

# Help! My Cluster Is On The Internet!

## Container Security Fundamentals

Tips & tricks on basic Kubernetes and container security from [Samuel Davidson](#).

This list is a simple, easy-to-digest summary of the advice from the [Kubecon EU 2020 talk Help! My Cluster Is On The Internet: Container Security Fundamentals](#).

## Legal-ish Preface

README:

**The ideas, tips, and tricks suggested within this doc are entirely that, suggestions. Do not take this doc as a 100% complete list of basic security needs for your containers and cluster (it certainly is not). There is no way to guarantee security and following this doc certainly will not “secure your cluster”. It merely provides some suggestions to hopefully decrease your risk. This document is intended for you to easily reference when auditing your own security practices to see if there is some room for improvement.**

**Furthermore, the ideas expressed within this doc are entirely my own. This document is not endorsed by Google, GKE, or any other organization. This document is just an easy to consume summary of my Kubecon talk + some extra tips. :)**

\* - means the topic wasn't in the original talk.

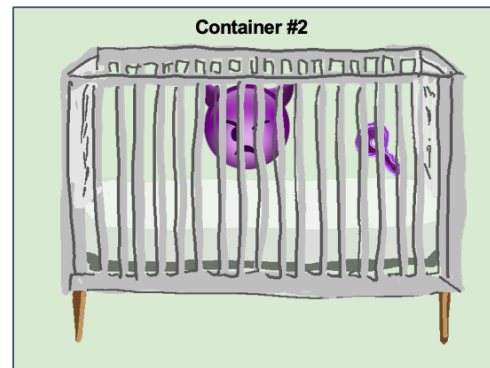
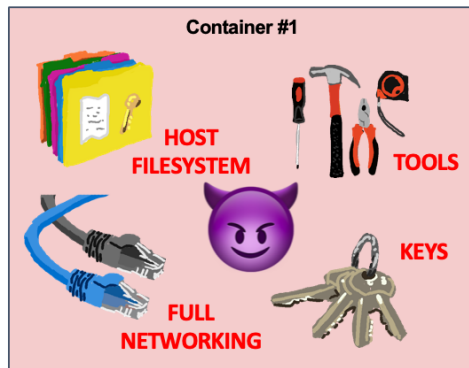
(make a copy of this doc, you can check off concepts as either *DONE* or *NA*)

Done	Idea/Tip/Trick/Suggestion
DONE	Check out this awesome useful doc! (example tip marked as <i>DONE</i> )
NA	Ignore this doc and ignore security. (example tip marked as ignored or <i>NA</i> )

## Workload

todo

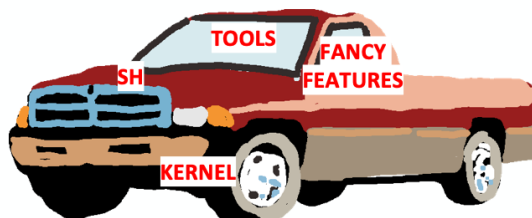
Assume you will be owned!



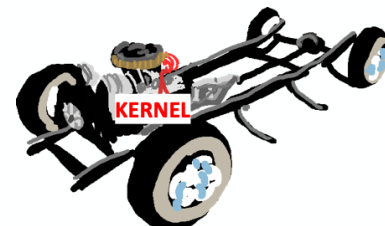
- There WILL be some yet-to-be-discovered bug or vulnerability in your code, dependencies, or base images. Always be thinking about how we can mitigate this risk.
- When making any software design decisions think about “what could an attacker do if they OWN this (whatever you’re working on)?” and how do you minimize that impact of that?

todo

Use a distroless base image.



