Creating Users/Groups and applying Role Based Access Controls in Kubernetes the Hard Way.

Introduction:

Let's get one thing clear, creating users, groups and implementing role based access controls on your newly created kubernetes cluster (k8s) for short is the last thing on your mind. You spent all this time achitecting all components in your cluster as well as your ingress from outside the organization. Or maybe this is just a home based test cluster for you to keep your skills sharp. Either way this is an import practice to be prepared for. Your security team will always ding you for this when you go to get your cluster approved for production use. Most likely you will hook your cluster to an enterprise authentication service such as Keycloak or a Windows Domain Trust. Both of those solutions are outside the scope of this article, we will go over how to:

- Setup new users via Certificate Signing Requests
- Creating Roles/Role-Bindings
- Configuring your kube.config settings
- Create ServiceAccounts

If you are planning on or working towards obtaining your Certified Kubernetes Administrator title, then this is a great article for reference. Granted this covers only the RBAC topic but great for having under your belt. I remember having to implement all of our RBAC rules a year after we stood up all of our various clusters when I worked for RedHat at the Securities and Exchange Commission in Washington D.C. We had various types of users we had to account for. We had Developers, Cluster-Admins, Admins, Cluster-Viewers. All with a different set of privileges. Thankfully once you get all the role and role-bindings figured out for one cluster then you can easily apply them to the other clusters with a simple `kubectl create –f <pathtoRole.yamls>`. Just as in most of linux type work, the hard work comes first and then you are either able to script it out, or save that config in confluence or some GIT repo.

I also want to cover the use cases for Service Accounts. There may be some instances where you will want, or more importantly need to use an account that isn't linked to a human entity user. I have personally used a service account via authentication token to preform image pruning nightly. Very useful and highly recommended.

Terminology:

First I want to get everyone on the same page with not only the various terms, or in the kubernetes world the resources that are configured to accomplish RBAC.

Role or ClusterRole: Contains rules that represent a set of permissions. Permissions are purely additive (there are no "deny" rules)

Whenever you create a Role, they will always set permissions for a specific namespace. This is why you must ensure you are setting these Roles to the correct namespace. If you don't do this, they will be applied to the default project or whichever project your context is currently set to.

ClusterRoles have other uses worth discussing:

- You can use them for a single namespace
- You can use them across all namespaces
- Define cluster-scoped resources

Sample Role:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: stage-1-admin
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods,deployments,replicasets,services"]
  verbs: ["get", "watch", "list","create",”delete”]
```

Here is what it looks like when I describe the role I just created:

```
[root@controlplane ~]# kubectl describe  roles red-team-ops
```

```
Name:         stage-1-admin
Labels:       <none>
Annotations:  <none>
PolicyRule:
  Resources                               Non-Resource URLs
Resource Names  Verbs
  ---------                               -----------------  -----
---------  -----
  pods,deployments,replicasets,services  []                 []
[get watch list create delete]
```

RoleBindings: RoleBindings bind whatever roles you create to subjects in the cluster which are users, groups or service accounts. Usernames in Kubernetes are stored as strings, and it's important to be mindful of what naming convention you use. For instance the prefix of `system:` is reserved for Kubernetes. The above role will grant a user the permissions to “get, watch, list, create” to “pods, deployments, replicasets, services”. All other resources are blocked from the user. You can use this to give fine grained access rights.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: evan-binding
  namespace: default
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: stage-1-admin
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: evan
```

```
[root@controlplane ~]# kubectl describe rolebinding evan-binding
Name:         evan-binding
Labels:       <none>
Annotations:  <none>
Role:
  Kind:  Role
  Name:  stage-1-admin
Subjects:
  Kind  Name  Namespace
  ----  ----  ---------
  User  evan
```

ClusterRoleBindings: Can also be used in this same way, but you will be able to grant permissions cluster wide instead of being restricted to a single namespace. Be careful to watch out for the namespace field in a ClusterRoleBinding since it will restrict permissions to that namespace value. I use the ClusterRoleBindings to create my admin users in the cluster forexample.

