Secure your apps and APIs inside K8s envs

CNCF webinar July 11th, 2019

Ivan Novikov
White Hat Hacker, CEO of Wallarm

K8s security

Six steps you need to make (in this direction)

1. Namespaces
2. RBAC
3. Network Policies
4. Pod Security Policies
5. Audit Policy
6. Credentials Management Policy
7. Application Security

How to start? - know your data

Remember! You want to protect data (your own, your customers', and their customers'), not environments, apps, servers, your ass…
So, let's figure out what kind of data you want to operate with before you try to isolate/secure/change it.
Example: pet toys online shop has some data like this:

- User data (emails, passwords, orders, shipping info, DoBs, etc.)
- Pet data (kind, breed, DoB, etc.)
- Carts (current state of orders)
- Items and their prices
- Banners, clicks, and other marketing things

Data is everything

So, it’s the best practice to split data by domains (personal, orders, etc.)

But the real world is different

You will be faced with corner cases all the time, like this:

- Caching servers, like memcached/redis stores parts of everything
- Temporary files, storages, and logs
- Frontend servers and apps proceeds all the data as a first tier

All these things should be solved at an application design stage before development by an architect (in an ideal world)

#1. Namespaces

To divide one cluster to a few virtual clusters.
Useful quotes from GCP team:
“K8s does not provide a mechanism to enforce security across Namespaces. You should only use it within trusted domains and not use when you need to be able to provide guarantees that a user of the cluster or pods be unable to access any of the other Namespaces resources”
“You may wish to, but you cannot create a hierarchy of namespaces. Namespaces cannot be nested within one another. ”

#1. Namespaces

No hierarchy, but we can do prefixes, like <team>-<env>
We need to control misidentifying and misusing namespaces we defined (kubectl context may help).

The suggestion here is to use the following schema (at least):
`<data-domain>-<environment>`, i.e. **`persdata-staging`**, or **`pcidss-production`**
But you can add other attributes.

#1. Namespaces.

Additional reading

- https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/
- https://kubernetes.io/docs/tasks/administer-cluster/namespaces-walkthrough/
- https://kubernetes.io/blog/2016/08/kubernetes-namespaces-use-cases-insights/

#2. RBAC

You probably noticed what we did with prefixes in namespaces? We implemented an attribute based access control, because...

“ABAC, is a powerful concept. However, as implemented in Kubernetes, ABAC is difficult to manage and understand. It requires ssh and root filesystem access on the master VM of the cluster to make authorization policy changes. For permission changes to take effect the cluster API server must be restarted.”
// GCP team

So, let’s proceed with attributes in namespaces and RBAC then

#2. RBAC. Core idea

Whitelist only (everything unlisted denied by default)

- Subject (developer, devops, process, etc)
- Resource (pod, service, etc)
- Operation (action or REST method)

ClusterRole is the same of Role, plus it allows you to give permissions for:

- Non-namespaced resources, like nodes
- Resources in all the namespaces of a cluster (please avoid this)
- Non-resource (built-in) endpoints, like /healthz

#2. RBAC to ABAC by namespace prefixes

```
contexts:
- context:
    cluster: my-cluster
    namespace: persdata-prod
    user: persdata-prod-reader
  name: persdata-prod
current-context: persdata-prod
```

#2. RBAC. What to read in addition

- https://kubernetes.io/docs/reference/access-authn-authz/rbac/
- https://kubernetes.io/blog/2017/04/rbac-support-in-kubernetes/
- https://www.cncf.io/blog/2018/08/01/demystifying-rbac-in-kubernetes/
- https://jeremievallee.com/2018/05/28/kubernetes-rbac-namespace-user.html
- https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#config_set-context/ context details

#3. Network Policies

“By default, pods are non-isolated; they accept traffic from any source.”

