Exploring the Security of Helm

Published by **Angus Lees** on December 6, 2017

#helm #security #kubernetes

Bitnami has been a part of the Helm community for a long while, but I personally started looking at Helm only a few weeks ago in the context of our work on kubeapps - a package agnostic launchpad for kubernetes apps. I was writing the manifests to deploy all the necessary tooling and started digging into a secure deployment configuration for Helm.

The executive summary is that I felt that Helm/tiller needed to be properly configured to be operated securely in a shared/production environment.

This post is a breakdown of some of what I've learned, and some of the approaches available to secure tiller within a cluster. The wider discussion about these issues has led to a good summary doc from the Helm folks - in the following I'm going to expand on many of the details. I have tried to include plenty of worked examples so readers can see how to explore their own tiller setup before and after making any changes.

Breaking Down the Problem

There are several angles from which someone might try to abuse Helm/Tiller:

- A **privileged API user**, such as a cluster-admin. We actually *want* these users to have access to the full power and convenience of helm charts.
- A **low-privilege API user**, such as a user who has been restricted to a single namespace using RBAC. We would *like* to allow these users to install charts into their namespace, but not affect other namespaces.
- An **in-cluster process**, such as a compromised webserver. There is no reason these processes should install helm charts, and we want to prevent them from doing so.
- A **hostile chart author** can create a chart containing unexpected resources. These can either escalate one of the other groups above, or run other malicious jobs.

It is easy to mash all the issues and conversations together, but I recommend keeping the above separation in mind when considering your own needs and solutions.

In the rest of this blog, I will show you what each of these risk actually mean and show you a remediation path.

Helm Architecture

First, some required background. "Helm" is a project and a command line tool to manage bundles of Kubernetes manifests called charts. `helm` (the command line tool) works by talking to a server-side component that runs in your cluster called `tiller`. Tiller by default runs as the `tiller-deploy` `Deployment` in `kube-system` namespace. Much of the power and flexibility of Helm comes from the fact that **charts can contain just about any Kubernetes resource**. There are good docs on the helm website if you want to read about any of this in more depth.

The `helm` -> `tiller` communication happens over gRPC connections. These are not your usual HTTP, but you can think of it in a similar way: `helm` sends a command and arguments down the gRPC TCP channel, tiller does the work, and then returns a result which `helm` usually displays to the user in some form.

One of the first concerning aspects of Tiller is that it is able to create just about any Kubernetes API resource in a cluster. It has to, because that is its job when installing a chart! **Tiller performs no authentication by default** - so if we can talk to tiller, then we can tell it to install just about anything.

In-cluster attacks

This is one of the most alarming of the cases above, so we will focus on it first. The design of Kubernetes RBAC, namespaces, pods, etc strives to isolate serving jobs from each other. If a webserver is compromised (for example), you really want that rogue process to stay contained and not be able to easily escalate and exploit the rest of your cluster.

By default tiller exposes its gRPC port inside the cluster, without authentication. That means *any* pod inside your cluster can ask tiller to install a chart that installs new ClusterRoles granting the local pod arbitrary, elevated RBAC privileges: Game over, thanks for playing.

What are we talking about, exactly?

Here is **a complete worked example**, so you can see what's involved, and how to test your own cluster.

First, install tiller. We're going to install tiller using the defaults.

```
$ helm init
$HELM_HOME has been configured at /home/gus/.helm.
```

```
Tiller (the Helm server-side component) has been installed into your Kubernetes Cluster.
Happy Helming!
```

What does this do? `helm init` creates a `Service` and a `Deployment`, both called `tiller-deploy` in the `kube-system` namespace. Note the default tiller port is 44134 - we're going to use that later.

```
$ kubectl -n kube-system get svc/tiller-deploy deploy/tiller-deploy
NAME                TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)     AGE
svc/tiller-deploy   ClusterIP   10.0.0.216   <none>        44134/TCP   1h

NAME                   DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/tiller-deploy   1         1         1            1           1h
```

A `ClusterIP Service` creates a hostname and port that can be accessed from elsewhere inside the cluster. *What happens if we poke at those directly?*