Debian10



Distroless Debian10

- <https://github.com/GoogleContainerTools/distroless>

- “Distroless” images are bare-bones versions of common base images.
- They have the bare-minimum needed to execute a binary.
- The shell and other *developer utilities* have been removed so that if/when an attacker gains control of your container, they can’t do much of anything.
  - No shell means no shell commands. No curl, no netstat, no nmap, no vim, no scripts, ect.
  - No package manager to download additional software.
- They are super easy to transition to, example:

Before:

```
FROM golang:1.13-buster as builder
WORKDIR /go/src/app
ADD . /go/src/app

RUN go get -d -v ./...

RUN go build -o /go/bin/app

# Now copy it into our base image.
FROM debian:10
COPY --from=builder /go/bin/app /
CMD ["/app"]
```

After:

```
FROM golang:1.13-buster as builder
WORKDIR /go/src/app
ADD . /go/src/app

RUN go get -d -v ./...

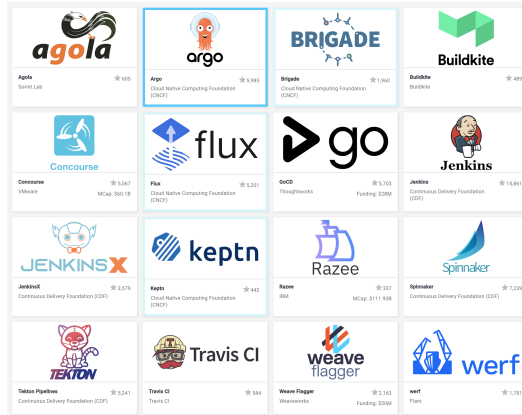
RUN go build -o /go/bin/app

# Now copy it into our base image.
FROM gcr.io/distroless/base-debian10
COPY --from=builder /go/bin/app /
CMD ["/app"]
```

- If possible, move all your containers to distroless for their prod build. But it might be useful to keep them built on non-distroless base images for testing environments.

todo

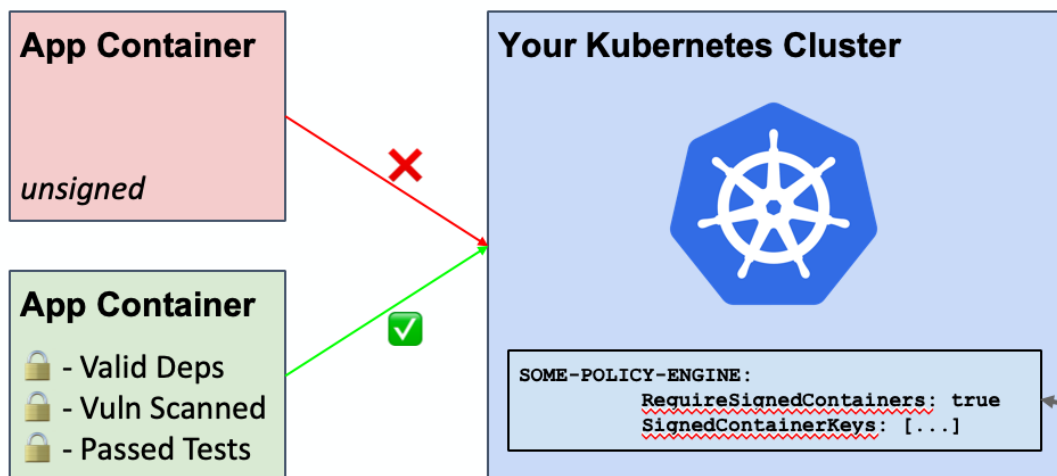
Your containers should be easy to rebuild and redeploy.



- It should be a very easy, mostly automated process to build and deploy a new version of your containers. Maybe just a click or two and some waiting after the code is submitted.
  - Typically this means integrating with a CI/CD platform ([cncf landscape options](#)).
- This is generally convenient and improves developer velocity but it actually has a big security benefit.
- Typically your biggest security vulnerabilities will be in a dependency.
  - E.g. you need to update the [express](#) version in your package.json. Then rebuild your container. Then deploy that rebuilt container to prod.
- If you have a fast easy build/deployment process then fixing vulns is often fast and easy.

todo

Sign all containers run within your cluster.

