Datei:Altenmarkt_Kapelle_-_Elfenbeinerner_Turm.jpg

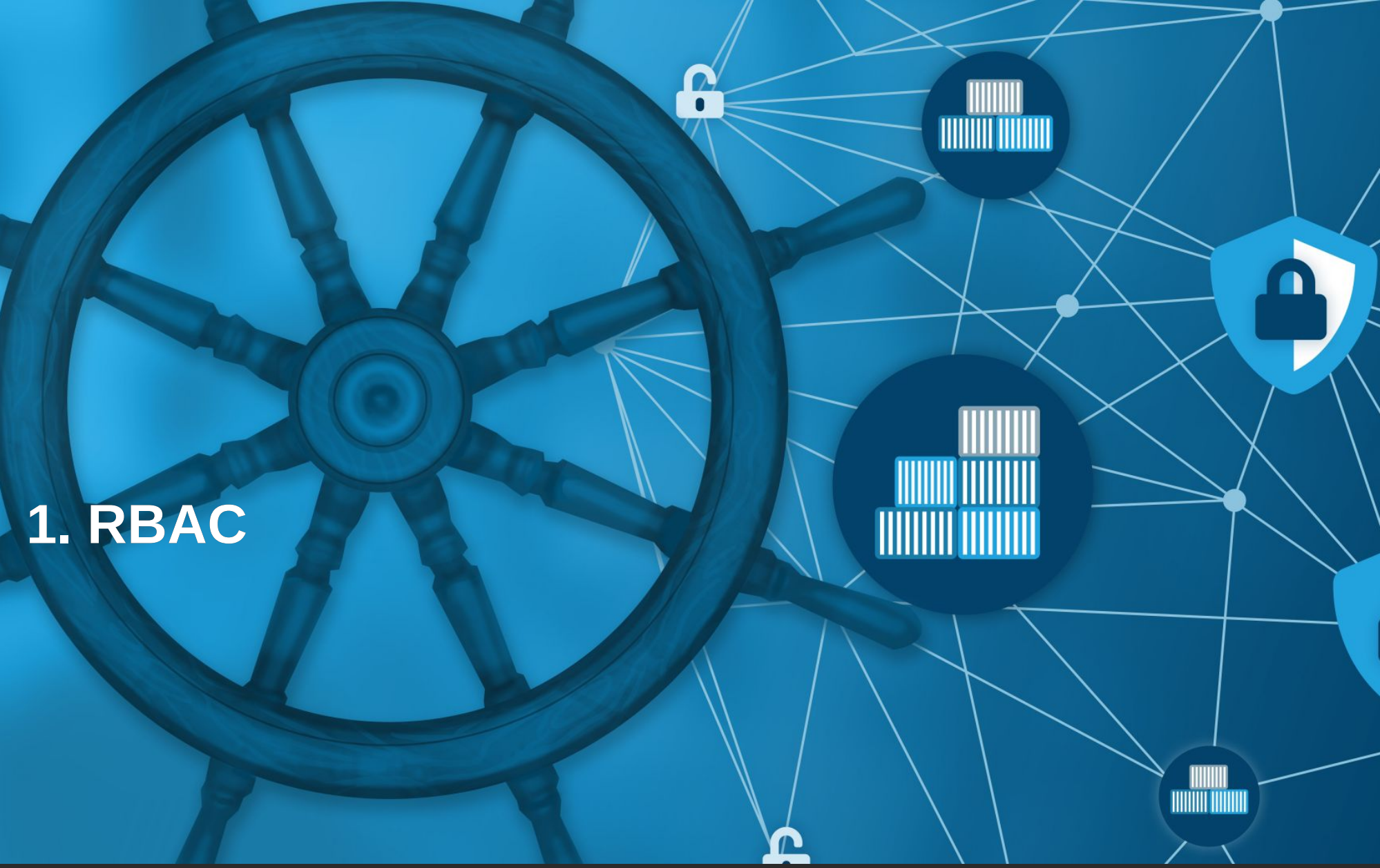
What about Security?