For now, lets test our access to tiller from inside the cluster. The following sets up a basic shell environment running in a temporary pod in the "default" namespace. Importantly, this pod has no special privileges.

```
$ kubectl run -n default --quiet --rm --restart=Never -ti --image=anguslees/helm-security-post incluster
root@incluster:/# helm version
Client: &version.Version{SemVer:"v2.7.2", GitCommit:"8478fb4fc723885b155c924d1c8c410b7a9444e6", GitTreeState:"clean"}
Error: cannot connect to Tiller
```

Good! "Error: cannot connect to Tiller" is exactly what we want to see! But wait, we saw earlier that tiller installs a service by default. *What happens if we try to talk to that host/port directly?*

```
root@incluster:/# telnet tiller-deploy.kube-system 44134
Trying 10.0.0.216...
Connected to tiller-deploy.kube-system.svc.cluster.local.
Escape character is '^]'.
^]
telnet> quit
Connection closed.
```

Huh, that looks like I *can* connect to tiller. Sure enough, the earlier "cannot connect" error wasn't quite true and what actually failed was the `helm` CLI trying to *discover* tiller. If we bypass the discovery step, **the actual connection to tiller works just fine**:

```
root@incluster:/# helm --host tiller-deploy.kube-system:44134 version
Client: &version.Version{SemVer:"v2.7.2", GitCommit:"8478fb4fc723885b155c924d1c8c410b7a9444e6", GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.7.0", GitCommit:"08c1144f5eb3e3b636d9775617287cc26e53dba4", GitTreeState:"clean"}
```

Just to drive home exactly what this means, here's the rest of an example exploit. I created a very simple chart that just binds the "default" service account to a wildcard "allow everything" RBAC ClusterRole. You can see that installing this chart immediately allows my previously-unprivileged pod to **read the secrets from `kube-system`** (for example).

```
root@incluster:/# kubectl get secrets -n kube-system
Error from server (Forbidden): secrets is forbidden: User "system:serviceaccount:default:default" cannot list secrets in the namespace "kube-system"

root@incluster:/# helm --host tiller-deploy.kube-system:44134 install /pwnchart
NAME:   dozing-moose
LAST DEPLOYED: Fri Dec  1 11:24:22 2017
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1beta1/ClusterRole
NAME           AGE
all-your-base  0s

==> v1beta1/ClusterRoleBinding
NAME          AGE
belong-to-us  0s


root@incluster:/# kubectl get secrets -n kube-system
NAME                                     TYPE                                  DATA      AGE
attachdetach-controller-token-v2mbl      kubernetes.io/service-account-token   3         1d
bootstrap-signer-token-svl22             kubernetes.io/service-account-token   3         1d
<etc ...>
```

How can we close down the tiller port?

Luckily this is easy for most users. If you *only* use tiller through the `helm` CLI tool, **this is sufficient to secure a default tiller install** from attacks inside your cluster:

```
kubectl -n kube-system delete service tiller-deploy
kubectl -n kube-system patch deployment tiller-deploy --patch '
spec:
  template:
    spec:
      containers:
        - name: tiller
          ports: []
          command: ["/tiller"]
          args: ["--listen=localhost:44134"]
'
```

What does this do? This patches the tiller `Deployment` to run with `--listen=localhost:44134` flag. This flag causes tiller to listen for gRPC connections **only on the localhost address inside the pod**. The `Service` and `ports` declaration are now useless, so we remove them too just to be nice.

Note that **just removing the `Service` is not sufficient**. The tiller port is still exposed, if you know the pod address.

```
$ kubectl -n kube-system delete service tiller-deploy
service "tiller-deploy" deleted
$ kubectl get pods -n kube-system -l app=helm,name=tiller -o custom-columns=NAME:.metadata.name,PODIP:.status.podIP
NAME                             PODIP
tiller-deploy-84b97f465c-pdfdp   172.17.0.3
$ kubectl run -n default --quiet --rm --restart=Never -ti --image=anguslees/helm-security-post incluster
root@incluster:/# helm --host=tiller-deploy.kube-system:44134 version
Client: &version.Version{SemVer:"v2.7.2", GitCommit:"8478fb4fc723885b155c924d1c8c410b7a9444e6", GitTreeState:"clean"}
Error: cannot connect to Tiller
root@incluster:/# helm --host=172.17.0.3:44134 version
Client: &version.Version{SemVer:"v2.7.2", GitCommit:"8478fb4fc723885b155c924d1c8c410b7a9444e6", GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.7.0", GitCommit:"08c1144f5eb3e3b636d9775617287cc26e53dba4", GitTreeState:"clean"}
```