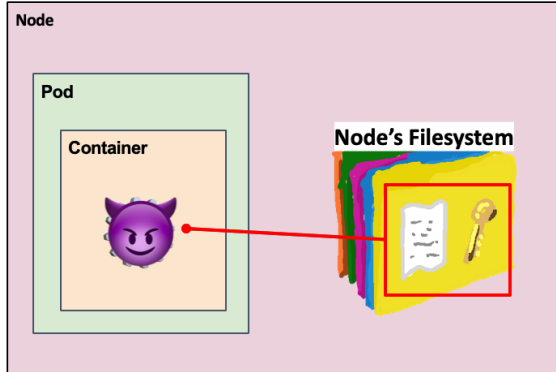


- Your cluster will be pulling images via a container URL.
  - E.g. `gcr.io/my-company/my-app:sometag`
- How do you trust that it is pulling a *good* version of your app? One that is running trusted code built from a trusted source?
  - Sure the image repository is trusted through TLS, but what if any developer or an attacker can push images to that repository?
- Your CI/CD platform should sign the containers it builds and intends for production.
  - It can attach a signature for each step in the CI/CD pipeline (e.g. vulnscan, valid dependency sources, passed E2E tests).
- Basically you generate a public/private key pair, give the private key to the CI/CD platform, the public key to the cluster, and require a valid signature before the container/pod is admitted into the cluster. A container with a bad signature or no signature will be rejected at admission-time (see image above).
- Not supported natively by Kubernetes. Requires the use of a [policy engine \(discussed below\)](#).

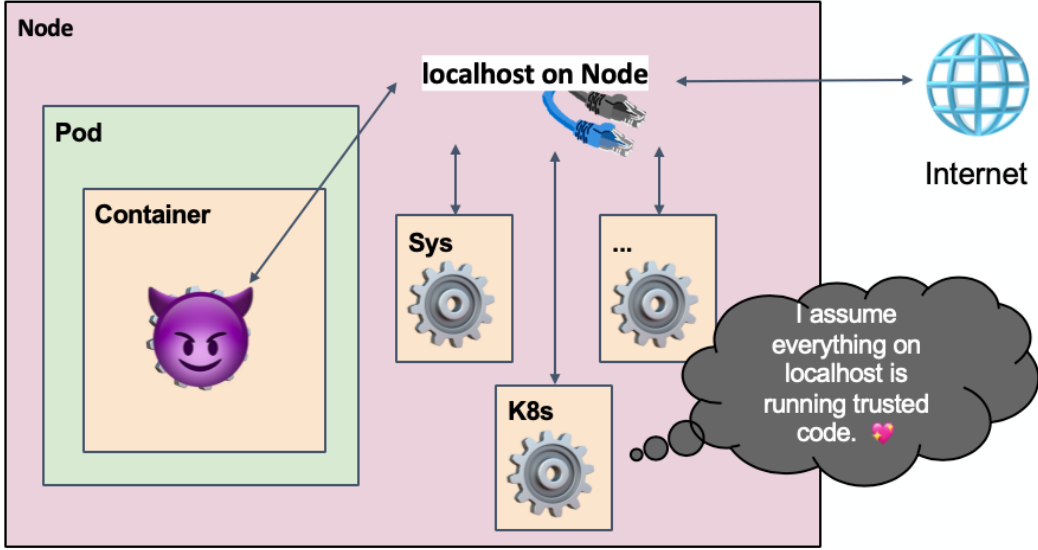
## Pod ([PodSpec](#))

todo

Don't use hostPath.



- This is a field under the `.volume` field within the container.
- Binds to the Node's filesystem. Sometimes used because it is convenient for loading in configs/keys/folders.
- Risky because when your container is owned, they have access to read/write to that directory.
  - What will that directory (and its subdirs) look like in a few months/years. Do all devs always know not to move/add to that folder?
- Also risky/bad practice because your pod shouldn't care about a specific node's filesystem. Any given node could crash/upgrade/restart and might lost that filesystem data. Also traditionally pods shouldn't know or care what Nodes they are scheduled on. Also replicating and persisting data in a

