Implementing a Webhook for Authorization

Learn how to implement a webhook authorization service.

We'll cover the following

Implement a webhook authorization service

A webhook authorization service is a web server, because the `kube-apiserver` invokes it through HTTPS `POST` requests.

Now, let's implement such a service step by step.

Step 1: Write a simple HTTP server

Let's write a simple HTTP server that responds with the mock authenticated user `mock` when requested for the `/authorize` resource over port `443`.

main.go

Search in directory...

/

main.go

```
1  package main
2
3  import (
4      "encoding/json"
5      "fmt"
6      "io/ioutil"
7      "log"
8      "net/http"
9
10     authorizationapi "k8s.io/api/authorization/v1beta1"
11 )
12
13 func authZ(sar *authorizationapi.SubjectAccessReview) {
14     // now we do some mock for demo
15     // Please replace this with your logic
16     if sar.Spec.User == "demo-user" {
17         sar.Status.Allowed = true
18     } else {
19         sar.Status.Reason = fmt.Sprintf("User %q is not allowed to
20             sar.Spec.User, sar.Spec.ResourceAttributes.Resource)
21     }
22 }
```

```
23
24 func helloHandler(w http.ResponseWriter, r *http.Request) {
25     log.Printf("Receiving %s", r.Method)
26
27     if r.Method != "POST" {
28         http.Error(w, "Only Accept POST requests", http.StatusMeth
29         return
30     }
31
```

HTTP server

In a real-world scenario, we only need to replace the mock codes in the function `authZ()` with our actual implementations to determine user privileges. This function should set `Status.Allowed` to `true` if the request performed by the user is allowed and `false` for invalid requests. For the denied reason, it can be set to the field `Status.Reason`.

The rest of the code is a simple HTTP handler that handles requests from the `kube-apiserver` and deserializes the payloads, which are expected to be a `SubjectAccessReview` object in JSON format, and then sends the authorization result back.

Step 2: Test it

Now, let's test the HTTP server above locally to see if the service works as expected.

