Implementing a Webhook for Token Authentication

Learn how to implement a webhook token authentication service.

We'll cover the following

- Implement a webhook token authentication service
  - Step 1: Write a simple HTTP server
  - Step 2: Test it
  - Step 3: Generate and use the certificates
  - Step 4: Final version

Implement a webhook token authentication service

A webhook token authentication 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 `/authenticate` resource over port `443`.

main.go

Search in directory...

/

main.go

```
package main

import (
    "encoding/json"
    "io/ioutil"
    "log"
    "net/http"

    authenticationapi "k8s.io/api/authentication/v1beta1"
)

func authN(tr *authenticationapi.TokenReview) {
    // Now we create a mock as an example
    // Please replace this with your logic
    tr.Status.Authenticated = true
    tr.Status.User = authenticationapi.UserInfo{
        Username: "mock",
        UID:      "mock",
        Groups:   []string{"group-mock"},
        Extra:    nil,
    }
}

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.StatusMet
29          return
30      }
31
```

The webhook HTTP server

In a real-world scenario, we only need to replace the mock codes in the function `authN` with our actual implementations to query our user-management system. This function should set a valid `UserInfo` if the provided token is legitimate. Moreover, `Status.Authenticated` should be set as well—`true` for legitimate users and `false` for invalid users. For the error message, it can be set to the field `Status.Error`.

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 `TokenReview` object in JSON format, and then sends the authentication 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/tokenreview.json` with the content below:

```
{
  "apiVersion": "authentication.k8s.io/v1beta1",
  "kind": "TokenReview",
  "spec": {
    "token": "ZGVtby11c2VyOmRlbW8tcGFzc3dvcmQ="
  }
}
```

The TokenReview payload

> **Note:** The token here is base64 encoded to handle special characters correctly.

Now, we decode the token in the file above:

```
echo "ZGVtby11c2VyOmRlbW8tcGFzc3dvcmQ=" | base64 -d
```

Token decoding

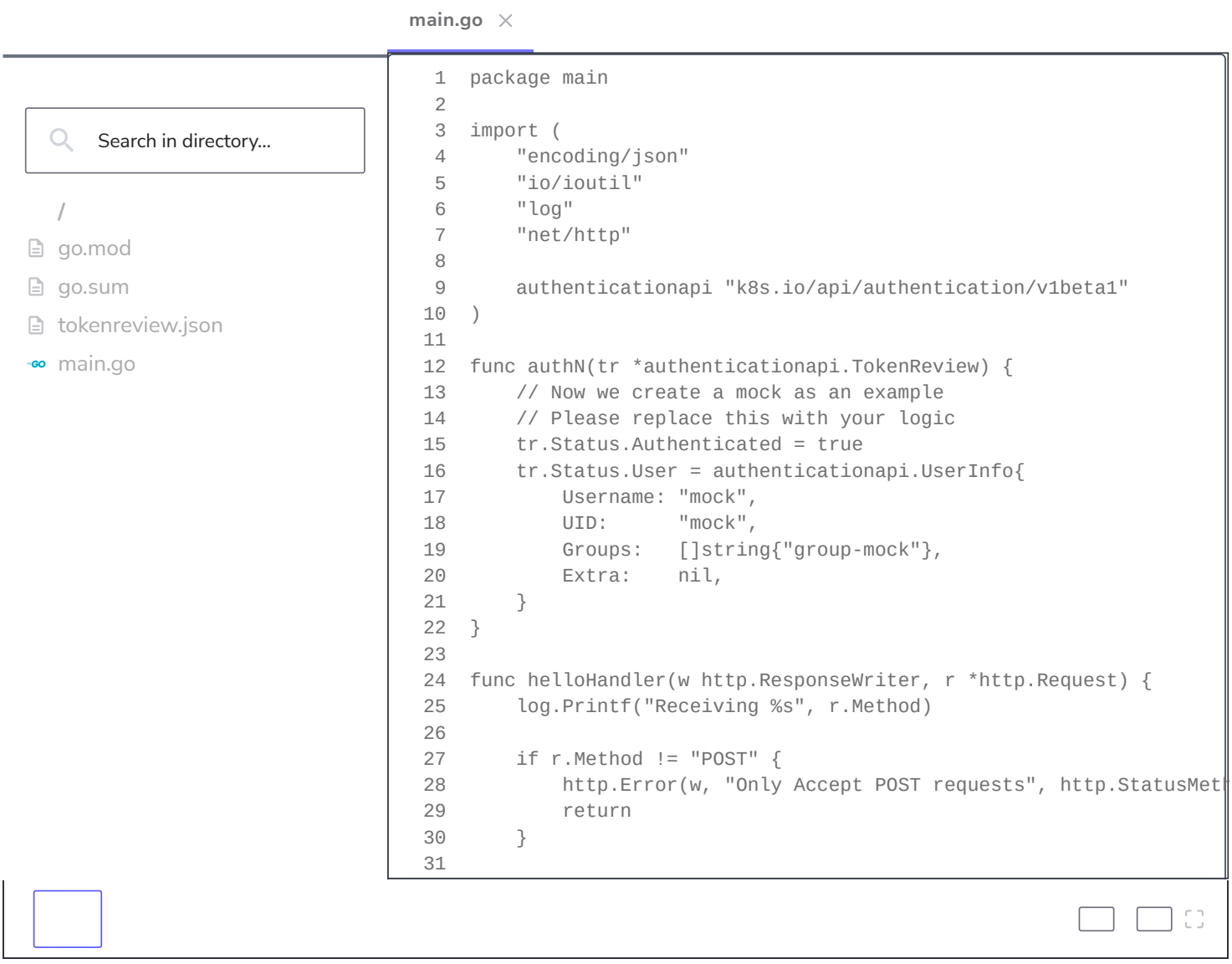
The output will be as follows:

```
1  demo-user:demo-password
```