	<p>hostPath across multiple nodes is a hard and unnecessary problem.</p> <ul style="list-style-type: none"> <li>Consider using a Secret, ConfigMap, or [Persistent]VolumeClaims instead.</li> </ul>
todo	<p>Don't use hostNetwork.</p>  <ul style="list-style-type: none"> <li>Binds the pod to the Node's network. Allows localhost communication with K8s infrastructure components running on the node. The pod uses the same port range as the node.</li> <li>Allows an attacker within your container to <i>sniff around</i> the node's network and communicate with running processes. Many things like the node's kubelet treats localhost as a trust domain.</li> <li>Maybe someone enabled it some time ago for a convenience reason. Ensure it is disabled and if it isn't investigate why.</li> </ul>
todo	<p>Be conscious of your pod's service account.</p> <ul style="list-style-type: none"> <li>ALL pods have a service account even if you don't set one in the podSpec. <ul style="list-style-type: none"> <li>If not specified, the pod is assigned the SA named "default" within that pod's namespace. All created namespaces start out with a "default" SA generated within it.</li> </ul> </li> <li>The credentials for your pods SA are automatically mounted within its filesystem and available to the container (which might eventually be OWNED by an attacker).</li> <li>Chances are your container has no need for those keys to do its job. So what can you do? Here are some options: <ul style="list-style-type: none"> <li>Specify a different SA within that namespace. Use the <a href="#">.serviceAccountName</a> to provide the string name of the SA. This SA should be one that is dedicated to that pod/workload and you trust/know the authorization bindings given to it.</li> </ul> </li> </ul>

	<pre> apiVersion: v1 kind: Pod metadata:   name: simple spec:   serviceAccountName: simple-sa   containers:   - image: gcr.io/org/app     name: simple </pre> <ul style="list-style-type: none"> <li>○ Put the pod into its own Namespace. You should be doing this anyways. Namespaces are the best default security boundary within Kubernetes.</li> </ul> <pre> apiVersion: v1 kind: Pod metadata:   name: simple   namespace: simple-app-ns spec:   containers:   - image: gcr.io/org/app     name: simple </pre> <ul style="list-style-type: none"> <li>○ Disable the mounting of SA credentials within your pod with <a href="#">.automountServiceAccountToken</a>. This is probably something you should default in all your podspecs.</li> </ul> <pre> apiVersion: v1 kind: Pod metadata:   name: simple spec:   automountServiceAccountToken: false   containers:   - image: gcr.io/org/app     name: simple </pre>
todo	<p>Avoid privileged containers in your pods</p> <ul style="list-style-type: none"> <li>● If you use the <a href="#">.privileged</a> field within a podspec you are running a “privileged container”, e.g:</li> </ul>

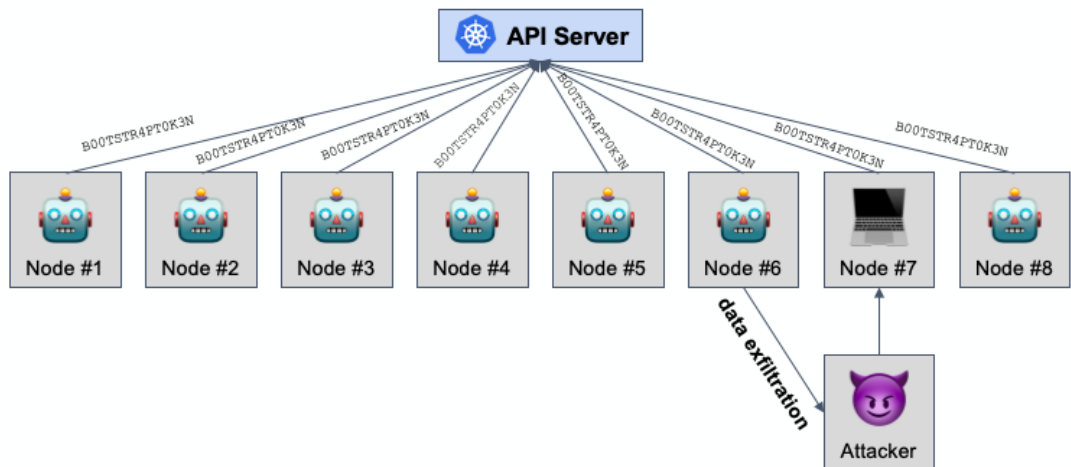
```
apiVersion: v1
kind: Pod
metadata:
  name: simple
spec:
  containers:
  - image: gcr.io/org/app
    name: simple
  securityContext:
    privileged: true
```

- This pod basically is run as root within your node, and therefore so does the attacker who OWNED your pod. This grants the pod a suite of privileges and OS capabilities.
- Similar to hostPath or hostNetwork, it might've been convenient for something, but you're opening yourself up to a ton of risk.
- <https://kubernetes.io/docs/tasks/configure-pod-container/security-context/>