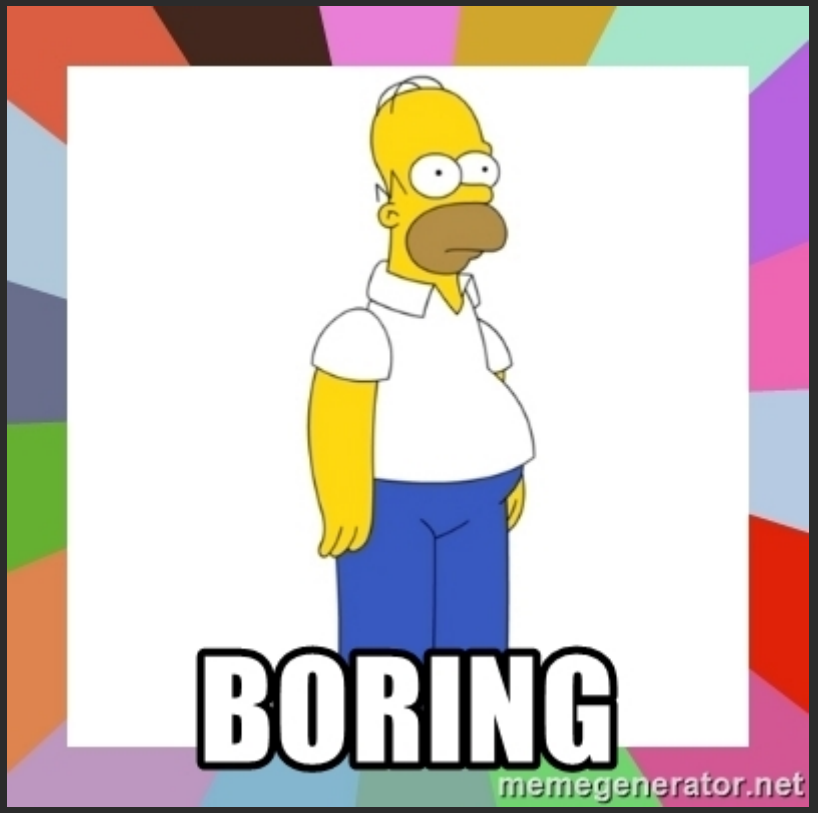
Plenty of security options


securityContext
runAsNonRoot
runAsUser
privileged
procMount
allowPrivilegeEscalation
readOnlyRootFilesystem
PodSecurityPolicy
RBAC
seccomp
Linux Capabilities
AppArmor
SELinux
NetworkPolicy
Falco
Open Policy Agent
gVisor
Kata Containers
Nabla Containers
Service Mesh
mTLS
KubeSec
KubeBench

3 things every developer should know about K8s security

- a very opinionated list of actions that make a huge difference with manageable effort
- distilled from the experience of the last years developing and operating apps on k8s


1. RBAC



https://memegenerator.net/instance/83566913/homer-simpson-boring

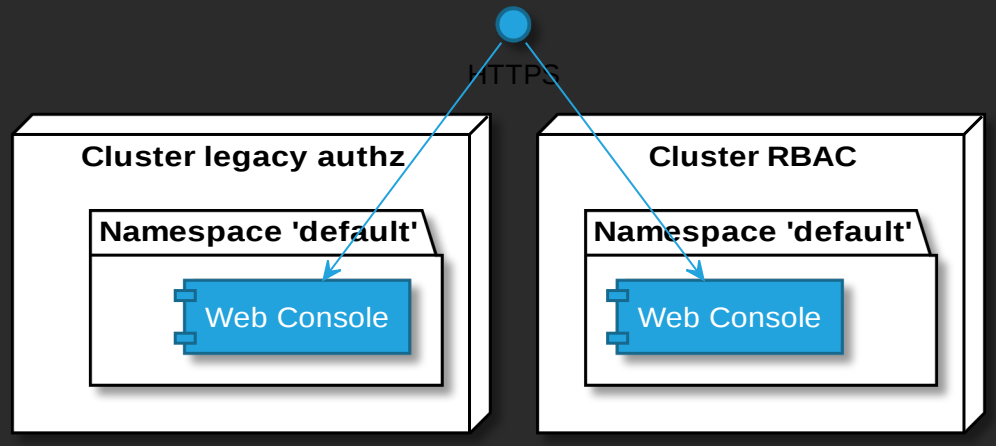
- RBAC active by default since K8s 1.6
- ... but not if you migrated!