The last command above demonstrates that tiller is still exposed on the pod IP address. Continuing this example to show that restarting tiller with `--listen=localhost:44134` *does* restrict access:

```
$ kubectl -n kube-system patch deployment tiller-deploy --patch '
spec:
  template:
    spec:
      containers:
        - name: tiller
          ports: []
          command: ["/tiller"]
          args: ["--listen=localhost:44134"]
'

deployment "tiller-deploy" patched
$ kubectl run -n default --quiet --rm --restart=Never -ti --image=anguslees/helm-security-post incluster
root@incluster:/# helm --host=172.17.0.3:44134 version
Client: &version.Version{SemVer:"v2.7.2", GitCommit:"8478fb4fc723885b155c924d1c8c410b7a9444e6", GitTreeState:"clean"}
Error: cannot connect to Tiller
```

I have never been happier to see a "cannot connect" error message. **By listening on localhost only, tiller is no longer accessible within the cluster.**

Note that the `helm` CLI tool still works, when used by privileged users. This is because the CLI tool creates a temporary *port forward* to the tiller-deploy pod directly, and then communicates down the forwarded connection. The port-forwarded connection "pops out" in the pod's own network namespace, and so *can* connect to the pod's idea of `localhost` just fine. I'll come back to the `helm` CLI again later.

```
$ kubectl auth can-i create pods --subresource portforward -n kube-system
yes
$ helm version
Client: &version.Version{SemVer:"v2.7.0", GitCommit:"08c1144f5eb3e3b636d9775617287cc26e53dba4", GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.7.0", GitCommit:"08c1144f5eb3e3b636d9775617287cc26e53dba4", GitTreeState:"clean"}
```

What if I *need* to access tiller port in-cluster?

There are some 3rd-party services that build on tiller, such as monocular and landscaper. These fall into a few categories depending on how they work.

Some tools use `helm` CLI directly (eg: a DIY jenkins CI/CD pipeline). These will work just fine in-cluster provided the service account they are running as is able to create a port-forward to the tiller pod. See the section on low-privileged API users below.

Some tools talk to the gRPC port directly. You have two basic choices with these: you can either set up tiller's gRPC TLS authentication, or bundle all the related services into the same pod as tiller.

Tiller with TLS-authenticated gRPC

There are some good upstream docs for this, however the TLS configuration was **ineffective until v2.7.2.**

For simplicity in the following example, I've wrapped up all the TLS certificate generation in a simple script that uses `openssl` - any other method of generating these keys is fine.

```
$ : Ensure at least v2.7.2 !
$ helm version --client --short
Client: v2.7.2+g8478fb4
$ ./tls.make
(output skipped)
$ kubectl delete deploy -n kube-system tiller-deploy
deployment "tiller-deploy" deleted
$ helm init --tiller-tls --tiller-tls-cert ./tiller.crt --tiller-tls-key ./tiller.key --tiller-tls-verify --tls-ca-cert ca.crt
$HELM_HOME has been configured at /home/gus/.helm.
(Use --client-only to suppress this message, or --upgrade to upgrade Tiller to the current version.)
Happy Helming!
```

What has this done? This deletes the old tiller Deployment (to clean up after our earlier `--listen=localhost:44134` change), and then installs a fresh tiller with the **gRPC port open inside the cluster, but *secured with TLS certificates**. If you are curious, the tiller TLS files have been uploaded into the `tiller-secret` Secret in `kube-system` namespace.

Now we can retry our earlier in-cluster test:

```
$ kubectl run -n default --quiet --rm --restart=Never -ti --image=anguslees/helm-security-post incluster
root@incluster:/# telnet tiller-deploy.kube-system 44134
Trying 10.0.0.6...
Connected to tiller-deploy.kube-system.svc.cluster.local.
Escape character is '^]'.
^]
telnet> quit
Connection closed.
root@incluster:/# helm --host=tiller-deploy.kube-system:44134 version
Client: &version.Version{SemVer:"v2.7.2", GitCommit:"8478fb4fc723885b155c924d1c8c410b7a9444e6", GitTreeState:"clean"}
Error: cannot connect to Tiller
```

Note the port is open (`telnet` can connect to it), but the tiller server immediately closes the gRPC connection because we don't have a suitable TLS certificate. Good.