## Node

todo

Bootstrap your nodes securely, trust your nodes.\*



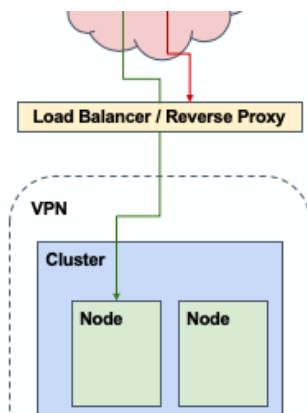
- How do you trust that your nodes are actually the machines that you think they are? Maybe this is a trivially easy problem if you just have a few nodes that you manually add to your cluster. But what about when you have 1000s of automated nodes, constantly adding, removing, re-adding themselves?
- Specially, when a computer/vm comes to your API server and says “Hey I am supposed to be a node in this cluster, give me workloads/data/secrets.”, how do you know if that machine IS supposed to be a node on the cluster?
  - Typically a bootstrap token is used, like the secret password “B00TSTR4PT0K3N” in the above picture.



- If this token is not unique or single-use what stops it from leaking and an attacker from adding their laptop as a node on your cluster?
- Even if it is single-use, it is possible an attacker could steal and use that single-use token before the real correct machine (maybe the correct machine took a long time to boot).
- This is a hard problem and it is mostly up to your K8s management plane.
  - If you're using GKE on GCP, the [Shielded Nodes](#) feature can provide a strong identity and trust in your nodes (I helped build it! :P) It leverages a [TPM signature](#) to prove the identity of the node.
  - I don't know about other managed Kubernetes solutions but feel free to suggest some so I can enumerate!

todo

Isolate your nodes (and cluster) from the internet.

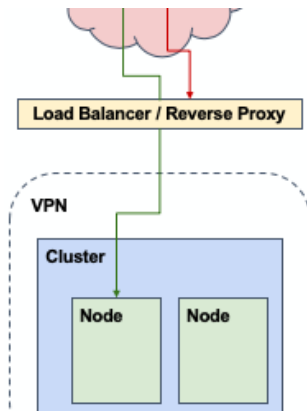


- Your nodes should not be on the internet with a publicly addressable IP.
  - Attackers can DOS your node, port scan, exfiltrate data, ect.
- Put your nodes within a VPN / behind some auth proxy.
- Use a replicated LB with a public IP to reverse proxy and forward *valid* traffic to the node machines in the internal network.
  - This prevents port scanning and can help mitigate DOS attacks.
  - Potentially you can put some authentication layer at your LB, so traffic to nodes is extra trusted with an identity tied to it! :)

Cluster

todo

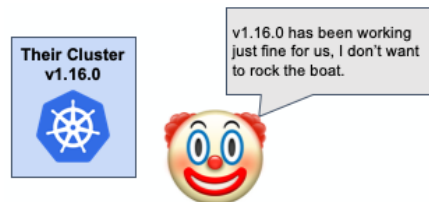
Isolate your cluster (and nodes) from the internet.



- [Same as nodes](#), no part of your control plane (and the rest of the cluster) should be on the internet with a public IP.
- Again, use a LB to forward valid (and perhaps authenticated) traffic to the API Server (control plane).

todo

Keep your cluster up to date.



- The Kubernetes community is constantly patching bugs and fixing security vulnerabilities.
  - (As of Aug 17, 2020) the latest patch version of GKE 1.16 is v1.16.14; between that version and 1.16.0 there have been [183 merged bugfix pull requests](#).
- Try to stay on the latest patch version of your minor release. And try not to run a super old, no longer maintained, minor release (~3 versions behind).
  - (As of Aug 17, 2020) the latest minor release is 1.18, so try to be running at least Kubernetes 1.15 or greater.

todo

For your secrets use the Secrets resource object.