Example ClusterRoleBinding:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: read-secrets-global
subjects:
- kind: Group
  name: manager
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: secret-reader
  apiGroup: rbac.authorization.k8s.io
```

Please take note that once this resource is created you cannot update or change the `roleRef` field. You must delete it and recreate the clusterrolebinding. This is a security feature to enforce the bindings and to not allow an entity to update this resource.

We have defined a role that would constrain the user “evan” to only be able to perform a couple of actions or “verbs” on a handful of resources within the default namespace.

Now that we went over and covered the basics, we can move to actually configuring these settings in our cluster.

Aggregated ClusterRoles: A Controller running in the cluster watches out for these very resources. If you are so inclined you may bundle or aggregate several ClusterRoles together as long as the `aggregationRule` is set. The `aggregationRule` defines a label selector that the controller uses to match other ClusterRole objects that should be combined into the `rules` field.

Here is an example aggregated ClusterRole:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRolemetadata:
  name: monitoring
aggregationRule:
  clusterRoleSelectors:
- matchLabels:      rbac.example.com/aggregate-to-monitoring: 
"true"
rules: [] # The control plane automatically fills in the rules
```

Service Accounts: A service account provides an identity for processes that run in a Pod. When you (a human) access the cluster (for example, using `kubectl`), you are authenticated by the apiserver as a particular User Account (currently this is usually `admin`, unless your cluster administrator has customized your cluster). Processes in containers inside pods can also contact the apiserver. When they do, they are authenticated as a particular Service Account (for example, `default`).

kubectl create serviceaccount cloud-admin -n default

```
[root@controlplane ~]# kubectl describe sa cloud-admin
Name:                cloud-admin
Namespace:           default
Labels:              <none>
Annotations:         <none>
Image pull secrets:  <none>
Mountable secrets:   cloud-admin-token-lt2q6
Tokens:              cloud-admin-token-lt2q6
Events:              <none>

[root@controlplane ~]# kubectl get secret
NAME                                TYPE
DATA   AGE
cloud-admin-token-lt2q6             kubernetes.io/service-
account-token   3      2m29s
default-token-22gzf                 kubernetes.io/service-
account-token   3      49d
```

As you see in the above commands I created a service account named cloud-admin in the default namespace, described the service account and could also see its secret token that was generated. We will use this token as its authentication into the cluster.

```
kubectl get secret cloud-admin-token-lt2q6 -o yaml
```

```
apiVersion: v1
data:
  ca.crt: LS0tLS1CRUdJTiBDR.........
  namespace: ZGVmYXVsdA==
  token: ZXlKaGJHY2lPa.........

Then:
  #   export TOKEN=’ ZXlKaGJHY2lPa.........’
  #   echo $TOKEN | base64 -d
```

Or you can use this method:
`kubectl get secret cloud-admin-token-lt2q6 -o jsonpath='{.data.token}' | base64 -d`

Copy the token value from either commands just don’t include the word “token:” just the value. You still do have to create the ClusterRole and ClusterRoleBinding, don’t forget.

As I mentioned earlier, I have used service accounts to kick off a prune job within the cluster for maintenance. Since those jobs require cluster-admin or something comparable, that’s why I chose this method. Maybe you want to use Jenkins or gitlab to perform some tasks. You would’t want it to use god-mode of the cluster-admin user; you would want Jenkins to have its own account with its own specific grouping of permissions.