The same is true even when the client tries to connect from localhost over the port-forward. **With TLS enabled, *everyone* has to use TLS certs.**

```
$ helm version
Client: &version.Version{SemVer:"v2.7.2", GitCommit:"8478fb4fc723885b155c924d1c8c410b7a9444e6", GitTreeState:"clean"}
Error: cannot connect to Tiller
$ ./tls.make myclient.crt myclient.key
(output skipped)
$ helm --tls --tls-ca-cert ca.crt --tls-cert myclient.crt --tls-key myclient.key version
Client: &version.Version{SemVer:"v2.7.2", GitCommit:"8478fb4fc723885b155c924d1c8c410b7a9444e6", GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.7.2", GitCommit:"8478fb4fc723885b155c924d1c8c410b7a9444e6", GitTreeState:"clean"}
$ cp ca.crt $(helm home)/ca.pem
$ cp myclient.crt $(helm home)/cert.pem
$ cp myclient.key $(helm home)/key.pem
$ : The following looks like a bug!
$ helm version --tls
could not read x509 key pair (cert: "/cert.pem", key: "/key.pem"): can't load key pair from cert /cert.pem and key /key.pem: open /cert.pem: no such fil
$ export HELM_HOME=$(helm home)
$ helm version --tls
Client: &version.Version{SemVer:"v2.7.2", GitCommit:"8478fb4fc723885b155c924d1c8c410b7a9444e6", GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.7.2", GitCommit:"8478fb4fc723885b155c924d1c8c410b7a9444e6", GitTreeState:"clean"}
```

Note that we now have to present a valid TLS certificate, even when using the port-forward method. We can save some typing by copying the TLS files into the helm config dir, but note there appears to be a bug that requires `HELM_HOME` to be explicitly set, and we still need to use `helm --tls` everywhere. This is annoying, but is **necessary for setups where tiller needs to be exposed** to 3rd party services within the cluster.

Alternative: Sidecars

Some tiller clients (eg monocular) do not support TLS-secured gRPC connections yet. The approach we took with kubeapps dashboard was to move the `monocular-api` component and tiller **into the same pod**, and use the earlier `--listen` flag to restrict tiller to localhost within that pod.

The details are messy but straightforward. You can see the relevant portion of the kubeapps manifests if you are curious. Note these configs are written using kubecfg (from the ksonnet project).

Limited Access for Low-privileged Users

The above is great, but completely ignores authorization. Even with TLS configured, tiller is still all-or-nothing: If you can send commands, then, you can install anything you want and effectively own the cluster.

What if you *want* to allow your users to use helm, but still restrict what a chart can do?

Tiller per Namespace

The basic approach is to use RBAC to limit the tiller service account. We can then **have many tiller instances**, each running with different service account, and different RBAC restrictions. By controlling who is allowed to access which tiller instance, we can control what those users are allowed to do *via* tiller.

One obvious example is providing a tiller instance per namespace, limited to only creating resources in *that* namespace.

```
$ NAMESPACE=default
$ kubectl -n $NAMESPACE create serviceaccount tiller
serviceaccount "tiller" created
$ kubectl -n $NAMESPACE create role tiller --verb '*' --resource 'services,deployments,configmaps,secrets,persistentvolumeclaims'
role "tiller" created
$ kubectl -n $NAMESPACE create rolebinding tiller --role tiller --serviceaccount ${NAMESPACE}:tiller
rolebinding "tiller" created
$ kubectl create clusterrole tiller --verb get --resource namespaces
clusterrole "tiller" created
$ kubectl create clusterrolebinding tiller --clusterrole tiller --serviceaccount ${NAMESPACE}:tiller
clusterrolebinding "tiller" created
$ helm init --service-account=tiller --tiller-namespace=$NAMESPACE
$HELM_HOME has been configured at /home/gus/.helm.

Tiller (the Helm server-side component) has been installed into your Kubernetes Cluster.
Happy Helming!
$ kubectl -n $NAMESPACE delete service tiller-deploy
service "tiller-deploy" deleted
$ kubectl -n $NAMESPACE patch deployment tiller-deploy --patch '
spec:
  template:
    spec:
      containers:
        - name: tiller
          ports: []
          command: ["/tiller"]
          args: ["--listen=localhost:44134"]
'
deployment "tiller-deploy" patched
```

This creates a new service account, with the RBAC permissions to do anything to services, deployments, configmaps, secrets and PVCs (extend as desired) within the specfied namespace. We then install a new tiller instance in this namespace using the new service account, and apply our earlier patches to restrict the tiller port to localhost.