- Every Container is mounted the token of its service account at `/var/run/secrets/kubernetes.io/serviceaccount/token`
  - With RBAC the default service account is only authorized to read publicly accessible API info
  - ⚠️ With legacy authz the default service account is cluster admin
- You can test if your pod is authorized by executing the following in it:

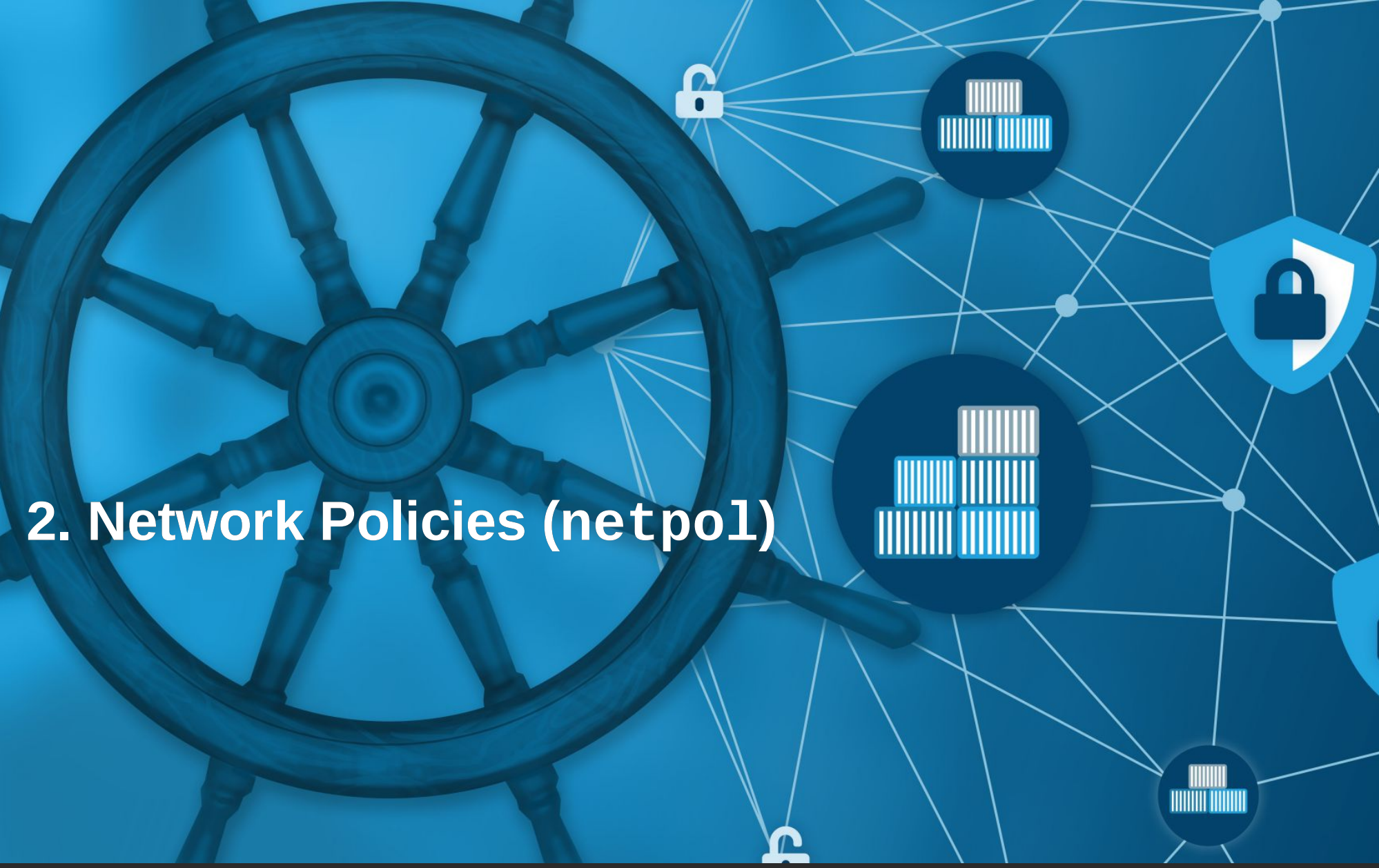
```
curl --cacert /var/run/secrets/kubernetes.io/serviceaccount/ca.crt \
  -H "Authorization: Bearer $(cat /var/run/secrets/kubernetes.io/serviceaccount/token)" \
  https://${KUBERNETES_SERVICE_HOST}/api/v1/secrets
```