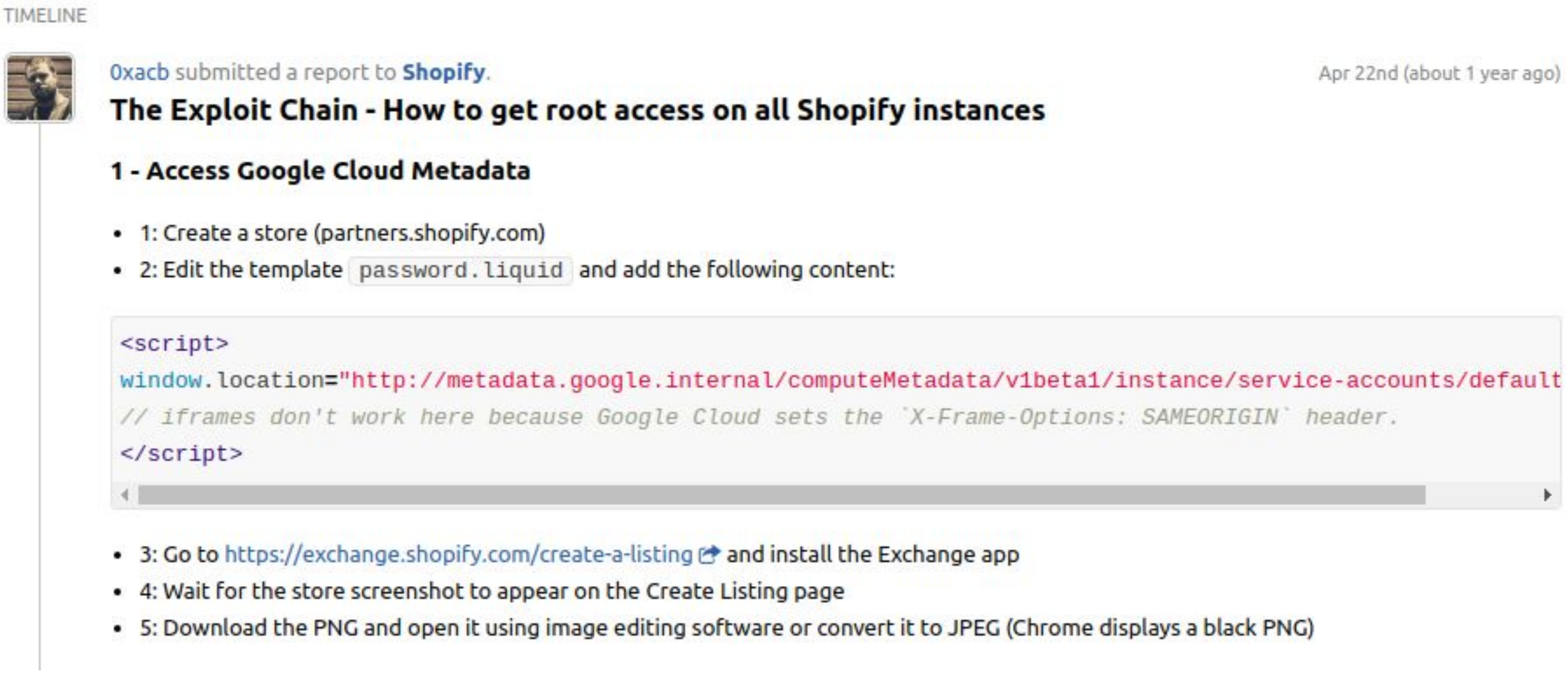
“Once there is any NetworkPolicy in a namespace selecting a particular pod, that pod will reject any connections that are not allowed by any NetworkPolicy. (Other pods in the namespace that are not selected by any NetworkPolicy will continue to accept all traffic.)”

#3. Network Policies

You also need to disable metadata API (Azure, GCP, AWS) / etcd access from all the pods to prevent local exploitation through server-side issues like SSRF (Shopify case).

Don’t forget to restrict access to your K8s services like dashboard, control panels, and others (Tesla case) wallarm

#3. Network Policies. Shopify SSRF case


TIMELINE

0xacb submitted a report to **Shopify**.