Here is a perfect example of someone else actually going through the whole process for Jenkins and cloud-bees:

https://support.cloudbees.com/hc/en-us/articles/360038636511-Kubernetes-Plugin-Authenticate-with-a-ServiceAccount-to-a-remote-cluster

Setting up the user Certificate Signing Request:

There are couple requirements that we first must take care of before a new user can authenticate to the cluster and API. There has to be a certificate issued by Kubernetes and the same cert present to invoke the API via kubectl. Hopefully you have your very own cluster to practice these steps, here we go:

- Create the private key
  - `openssl genrsa -out evan.key 2048`
  - `openssl req -new -key evan.key -out evan.csr`
- Create the Ciertificate Signing Request

  -
    ```
    cat <<EOF > evan.request
     apiVersion: certificates.k8s.io/v1
     kind: CertificateSigningRequest
     metadata:
       name: evan
     spec:
       groups:
       - system:authenticated
     request: <PUT BASE64 ENCODED CSR HERE>
     signerName: kubernetes.io/kube-apiserver-client
     usages:  - client auth
     EOF
    ```
  - `# cat evan.csr | base64 | tr –d “\n”`
  - Then Paste that output in the “request:” field is.
- Create CSR in the cluster
  - `# kubectl create –f evan.request`
- Veiw and approve CSR
  - `# kubectl get csr`

```
NAME  AGE  SIGNERNAME                           REQUESTOR         CONDITION
evan  103s kubernetes.io/kube-apiserver-client  kubernetes-admin  Pending
```

  - `# kubectl certificate approve evan`

certificatesigningrequest.certificates.k8s.io/evan approved

- The following command grabs the kubernetes signed certificate from use “evan” and puts it in it’s own file called evan.crt

```
[root@controlplane ~]# kubectl get csr/evan -
ojsonpath='{.status.certificate}' | base64 -d > evan.crt
```

- Next is to add the user and their credentials to your kubeconfig

```
[root@controlplane ~]# kubectl config set-credentials evan --
client-key=evan.key --client-certificate=evan.crt --embed-
certs=true
User "evan" set.
[root@controlplane ~]# kubectl config set-context evan --
cluster=kubernetes --user=evan
Context "evan" created.
[root@controlplane ~]# kubectl config use-context evan
Switched to context "evan".
[root@controlplane ~]# kubectl get po
Error from server (Forbidden): pods is forbidden: User "evan"
cannot list resource "pods" in API group "" in the namespace
"default"
```

- Since we just created the new user, they don’t have to permissions to do anything within the cluster. So next we will create the Roles and RoleBinings to allows “evan” to stop being a lazy devops guy.

```
[root@controlplane ~]# kubectl create role stage-1-admin --
namespace default --
resource=pods,deployments,services,replicasets --
verb=create,delete,get,list,watch
 role.rbac.authorization.k8s.io/stage-1-admin created
[root@controlplane ~]# kubectl create rolebinding evan-binding --
role=stage-1-admin --user=evan
rolebinding.rbac.authorization.k8s.io/evan-binding created
```

- Now by using the kubectl auth command you can see what evan can and can’t do in the cluster.

```
[root@controlplane ~]# kubectl auth can-i get po --user evan

yes

[root@controlplane ~]# kubectl auth can-i get daemonsets --user
evan
```

no

Here is the command to switch back to the kubernetes-admin user:

```
#kubectl config use-context kubernetes-admin@kubernetes --
cluster=kubernetes --user=kubernetes-admin
```

Conclusion

Knowing how to set up users in your cluster via RBAC is a good skill to keep in your back pocket. You never know if your organization wants to stick to in house solutions for solving user authentication and seperation of duties. Me personally, I would use something like Keycloak that's performing a group sync from a Domain Controller but we don't always live in a perfect world. Even if you are not managing users the way, you will still come across reasons to implement RoleBindings or ClusterRoleBindings in your cluster. Lastly, if it has the word "Cluster" in it, just think to yourself.... do they really need that much power?

I hope this helps you on your way to kubernetes greatness and securing your cluster just with the power of k8s.

References:

https://kubernetes.io/docs/reference/access-authn-authz/certificate-signing-requests/

https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/

https://kubernetes.io/docs/reference/access-authn-authz/rbac/#api-overview

https://support.cloudbees.com/hc/en-us/articles/360038636511-Kubernetes-Plugin-Authenticate-with-a-ServiceAccount-to-a-remote-cluster