- If a pod does not need access to K8s API, mounting the token can be disabled in the pod spec: `automountServiceAccountToken: false`

Demo



- legacy-authz
- RBAC

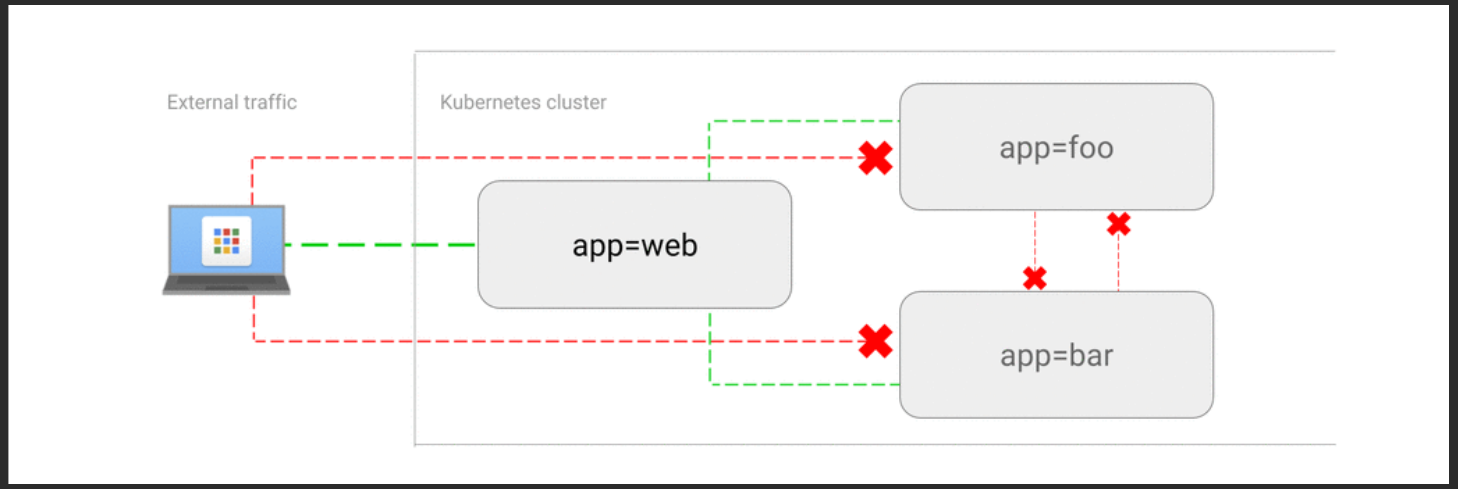


2. Network Policies (`netpol`)

A kind of firewall for communication between pods.