Apr 22nd (about 1 year ago)

The Exploit Chain - How to get root access on all Shopify instances

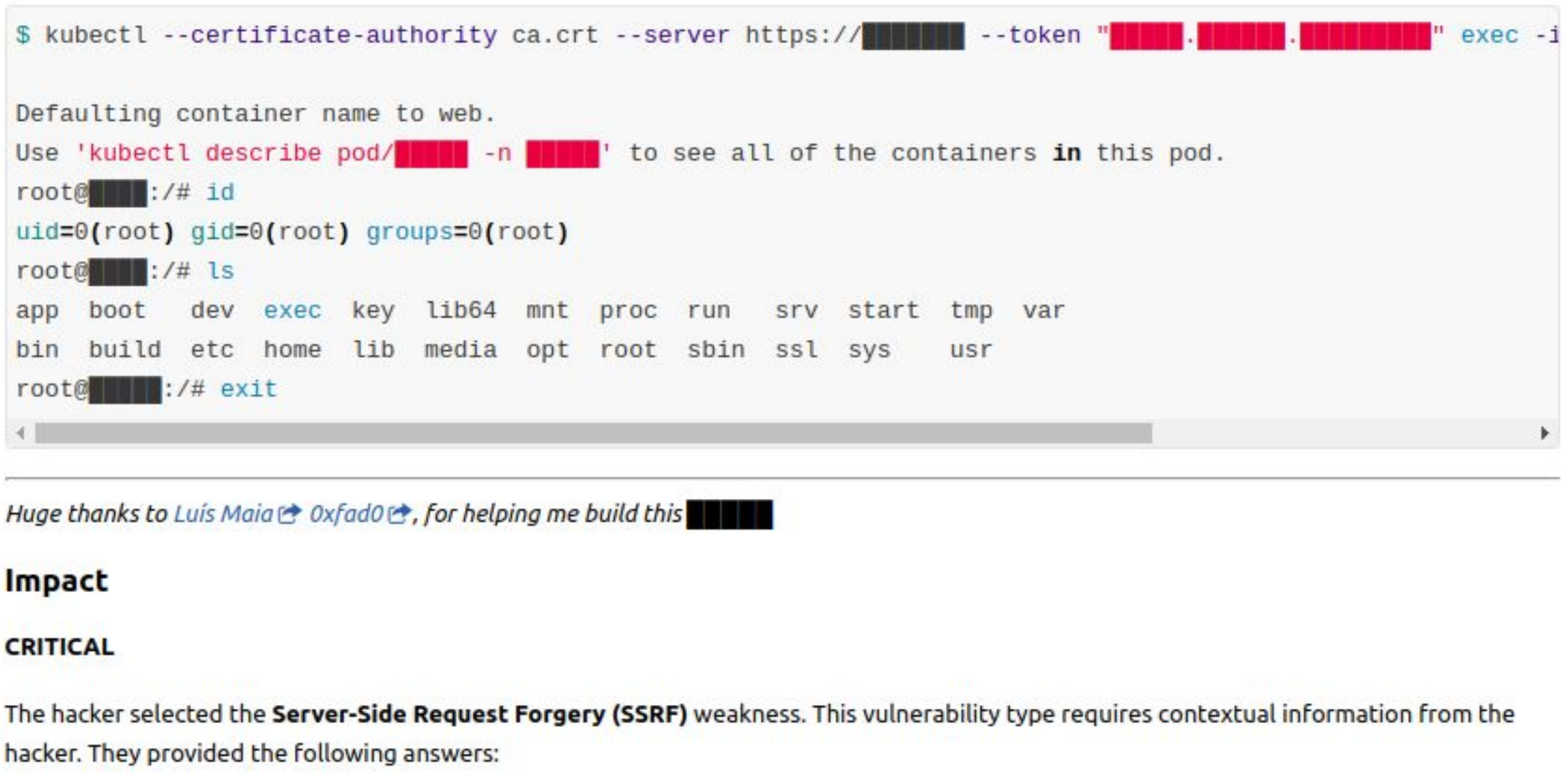
1 - Access Google Cloud Metadata

- 1: Create a store (partners.shopify.com)
- 2: Edit the template `password.liquid` and add the following content:

```
<script>
window.location="http://metadata.google.internal/computeMetadata/v1beta1/instance/service-accounts/default
// iframes don't work here because Google Cloud sets the `X-Frame-Options: SAMEORIGIN` header.
</script>
```

- 3: Go to https://exchange.shopify.com/create-a-listing and install the Exchange app
- 4: Wait for the store screenshot to appear on the Create Listing page
- 5: Download the PNG and open it using image editing software or convert it to JPEG (Chrome displays a black PNG)