Let's test it out:

```
$ helm --tiller-namespace=$NAMESPACE install stable/wordpress
NAME:   intent-mite
LAST DEPLOYED: Sun Dec  3 13:28:56 2017
NAMESPACE: default
STATUS: DEPLOYED
```

```
(snip)

$ helm --tiller-namespace=$NAMESPACE install ./pwnchart
Error: release boisterous-quetzal failed: clusterroles.rbac.authorization.k8s.io is forbidden: User "system:serviceaccount:default:tiller" cannot create
```

Success! A "nice" single-namespace chart like wordpress installed successfully, but we were **unable to install a chart that tried to modify a global resource**.

Note the above setup needs to be repeated for *every* namespace (or desired tiller isolation boundary). Each `helm` CLI user will also need to be able to find and port-forward to "their" tiller instance. In the RBAC rule language, this means the ability to "list" "pods" and "create" a "pods/portforward" resource.

The downside of this approach is having to manage the setup explicitly for each namespace, and the cumulative runtime overhead of all those separate tiller instances. In our Bitnami internal development cluster (for example), we have a namespace for each developer and this approach translates to almost an entire VM dedicated to just running tillers.

Helm CRD

With Custom Resource Definitions (CRDs) becoming a standard extension mechanism for Kubernetes, I wanted to see what Helm would look like as a conventional Kubernetes addon. So I wrote a simple Helm CRD controller. This allows you to **manage Helm charts by creating regular Kubernetes resources** instead of using the `helm` CLI and sending gRPC commands.

Using Kubernetes resources has **a number of benefits**:

- The tiller gRPC channel is not exposed
- HelmRelease resource can be restricted using RBAC as usual
- Integrates with other YAML-based tools and declarative workflows, like `kubectl` and kubecfg
- No need for the `helm` CLI for simple operations

The helm-crd controller also comes with an optional `kubectl` plugin to simplify command-line use, but it can also be operated directly with regular `kubectl` and YAML files. Using the plugin looks like this:

```
$ mkdir -p ~/.kube/plugins/helm
$ pushd ~/.kube/plugins/helm
$ wget \
 https://raw.githubusercontent.com/bitnami/helm-crd/master/plugin/helm/helm \
 https://raw.githubusercontent.com/bitnami/helm-crd/master/plugin/helm/plugin.yaml
$ chmod +x helm
$ popd
$ kubectl delete deploy -n kube-system tiller-deploy
deployment "tiller-deploy" deleted
$ kubectl plugin helm init
customresourcedefinition "helmreleases.helm.bitnami.com" created
deployment "tiller-deploy" created
```

This installs the `kubectl` plugin, and then uses the plugin to install tiller, the helm-crd controller, and `HelmRelease` custom resource definition. Note that helm-crd currently always installs tiller into `kube-system`, with the CRD controller and tiller running in the same pod and the tiller gRPC port restricted to localhost (as described earlier).

```
$ kubectl plugin helm install mariadb --version 2.0.1
helmrelease "mariadb-5w9hd" created
$ kubectl get helmrelease mariadb-5w9hd -o yaml
apiVersion: helm.bitnami.com/v1
kind: HelmRelease
metadata:
  clusterName: ""
  creationTimestamp: 2017-12-03T03:13:40Z
  deletionGracePeriodSeconds: null
  deletionTimestamp: null
  generateName: mariadb-
  name: mariadb-5w9hd
  namespace: default
  resourceVersion: "209450"
  selfLink: /apis/helm.bitnami.com/v1/namespaces/default/helmreleases/mariadb-5w9hd
  uid: f647e29f-d7d7-11e7-be67-525400793f03
spec:
  chartName: mariadb
  repoUrl: https://kubernetes-charts.storage.googleapis.com
  values: ""
  version: 2.0.1
```

The controller has only seen light testing so far, but is simple, and is usable right now. Currently it does not deal with the "hostile chart" problem, but I have a new tiller feature that will allow me to create a separate "cluster level" CRD resource, much like the Role/ClusterRole split in RBAC.