- Apply to pods (`podSelector`)
  - within a namespace
  - via labels
- Ingress or egress
  - to/from pods (in namespaces) or CIDRs (egress only)
  - for specific ports (optional)
- Are enforced by the CNI Plugin (e.g. Calico)
- ⚠️ No Network Policies: All traffic allowed

📌 Helpful to get started:



- https://github.com/ahmetb/kubernetes-network-policy-recipes
- Securing Cluster Networking with Network Policies - Ahmet Balkan
  https://www.youtube.com/watch?v=3gGpMmYeEO8
- Interactively describes what a netpol does:

```
kubectl describe netpol <name>
```

Recommendation: Whitelist ingress traffic

In every namespace except `kube-system`:

- Deny all ingress traffic between pods ...
- ... and then whitelist all allowed routes.

Advanced: ingress to `kube-system` namespace

⚠️ You might stop the apps in your cluster from working

For example, don't forget to:

- Allow external access to ingress controller
  (otherwise no more external access on any cluster resource)
- Allow access to kube-dns/core-dns to every namespace
  (otherwise no more service discovery by name)

Advanced: egress

- Verbose solution:
  - Deny all egress traffic between pods ...
  - ... and then whitelist all allowed routes...
  - ... repeating all ingress rules. 😕
- More pragmatic solution:
  - Allow only egress traffic within the cluster...
  - ... and then whitelist pods that need access to the internet.

Net pol pitfalls

- Don't forget to whitelist your monitoring tools (e.g. Prometheus)
- A restart of the pods might be necessary for the netpol to become effective (e.g. Prometheus)
- In order to match namespaces, labels need to be added to the namespaces, e.g.

```
kubectl label namespace/kube-system namespace=kube-system
```

- Matching both pods and namespace is only possible from k8s 1.11+
- Restricting `kube-system` might be more of a challenge (DNS, ingress controller)
- `egress` rules are more recent feature than `ingress` rules and seem less sophisticated
- Policies might not be supported by CNI Plugin.
  Make sure to test them!
  https://www.inovex.de/blog/test-kubernetes-network-policies/
- On GKE: "at least 2 nodes of type n1-standard-1" are required

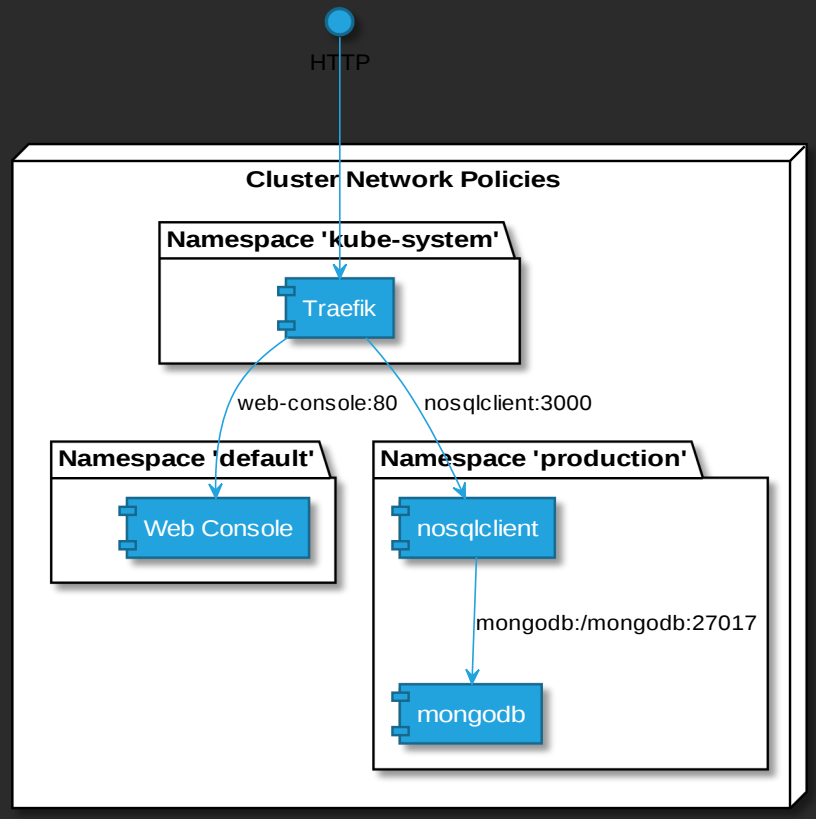
Limitations

- no option for cluster-wide policies
- whitelisting egress for domain names instead of CIDRs
- filtering on L7 (e.g. HTTP or gRPC)
- netpols will not work in multi-cloud / cluster-federation scenarios