#3. Network Policies. Shopify SSRF case



```
$ kubectl --certificate-authority ca.crt --server https://[redacted] --token "[redacted].[redacted].[redacted]" exec -i

Defaulting container name to web.
Use 'kubectl describe pod/[redacted] -n [redacted]' to see all of the containers in this pod.
root@[redacted]:/# id
uid=0(root) gid=0(root) groups=0(root)
root@[redacted]:/# ls
app  boot   dev  exec  key  lib64  mnt  proc  run   srv  start  tmp  var
bin  build  etc  home  lib  media  opt  root  sbin  ssl  sys    usr
root@[redacted]:/# exit
```

Huge thanks to Luís Maia 0xfad0, for helping me build this [redacted]

Impact

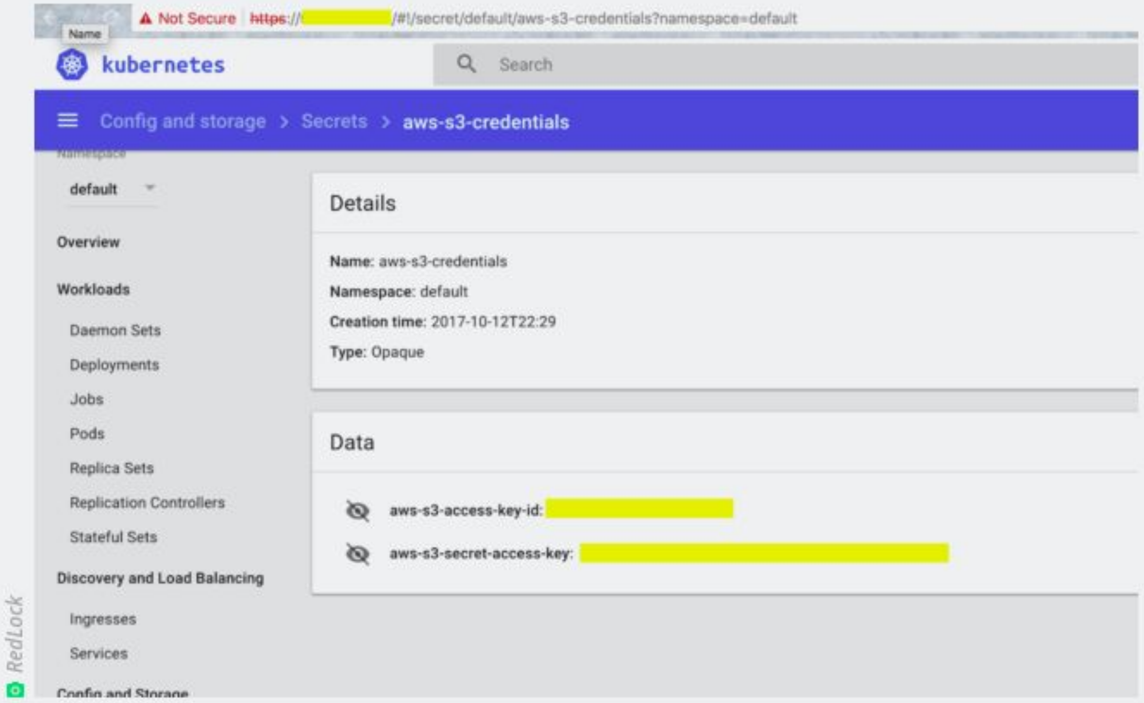
CRITICAL

The hacker selected the **Server-Side Request Forgery (SSRF)** weakness. This vulnerability type requires contextual information from the hacker. They provided the following answers:

#3. Network Policies. Tesla case

wallarm

The initial point of entry for the Tesla cloud breach, Tuesday's report said, was an unsecured administrative console for Kubernetes, an open source package used by companies to deploy and manage large numbers of cloud-based applications and resources.