Additional usage and feedback is valuable, and we hope to be able to use this experience to inform the "Helm v3" discussions starting in February.

Conclusion

This was a long doc. The basic take-away is that when using helm in a shared/production environment, **the `helm init` default installation is not sufficient**.

- If you have *no 3rd-party services* that talk to tiller: Restart tiller with `--listen=localhost:44134` flag.

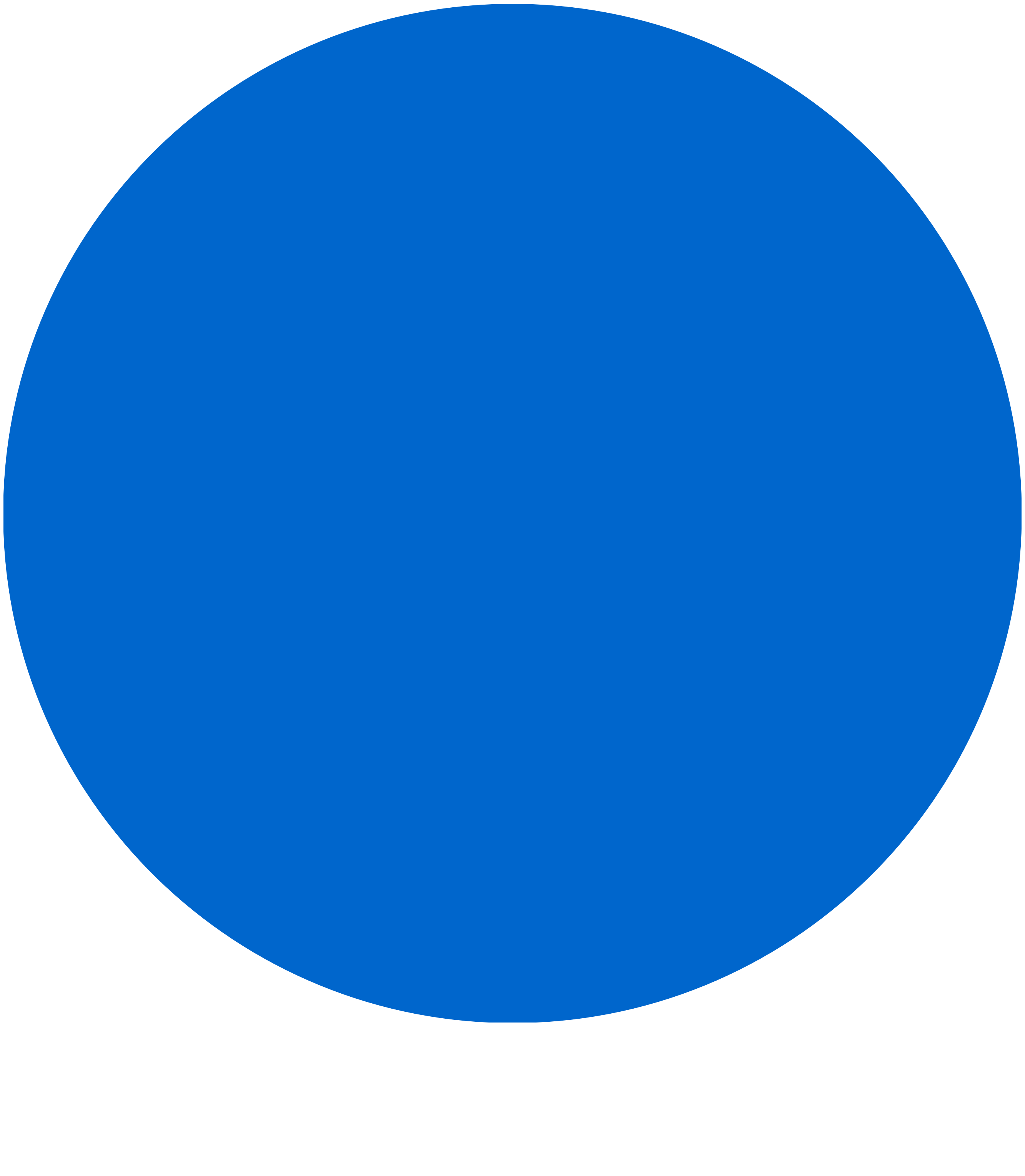
- If you have *3rd-party services* that talk to tiller: Configure TLS authentication.