Possible solutions:

- Proprietary extensions of CNI Plugin (e.g. cilium or calico)
- Service Meshes provides similar features and work also work with multiple clusters.
  Service Meshes operate on L7, NetPol on L3/4
  ➜ different strengths, support each other
  🌐 https://istio.io/blog/2017/0.1-using-network-policy/

Demo



- nosqlclient
- web-console

🎁 Wrap-Up: Network Policies

My recommendations:

- Definitely use DENY all ingress rule in non-`kube-system` namespaces
- Use with care
  - rules in `kube-system`
  - `egress` rules

3. Security Context

> Defines privilege and access control settings for a Pod or Container

https://kubernetes.io/docs/tasks/configure-pod-container/security-context/

See also: Secure Pods - Tim Allclair

https://www.youtube.com/watch?v=GLwmJh-j3rs

Recommendation per Container

```
apiVersion: v1
kind: Pod
# ...
metadata:
  annotations:
    seccomp.security.alpha.kubernetes.io/pod: runtime/default
spec:
  containers:
  - name: restricted
    securityContext:
      runAsNonRoot: true
      runAsUser: 100000
      runAsGroup: 100000
      readOnlyRootFilesystem: true
      allowPrivilegeEscalation: false
      capabilities:
        drop:
          - ALL
```

There is also a securityContext on pod level, but not all of those settings cannot be applied there.

Recommendation per Container in Detail (1)

- `allowPrivilegeEscalation: false`
  - mitigates a process within the container from gaining higher privileges than its parent (the container process)
  - E.g. `sudo`, `setuid`, Kernel vulnerabilities
- `seccomp.security.alpha.kubernetes.io/pod: runtime/default`
  - Enables e.g. docker's seccomp default profile that block 44/~300 Syscalls
  - Has mitigated some Kernel vulns in the past and might in the future 🔮:
    🌐 https://docs.docker.com/engine/security/non-events/
  - no seccomp profile is also one of the findings of the k8s security audit:
    🌐 https://www.cncf.io/blog/2019/08/06/open-sourcing-the-kubernetes-security-audit/
- `"capabilities": { "drop": [ "ALL" ] }`
  - Reduces attack surface
  - Drops even the default caps:
    🌐 https://github.com/moby/moby/blob/master/oci/defaults.go#L14-L30

Recommendation per Container in Detail (2)

- `runAsNonRoot: true` - Container is not started when the user is root
- `runAsUser` and `runAsGroup` > 10000
  - Reduces risk to run as user existing on host
  - In case of container escape UID/GID does not have privileges on host/filesystem
- `readOnlyRootFilesystem: true`
  - Mounts the whole file system in the container read-only. Writing only allowed in volumes.
  - Makes sure that config or code within the container cannot be manipulated.
  - It's also more efficient (no CoW).

🚧 Security context pitfalls

- `readOnlyRootFilesystem` - most applications need temp folders to write to
  - Run image locally using docker, access app (📌 run automated e2e/integration tests)
  - Then use `docker diff` to see a diff between container layer and image
  - and mount all folders listed there as `emptyDir` volumes in your pod
- `capabilities` - some images require capabilities
  - Start container locally with docker and `--cap-drop ALL`, then check logs for errors
  - Start again add caps as needed with e.g. `--cap-add CAP_CHOWN`, check logs for errors
  - Start again with additional caps and so forth.
  - Add all necessary caps to k8s resource
  - Alternative: Find an image of same app that does not require caps, e.g. `nginxinc/nginx-unprivileged`
- `runAsGroup` - beta from K8s 1.14. Before that defaults to GID 0 🙁