Decode the token

For testing this service, we'll simulate 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 using the commands in the code snippet below:

```
1  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 @tokenreview.json http://localhost:443/authenticate
```

Send a POST request

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

> **Note:** jq is a tool that helps handle JSON strings.


```
{
  "kind": "TokenReview",
  "apiVersion": "authentication.k8s.io/v1beta1",
  "metadata": {
    "creationTimestamp": null
  },
  "spec": {
    "token": "ZGVtby11c2VyOmRlbW8tcGFzc3dvcmQ="
  },
  "status": {
    "authenticated": true,
    "user": {
      "username": "mock",
      "uid": "mock",
      "groups": [
        "group-mock"
      ]
    }
  }
}
```


The response

We can see the `status.authenticated` file has been set to `true`. Detailed user info is given as well. This means the authentication 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-authn.kube-system.svc" -keyout key.pem -out cert
ls -alh
```

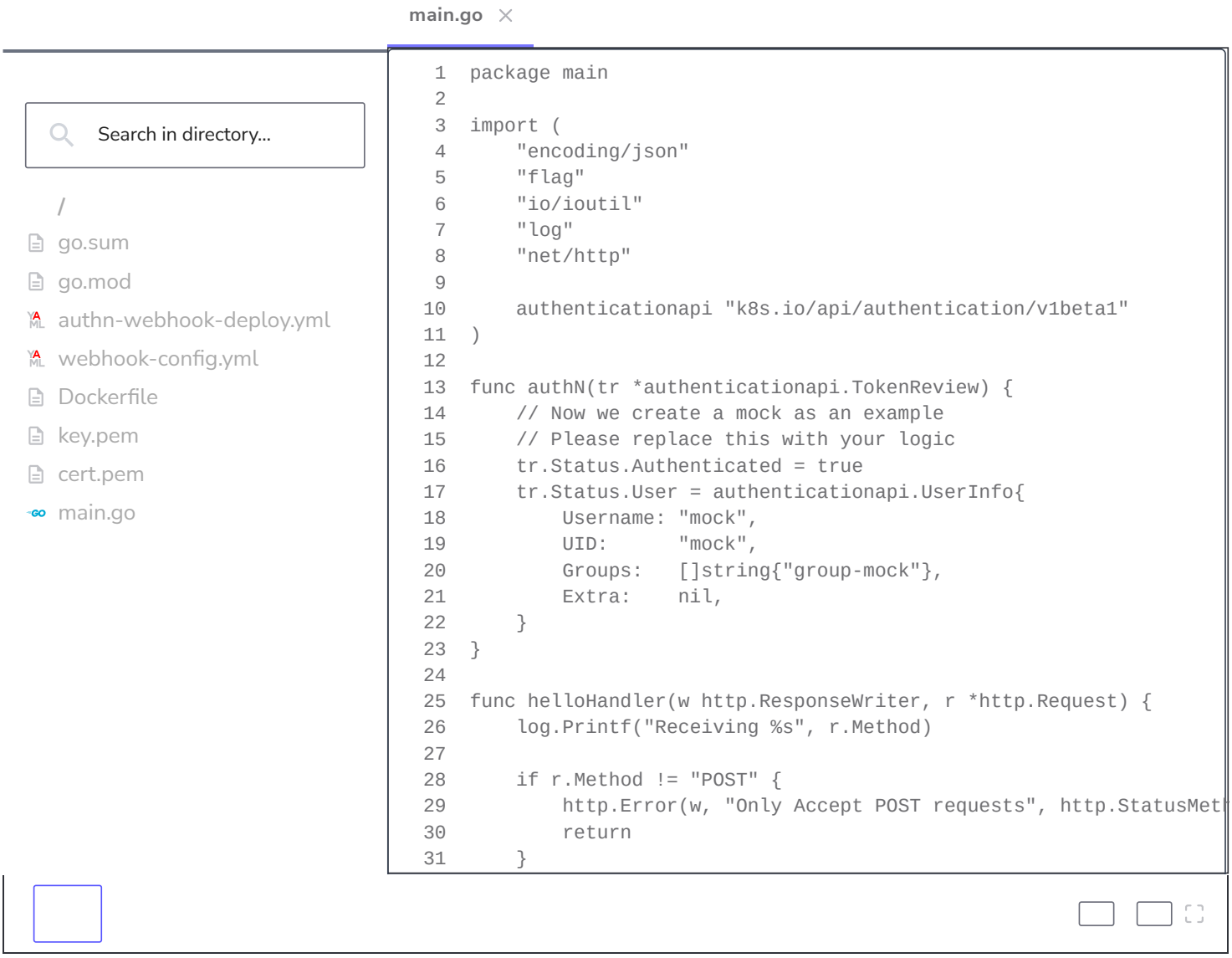
Generate self-signed certificates

Now, we should have `cert.pem` and `key.pem` in our directory.

Here, we generate a self-signed certificate with the `CN` field set to `my-authn.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-authn.kube-system.svc` indicates that there's a `Service` named `my-authn` running in the namespace `kube-system`.

Step 4: Final version

Now, let's put everything together and deploy it with secure serving.



Final version of the webhook authentication

After hitting the "Run" button in the code widget above, let's first build a container image with the following command:

```
1  docker build -t pwk/authn-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/authn-webhook-deploy.yml
```

Deploy the webhook

In order to let our static `kube-apiserver` Pod access the webhook service with the DNS name `my-authn.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-authn | grep -v NAME | awk '{print $3}') my-authn.kube-system.