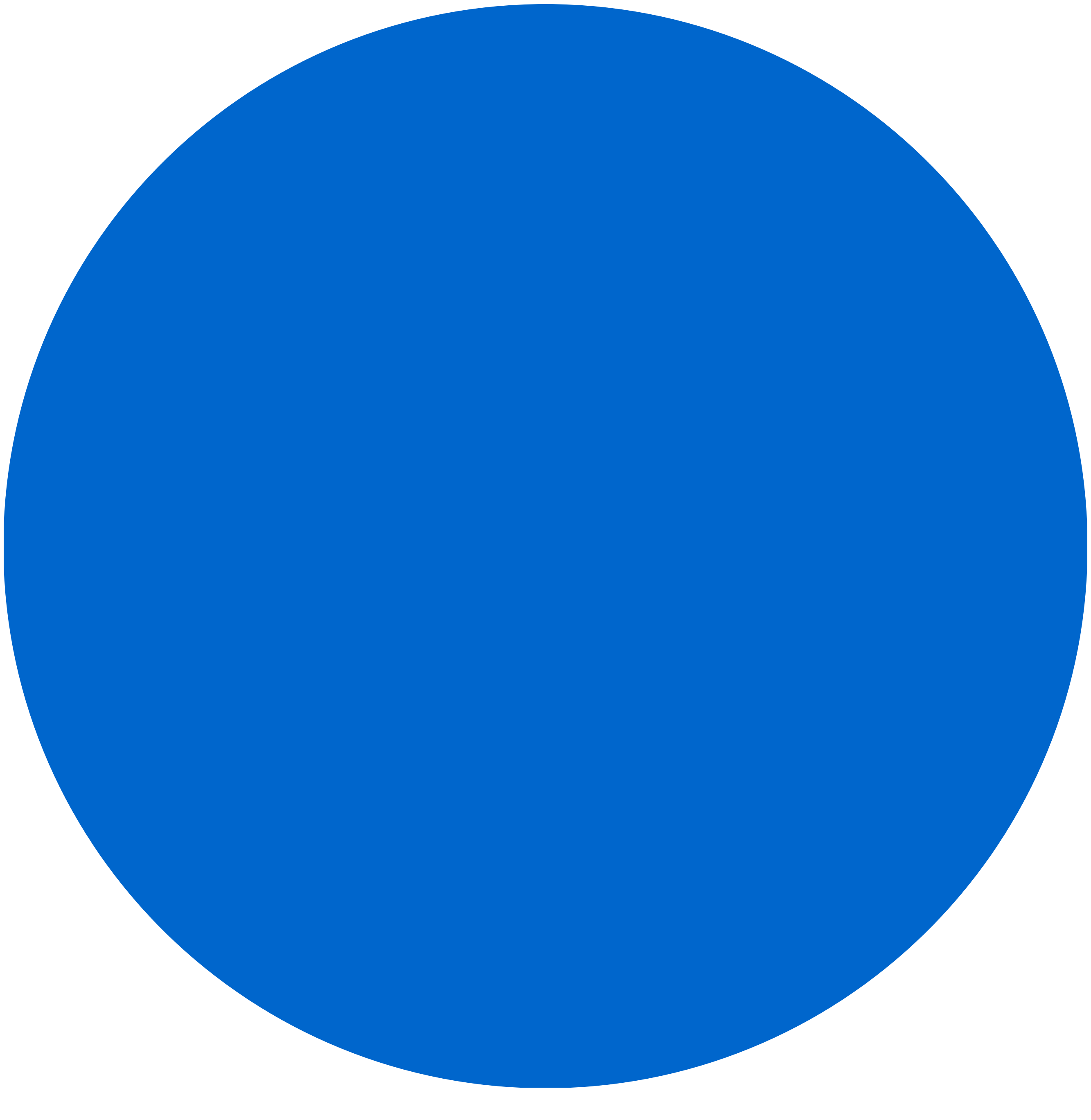
- If you need to offer *different levels of access* via tiller: Complicated. Multiple tiller instances, or Helm-crd.