- Secrets are:
  - Great for Access Keys, Passwords, Tokens, etc.
  - On a Node that needs them, they are stored in memory (never to

	<p>physical memory) and made readable in the filesystem through <a href="#">tempfs</a>.</p> <ul style="list-style-type: none"> <li> <ul style="list-style-type: none"> <li>■ <b>Note:</b> root users on the node can still easily access the secret within the container through the filesystem as long as that container is alive.</li> <li>○ Only loaded as-needed by pods.</li> <li>○ Easy authorization policy with RBAC.</li> <li>○ <b>Not great for non-sensitive or lengthy configs, documents, large files.</b> <ul style="list-style-type: none"> <li>■ Use ConfigMaps or other storage.</li> </ul> </li> </ul> </li> <li>• The kubernetes docs on Secrets are great, READ THEM!       <ul style="list-style-type: none"> <li>○ <a href="https://kubernetes.io/docs/concepts/configuration/secret/">https://kubernetes.io/docs/concepts/configuration/secret/</a></li> </ul> </li> <li>• Super easy to make secrets:       <div data-bbox="415 672 1419 772" style="border: 1px solid black; padding: 10px; margin: 10px 0;"> <pre>kubectl create secret generic sensitive-key --from-file=./sensitive.key --namespace=app-sensitive</pre> </div> </li> <li>• Super easy to use secrets in pods:       <div data-bbox="415 884 1419 1520" style="border: 1px solid black; padding: 10px; margin: 10px 0;"> <pre>apiVersion: v1 kind: Pod metadata:   name: pod-with-secret   namespace: app-sensitive spec:   containers:     - image: gcr.io/org/app       name: app-with-secret   volumeMounts:     - name: keys       mountPath: "/etc/key"       readOnly: true   volumes:     - name: keys       secret:         secretName: sensitive-key</pre> </div> </li> <li>• <b>The security of Secrets comes from RBAC and using many namespaces as security boundaries.</b></li> <li>• API reference:       <a href="https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.18/#secret-v1-core">https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.18/#secret-v1-core</a> </li> </ul>
todo	<p>Don't use basic auth (static password file).*</p> <ul style="list-style-type: none"> <li>• See the Kubernetes docs on basic auth:         <a href="https://kubernetes.io/docs/reference/access-authn-authz/authentication/#static-password-file">https://kubernetes.io/docs/reference/access-authn-authz/authentication/#static-password-file</a> </li> </ul>

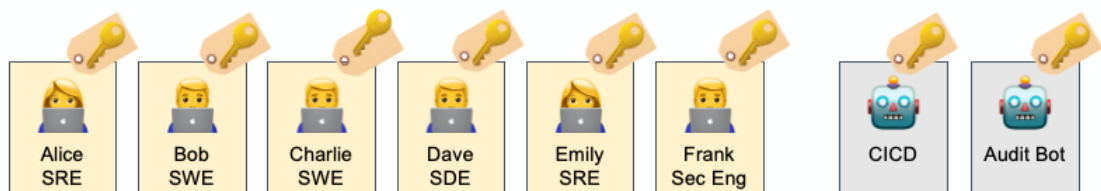
[c-password-file](#)

- Basically it is a quick convenient way to set up usernames and passwords with your API Server using this flag:  
    --basic-auth-file=...
- That file is accessible to devs with access to the API Server machine. If it leaks you are in a really bad place. Plus it could be brute-forced eventually.
- Any mature production K8s cluster **should NOT be using this flag**, ensure it is no longer in use.

## Dev/User

todo

All your devs and robots (CICD, automation) should have a unique identity.\*



- <https://kubernetes.io/docs/reference/access-authn-authz/authentication/>
- How do you trust your users and robots? Kubernetes has some built in authorization capabilities through username/passwords and service account tokens.
- It is important that every developer has an identity unique to them when interacting with your cluster's control plane. **Don't ever share identities.**
  - Example why: if a bunch of devs share a common "cluster-admin" token to authenticate with the cluster, how do you revoke privileges when one of the dev's is doing something malicious?
    - Same question for re-using tokens for your robots.
- Ensuring everybody has a unique identity is very dependent on your organization's tech. Maybe you use GSuite? Azure Active Directory? Something else?
  - Your Kubernetes cluster can integrate with these platforms via [Webhook Authentication](#) such that your devs are actually authenticated with the cluster as you@mygsuitecompany.com or whatever .
  - This probably comes for free if you are using a managed Kubernetes offering (GKE, AKS, EKS, etc...).