  🌐 https://github.com/kubernetes/enhancements/issues/213

Security context pitfalls - `runAsNonRoot`

- Non-root verification only supports numeric user. 🙄
  - `runAsUser: 100000` in `securityContext` of pod or
  - `USER 100000` in `Dockerfile` of image.
- Some official images run as root by default.
  - Find a **trusted** image that does not run as root
    e.g. for nginx, or postgres: https://hub.docker.com/r/bitnami/
  - Derive from the original image and create your own non-root image
    e.g. nginx: https://github.com/schnatterer/nginx-unpriv
- UID 100000 might not have permissions to read/write. Possible solutions:
  - Init Container sets permissions for PVCs
  - Wrong permissions in container ➜ `chmod`/`chown` in `Dockerfile`
- Some applications require a user for UID in `/etc/passwd`
  - New image that contains a user for UID e.g. `100000` or
  - Create `/etc/passwd` with user in init container and mount into application container

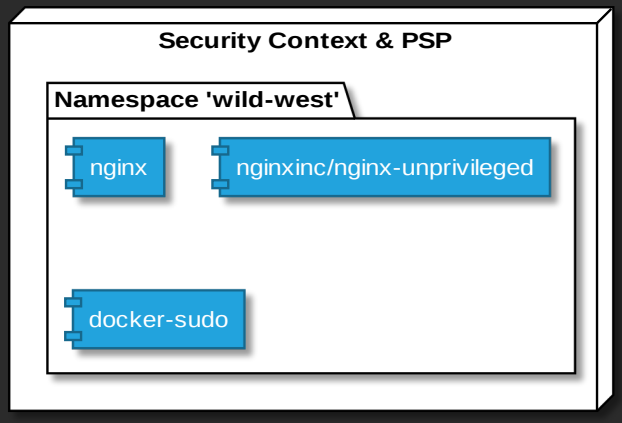
Tools

Find out if your cluster adheres to these and other good security practices:

- controlplaneio/kubesec - managable amount of checks
- Shopify/kubeaudit
  - a whole lot of checks,
  - even deny all ingress and egress NetPols and AppArmor Annotations

➜ Be prepared for a lot of findings
➜ Create your own good practices

Demo


Security Context & PSP
Namespace 'wild-west'
nginx
nginxinc/nginx-unprivileged
docker-sudo

🎁 Wrap-Up: Security Context

My recommendations

- Security Context
  - Start with least privilege
  - Only differ if there's absolutely no other way
- BTW - you can enforce Security Context Settings by using Pod Security Policies. However, those cause a lot more effort to maintain.


4 things every developer should know about K8s security?


4 things every developer should know about K8s security?
Pod Security Policies (PSP)

> a cluster-level resource [..] that define a set of conditions that a pod must run with in order to be accepted into the system

https://kubernetes.io/docs/concepts/policy/pod-security-policy/

- can be used to enforce security context cluster-wide
- has additional options such as block pods that try to
  - enter node's Linux namespaces (net, PID, etc.)
  - mounting the docker socket,
  - binding ports to nodes,
  - starting `privileged` containers
  - etc.
- more effort than security context and different syntax as in `securityContext` 😕

➜ Still highly recommended!

Too much ground to cover for 45 min!

See Demo Repo on last slide

Summary

IMHO ever person working with k8s should *at least* adhere to the following:

- Enable RBAC!
- Don't allow arbitrary connection between pods.
  (e.g. use Network Policies to whitelist ingresses)
- Start with least privilege for your containers:
  - Block privilege escalation via the security context of each container
  - Enable the seccomp default module via annotation of each pod
  - Try to run your containers
    - as non-root user, with UID & GID >= 10000,
    - with a read-only file system and
    - without capabilities.
- Least privilege rules can either be set per container (securityContext) or cluster-wide (PodSecurityPolicy)

Johannes Schnatterer

Cloudogu GmbH

https://cloudogu.com/schulungen

K8s Security series on JavaSPEKTRUM starting 05/2019

See also https://cloudogu.com/blog

@jschnatterer

@cloudogu

Demo Source: https://github.com/cloudogu/k8s-security-demos