```

Add a DNS record

Now, we need to tell the `kube-apiserver` how to use our authentication webhook. This can be configured using the flag `--authentication-token-webhook-config-file` of the `kube-apiserver`.

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. Then, the `kubelet` will be notified of the change and restart it later. In the terminal above, follow the following steps to enable webhook authentication in the `kube-apiserver`.

```
sed -i "18i\    - --authentication-token-webhook-config-file=/etc/kubernetes/pki/webhook-config.yml" /et
docker ps -a | grep apiserver | grep -v "pause" | awk '{print $1}' | xargs docker rm -f
```

Enable webhook authentication in the kube-apiserver

Let's do some tests to see if the `kube-apiserver` can successfully use the token webhook authentication service at `https://my-authn.kube-system.svc/authorize`. We can use the token mentioned in Step 2 to construct a `kubeconfig` file using the commands below in the terminal above:

```
cp ~/.kube/config ./
kubectl config --kubeconfig=./config set-credentials webhook-test --token="ZGVtby11c2VyOmRlbW8tcGFzc3dvc
kubectl config --kubeconfig=./config set-context kubernetes-admin@kubernetes --user=webhook-test
```

Construct a kubeconfig file for testing

With this `kubeconfig` file, we can run the command below in the terminal above to see what happened:

```
kubectl get ns --kubeconfig=./config
```

List namespaces with token

The output should be similar to the following:

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

Output of the command

We can see that the token is identified correctly by our webhook. Additionally, this output just shows that the webhook is working as expected, because the webhook treats the token as the user `mock`, who apparently has no right (we didn't assign any RBAC roles for this mock user) to list namespaces.

Ta-da! The authentication webhook works!

Back

How Webhook Token Authentication Works

Next

How Webhook Authorization Works