todo

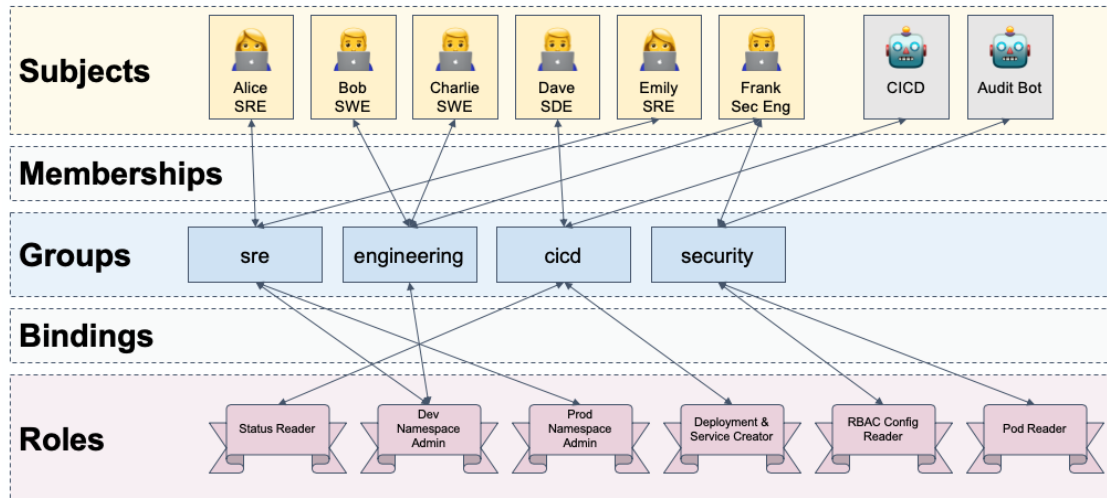
Use RBAC as your authorization engine (probably).\*

- <https://kubernetes.io/docs/reference/access-authn-authz/rbac/>
- RBAC is awesome! Super fast, built-in, fine grained authorization control in your cluster. If you're not using it to set authorization policy, why not?

- The goal is the [Principle of Least Privilege](#) (PoLP) within your cluster and RBAC's fine-grained controls make that easy.
- You might already be using another authorization layer, perhaps one that came with your cloud provider (e.g. GCP's IAM).
  - Is this authorization layer designed with Kubernetes in mind? Does it offer the fine grained control you need to enforce the PoLP?
  - For example, GCP's IAM is a great product and is integrated out of the box with a GKE cluster if you want to use it. However GCP's IAM is an older product and is NOT a kubernetes-first design. It doesn't offer fine-grained enough controls to properly enforce the PoLP, so it is recommended to use RBAC over GCP's IAM within a GKE cluster.
  - Your organization with its cloud provider / authz layer might be in a very similar situation.
  - When I say "fine-grained" enough I mean can you set allow/deny policy on API groups, namespaces, resource names, verbs, subresources, ect?

todo

Using RBAC, bind policies against *groups* not individuals.