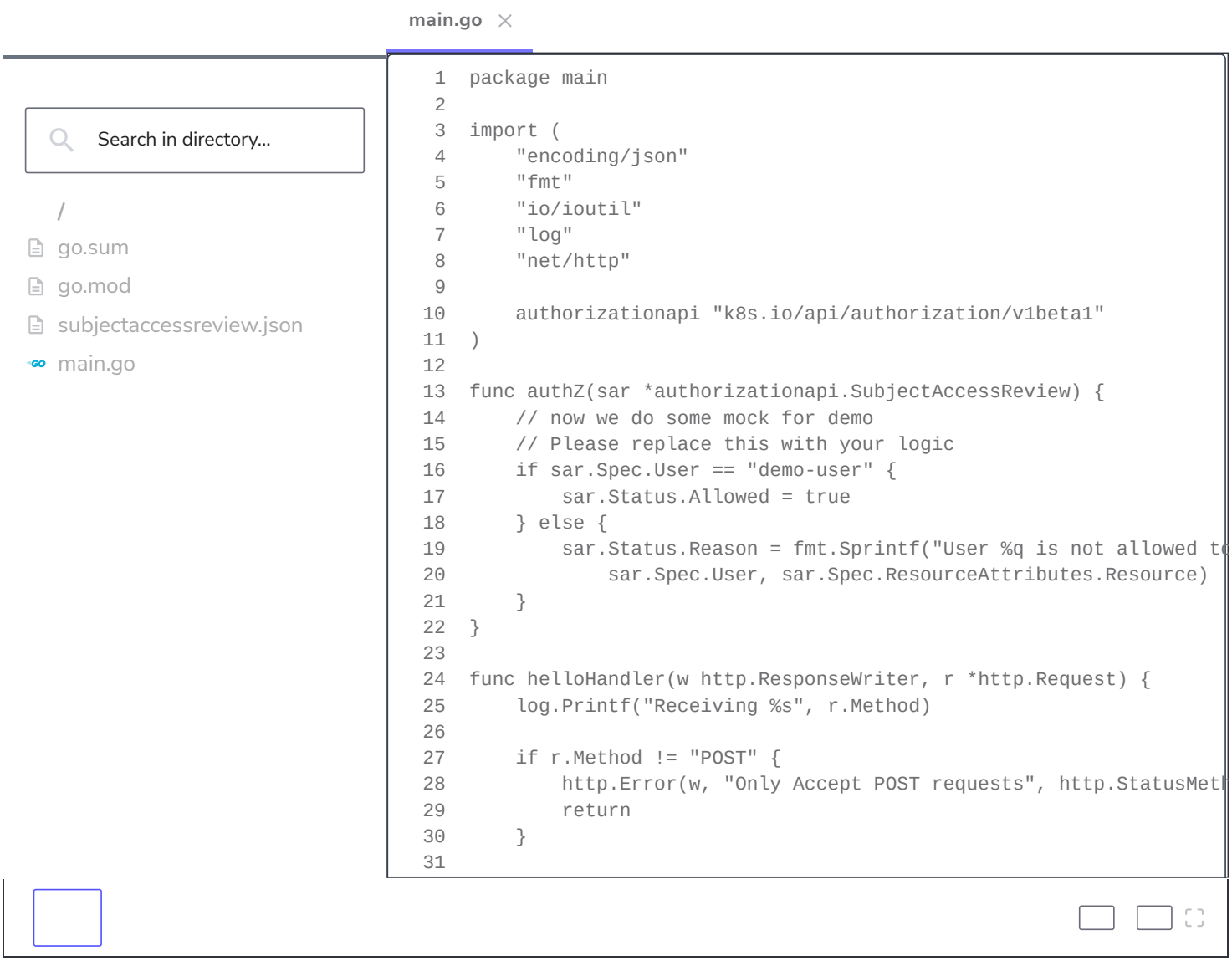
Create a file called `/root/subjectaccessreview.json` with the content below:

```
{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "spec": {
    "resourceAttributes": {
      "namespace": "demo-ns",
      "verb": "get",
      "resource": "pods",
      "version": "v1"
    },
    "user": "demo-user",
    "group": [
      "system:authenticated",
      "demo-group"
    ]
  }
}
```

The SubjectAccessReview payload

For testing this service, we'll simulate such a `POST` request from the `kube-apiserver` by making it manually from our local machine.

Now, we can click the “Run” button in the code widget below to get a terminal.



Test our webhook

In the terminal, we can start the program with the following commands:

```
go run main.go
```

Start the program

Now, we can open another terminal and send a `POST` request to test our webhook.

```
curl -k -X POST -d @subjectaccessreview.json http://localhost:443/authorize
```

Send a POST request

To better view the output, we pipe the output to jq. The response should look as follows:

```
{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "spec": {
    "resourceAttributes": {
```

```
      "namespace": "demo-ns",
      "verb": "get",
      "resource": "pods",
      "version": "v1"
    },
    "user": "demo-user",
    "group": [
      "system:authenticated",
      "demo-group"
    ]
  },
  "status": {
    "allowed": true
  }
}
```

The response

We can see that the `status.allowed` file has been set to `true`. This means that the authorization service works as expected.

Step 3: Generate and use the certificates

In production environments, using HTTPS is more secure. Now, let's create some certificates for safe serving.

```
openssl req -x509 -newkey rsa:2048 -nodes -subj "/CN=my-authz.kube-system.svc" -keyout key.pem -out cert
ls -alh
```

Generate self-signed certificates

Here, we generate a self-signed certificate with the `CN` field set to `my-authz.kube-system.svc`. We're going to run this service in Kubernetes, which is the endpoint that we want to expose. This DNS name `my-authz.kube-system.svc` indicates that there's a `Service` named `my-authz` running in the namespace `kube-system`.

Step 4: Final verison

Now, let's modify our code to enable secure serving.

main.go

Search in directory...