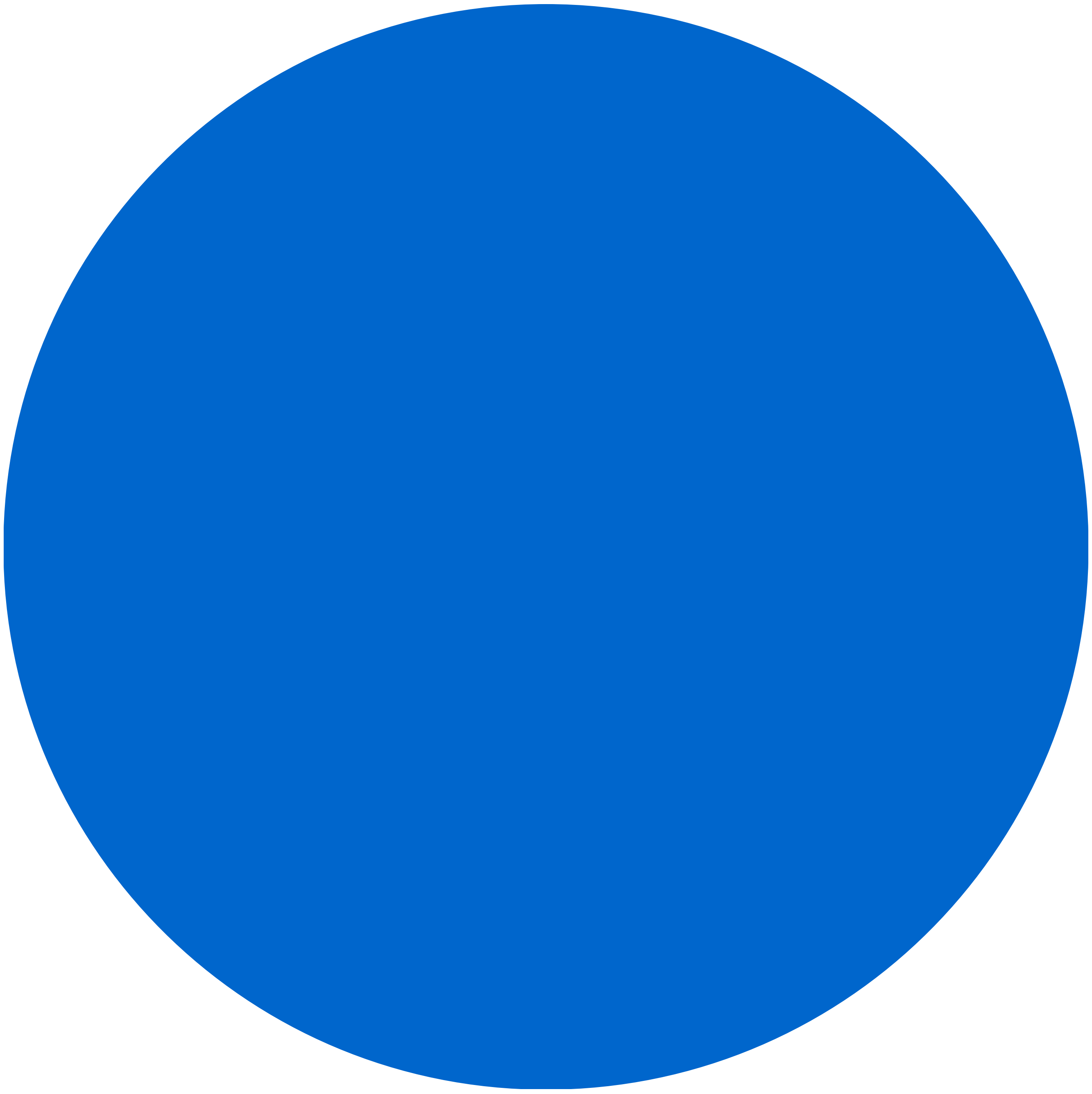
Did you like this article? Share it with your friends.

About the author

Angus Lees







Products

- Application Catalog

Kubernetes

- Kubeapps
- BKPR

Solutions

- Application Packaging
- Cloud Migration
- Kubernetes
- Partners

Company

- About Us
- Careers
- Resources
- Newsroom
- Contact

Legal

- Terms of Use
- Privacy
- Trademark

Support

- Docs
- Community
- Helpdesk
- Webinars
- Training