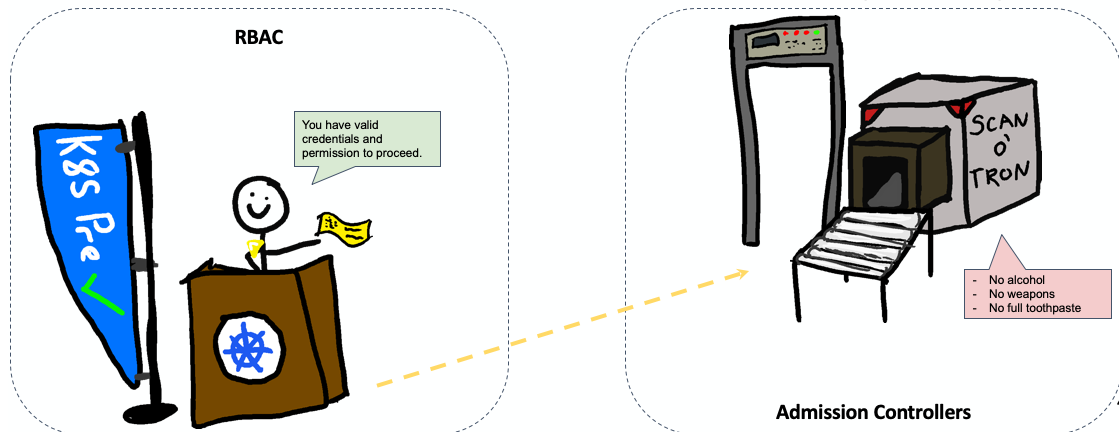
- Maintaining bindings between subjects and roles is complicated and decays quickly.
  - E.g. What happens when Bob quits? Dave switches jobs to SRE? Emily switches to become a SWE? Maybe you need to grant all SWEs a new role? It's hard to get right.
- Instead of binding your subjects (developers, admins, robots) to roles within your cluster use *groups* as a "middleman".
- [In a \[Cluster\]RoleBinding's Subject](#), you would specify `kind: group` and `name: <name-of-group>`.
- It is much easier to think about "what roles does the *SRE* group need" in isolation. Then think about "who should be in the *SRE* group?". It is much easier to change group memberships without worrying about role bindings.
- Similar with the identity section, your "group" infrastructure depends on your

organization and it is up to your [webhook authenticator](#) to tell your cluster you@mygsuitecompany.com is a member of group "sre" or whatever.

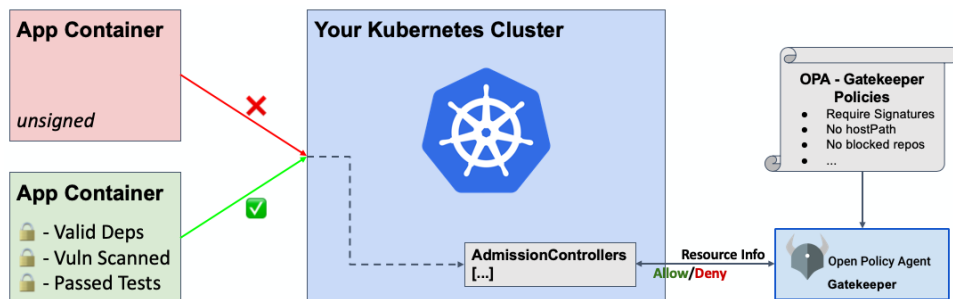
- E.g. you could use [Google Groups](#) or something else (feel free to suggest).

todo

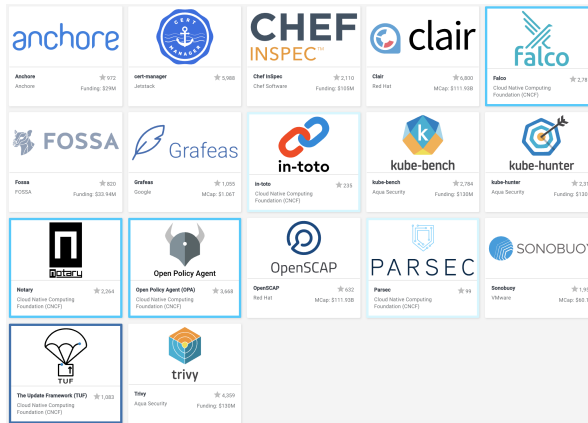
Use a policy agent to protect your cluster.



- A policy agent can at runtime enforce all the best practices outlined in this doc. You can configure it to scan all API objects being created (like pods and RBAC policies) to ensure they meet any arbitrary requirements (valid signatures, no hostPath, no hostNetwork, and no privileged containers, ect).
- An analogy from the picture above: At an airport, the ticket agent is kind of like the authn/authz layer, they check your ID and ticket and allow you to proceed. The metal detector and bag scanner are kind of like a policy agent, they alert/enforce rules like "no weapons or large fluids" to everybody who passes through, no matter how privileged.
- Policy agents typically work as [Kubernetes Admission Controllers](#), live within your cluster, and scan all API resources.



- There are many policy agents that do all kinds of security inspections / enforcement. [From the CNCF landscape](#):



- I like OPA Gatekeeper <https://github.com/open-policy-agent/gatekeeper> it is very easy to configure and get up and running.

That is it. Hopefully you found this helpful. :)