/

```
package main

import (
    "encoding/json"
    "flag"
    "fmt"
    "io/ioutil"
```

authz-webhook-deploy.yml
webhook-config.yml
Dockerfile
go.sum
go.mod
key.pem
cert.pem
main.go

```
    "log"
    "net/http"

    authorizationapi "k8s.io/api/authorization/v1beta1"
)

func authZ(sar *authorizationapi.SubjectAccessReview) {
    // now we do some mock for demo
    // Please replace this with your logic
    if sar.Spec.User == "demo-user" {
        sar.Status.Allowed = true
    } else {
        sar.Status.Reason = fmt.Sprintf("User %q is not allowed to
            sar.Spec.User, sar.Spec.ResourceAttributes.Resource)
    }
}

func helloHandler(w http.ResponseWriter, r *http.Request) {
    log.Printf("Receiving %s", r.Method)

    if r.Method != "POST" {
        http.Error(w, "Only Accept POST requests", http.StatusMeth
        return
    }
```

Final version of the webhook authorization

After clicking the “Run” button in the widget above, let’s first build a container image with the following command:

```
docker build -t pwk/authz-webhook:v0.1 ./
```

Build a container image

After the image has been built successfully, we can deploy the webhook in Kubernetes. Now, let’s run the following commands in the terminal above:

```
cp /usercode/webhook-config.yml /etc/kubernetes/pki/
kubectl apply -f /usercode/authz-webhook-deploy.yml
```

Deploy the webhook

In order to let our static `kube-apiserver` Pod access the webhook service with the DNS name `my-authz.kube-system.svc`, we need to export the DNS record as follows. We can run it in the terminal above.

```
echo "$(kubectl get svc -n kube-system my-authz | grep -v NAME | awk '{print $3}') my-authz.kube-system
```

Add a DNS record

Now, we need to tell the `kube-apiserver` how to use our authorization webhook. This can be configured with the flag `--authorization-webhook-config-file=<my-config-file-path>` of the `kube-apiserver`.

Then, append the webhook mode to the flag `--authorization-mode`. In this lesson, our `kube-apiserver` is running as a static Pod, whose lifecycle is managed by the `kubelet`. We only need to update the `kube-apiserver` manifest file, and then the `kubelet` will be notified of the change and restart it later.

In the terminal above, we can follow the steps below to enable webhook authorization in the `kube-apiserver`.

```
sed -i "s|Node,RBAC|Node,RBAC,Webhook|" /etc/kubernetes/manifests/kube-apiserver.yaml
sed -i "18i\    - --authorization-webhook-config-file=/etc/kubernetes/pki/webhook-config.yml" /etc/kuber
docker ps -a | grep apiserver | grep -v "pause" | awk '{print $1}' | xargs docker rm -f
```

Enable webhook authorization in the kube-apiserver

Test webhook authorizations

Let's do some tests to see if the `kube-apiserver` could successfully use the token webhook authorization service at `https://my-authz.kube-system.svc/authorize`. We use `kubectl --as` to impersonate users for operations.

Run the following commands in the terminal above:

```
kubectl get ns --as abc
```

Access the namespace for the abc user

The output should be similar to this:

```
Error from server (Forbidden): namespaces is forbidden: User "abc" cannot list resource "namespaces" in API g
```

Output for the command

Now, let's try to access the namespaces for the `demo-user`:

```
kubectl get ns --as demo-user
```

Access the namespace for the demo-user

The output should be similar to this:

```
NAME              STATUS   AGE
default           Active   10m
kube-node-lease   Active   10m
kube-public       Active   10m
kube-system       Active   10m
```

The demo-user namespace

When we impersonate a random user, such as `abc`, to list namespaces, the request is forbidden. This is exactly what we expected. On the other hand, when we use our mock user `demo-user`, listing all the namespaces is allowed. Ta-da! The authorization webhook works!

Back

How Webhook Authorization Works

Next

Quiz on Customizing AuthX