Not Secure ~~https~~://[redacted]/#!/secret/default/aws-s3-credentials?namespace=default

kubernetes | Search

Config and storage > Secrets > aws-s3-credentials

Namespace: default

Overview
Workloads
Daemon Sets
Deployments
Jobs
Pods
Replica Sets
Replication Controllers
Stateful Sets
Discovery and Load Balancing
Ingresses
Services
Config and Storage

Details

Name: aws-s3-credentials
Namespace: default
Creation time: 2017-10-12T22:29
Type: Opaque

Data

aws-s3-access-key-id: [redacted]
aws-s3-secret-access-key: [redacted]

#3. Network Policies

Additional readings

- https://kubernetes.io/docs/concepts/services-networking/network-policies/
- https://kubernetes.io/docs/concepts/cluster-administration/networking/
- https://docs.bitnami.com/kubernetes/how-to/secure-your-kubernetes-application-with-networkpolicies/
- https://github.com/ahmetb/kubernetes-network-policy-recipes
- https://medium.com/@reuvenharrison/an-introduction-to-kubernetes-network-policies-for-security-people-ba92dd4c809d

#4. Pods Security Policies

“The PodSecurityPolicy objects define a set of conditions that a pod must run with in order to be accepted into the system, as well as defaults for the related fields”

“When a PodSecurityPolicy resource is created, it does nothing. In order to use it, the requesting user or target pod’s service account must be authorized to use the policy, by allowing the use verb on the policy.”

#4. Pods Security Policies

This is something like “all-in-one” endpoint security management policies:

SELinux + AppArmor + FileSystem + docker + sysctl blacklist + seccomp + addons

#4. Pods Security Policies

`privileged: false` ← to restrict container to get an access to network and devices at the host (docker thing)

`readOnlyRootFilesystem: true` ← read-only FS (remember chroot/jail?)

`MustRunAsNonRoot: true` ← chroot/jail-like thing

`allowPrivilegeEscalation: false` ← child process can't take more privileges than parent (setuid binaries like ping will be disabled)

And additional other features I mentioned above

#4. Pods Security Policies.

Additional readings

- https://kubernetes.io/docs/concepts/policy/pod-security-policy/
- https://resources.whitesourcesoftware.com/blog-whitesource/kubernetes-pod-security-policy
- https://kubernetes.io/docs/tasks/configure-pod-container/security-context/

#5. Audit Policy

wallarm

Stages

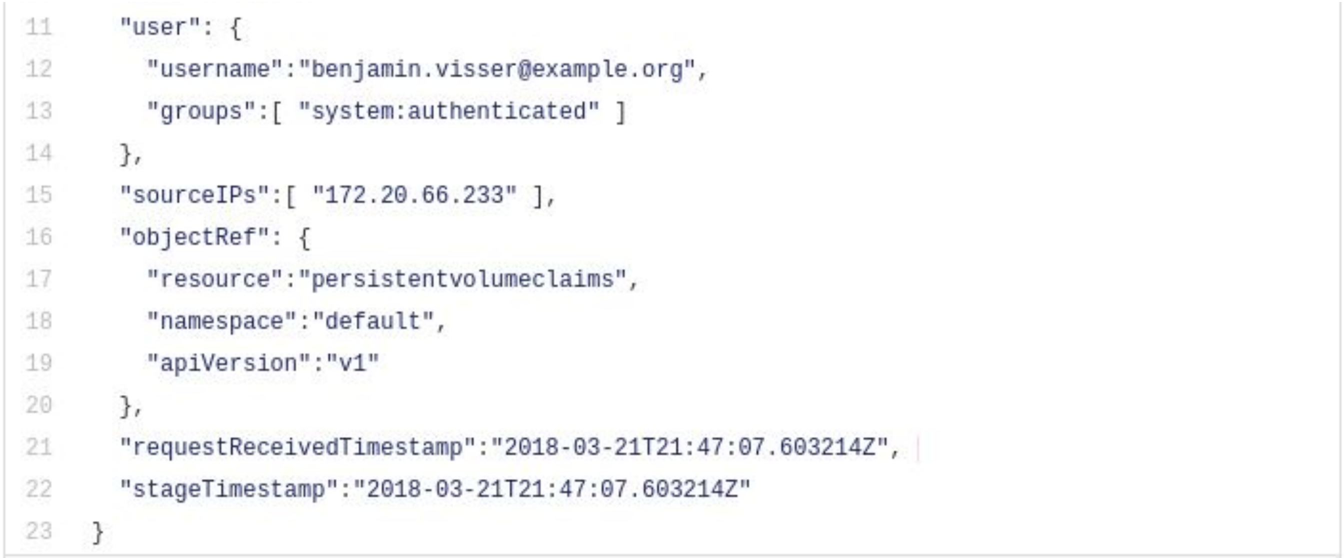
RequestReceived - The stage for events generated as soon as the audit handler receives the request.
ResponseStarted - Once the response headers are sent, but before the response body is sent. This stage is only generated for long-running requests (e.g. watch).
ResponseComplete - Once the response body has been completed.
Panic - Events generated when a panic occurred.

Levels

None - Don’t log events that match this rule.
Metadata - Log request metadata (requesting user, timestamp, resource, verb, etc.) but not request or response body.
Request - Tog event metadata and request body but not response body.
RequestResponse - Log event metadata, request and response bodies.

#5. Log your apps in the same way!

The best practice is to make your application/API logs 100% compatible to the same log format.
In this case you can easily track, correlate, and manage them together.



```
    "user": {
      "username":"benjamin.visser@example.org",
      "groups":[ "system:authenticated" ]
    },
    "sourceIPs":[ "172.20.66.233" ],
    "objectRef": {
      "resource":"persistentvolumeclaims",
      "namespace":"default",
      "apiVersion":"v1"
    },
    "requestReceivedTimestamp":"2018-03-21T21:47:07.603214Z",
    "stageTimestamp":"2018-03-21T21:47:07.603214Z"
  }
```

#5. Audit Policy
Additional readings

- https://kubernetes.io/docs/tasks/debug-application-cluster/audit/
- https://cloud.google.com/kubernetes-engine/docs/concepts/audit-policy
- https://www.noqcks.io/notes/2018/03/31/kubernetes-audit-logging-tutorial/
- https://medium.com/@noqcks/kubernetes-audit-logging-introduction-464a34a53f6c

#6. Credentials Management Policy

This thing you need to do by yourself. K8s will not help you (almost).

Certificates-only access policy as the best practice:

- PKI management tools and products can help to manage
- Expiration inside certificates already (you can’t avoid it as you can with tokens and secrets)
- OIDs for additional attribute-based features as leverage

Not only for infrastructure credentials, but for all the services to make everything in a one unified way, (I know, this is an ideal world)

Encryption native support https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/)

#7. Application Security

Server-side vulnerabilities to gain access to pods

- Here, all the steps (pp.1-6)we did before may help to mitigate the risk
- Bad news, if we did something wrong there, the rest will not help

Client-side vulnerabilities to take an access to users' data

- Audit Policies may help to identify exploitation and understand damage better

#7. Application Security

Hackers' insight

- Always use privileged UNIX ports (0-1023) for your services to avoid overmapping by race conditions
- Disable net.ipv4.tcp_fastopen inside if you don't mind
- Run everything from non-root
- Do not use host-based authentication
- Check code for vulnerabilities by SAST and DAST both
- Implement L7 firewalls (API WAFs)

#7. Application Security

Attack scenario from the wild #1

- RCE in a Python app because of the Pickle deserialization
- Python (tornado) web service was at unprivileged port 5001
- Self-kill + port re-open from other process to sniff data
- Stolen East-West secret was used to steal the data

#7. Application Security

Attack scenario from the wild #2

- RoR marshaller Remote Code Execution as an entry point
- Push-based microservices publishing at API gateway (Service Registry)
- Compromised service (let’s say shipping service) registered themself as an authentication service (login+passwords stolen)

How to keep everything we made unchanged

We need to pass through pp.1-7 and then keep it consistent.

This means that we need to avoid inconsistent namespaces, RBAC, logs, etc.

It's solvable by automated checks for YAML-based K8s configs.

Some open-source tools can help with this, like Kubeaudit (by Shopify).

What to read in additional

https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/ - just print it and read daily).

https://github.com/Shopify/kubeaudit the perfect tools for automation controls (yes, but that folks who got hacked completely because of K8s).

More About K8s Security

https://wallarm.com/solutions/waf-for-kubernetes

https://github.com/wallarm/ingress

DM us on Twitter

Ivan Novikov	Wallarm
@d0znpp	@wallarm

Ask Questions on Slack
#k8s-security-webr-711 @cloud-native.slack.com