How Webhook Token Authentication Works

Learn how webhook token authentication works in Kubernetes.

We'll cover the following

- Webhook token authentication
  - How to configure the plugin
- How it works
- Summary

Webhook token authentication

The task during the authentication stage is to identify if a request comes from a legitimate user and to reject all the other requests that don't.

Kubernetes bundles a group of authentication plugins as a union authentication chain, as shown in the code snippet below:

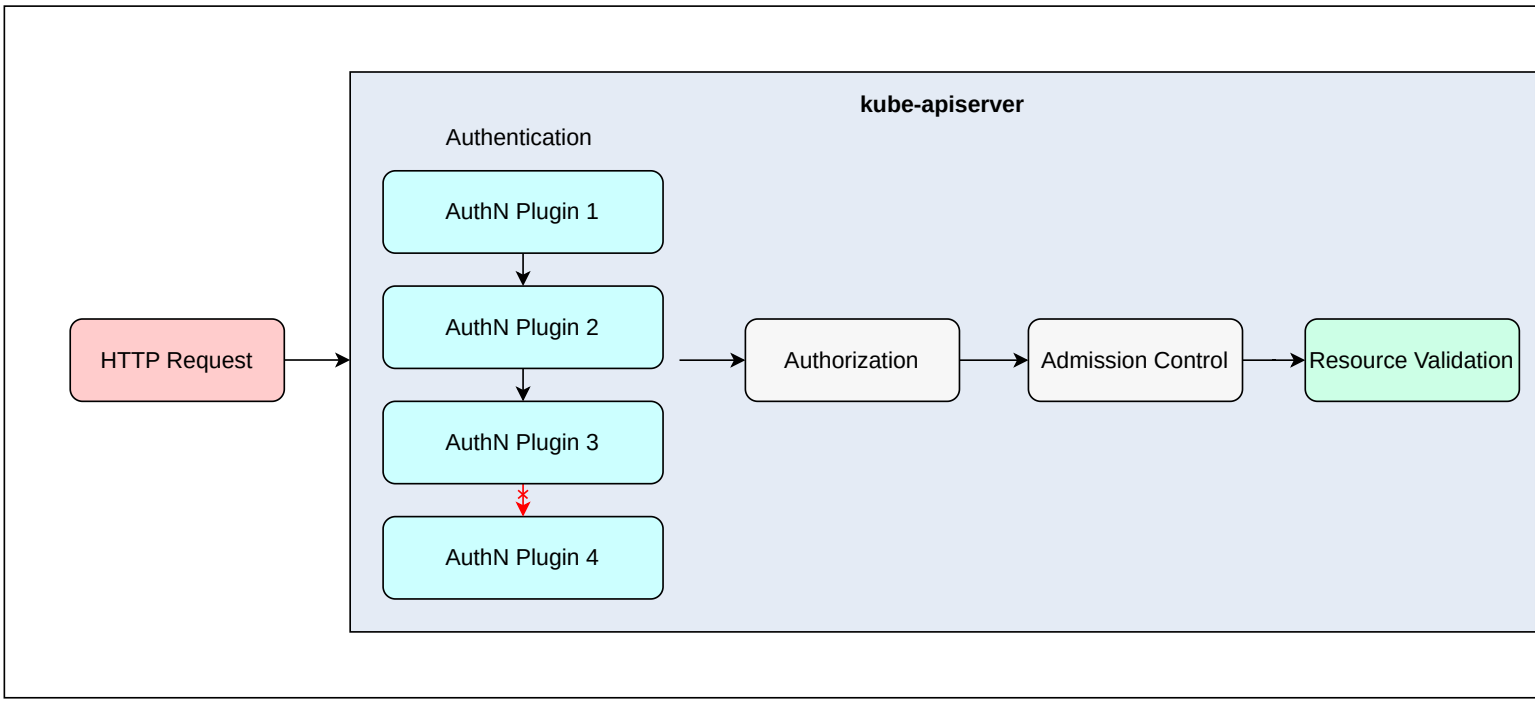
```
// Code from https://github.com/kubernetes/kubernetes/blob/master/staging/src/k8s.io/apiserver/pkg/authe
// New() returns a request authenticator that validates credentials using a chain of authenticator.Reque
// The entire chain is tried until one succeeds. If all fail, an aggregate error is returned.
func New(authRequestHandlers ...authenticator.Request) authenticator.Request {
    if len(authRequestHandlers) == 1 {
        return authRequestHandlers[0]
    }
    return &unionAuthRequestHandler{Handlers: authRequestHandlers, FailOnError: false}
}

// NewFailOnError() returns a request authenticator that validates credentials using a chain of authent
// The first error short-circuits the chain.
func NewFailOnError(authRequestHandlers ...authenticator.Request) authenticator.Request {
    if len(authRequestHandlers) == 1 {
        return authRequestHandlers[0]
    }
    return &unionAuthRequestHandler{Handlers: authRequestHandlers, FailOnError: true}
}

// AuthenticateRequest authenticates the request using a chain of authenticator.Request objects.
func (authHandler *unionAuthRequestHandler) AuthenticateRequest(req *http.Request) (*authenticator.Respo
    var errlist []error
    for _, currAuthRequestHandler := range authHandler.Handlers {
        resp, ok, err := currAuthRequestHandler.AuthenticateRequest(req)
        if err != nil {
            if authHandler.FailOnError {
                return resp, ok, err
            }
            errlist = append(errlist, err)
            continue
```

Union authenticator in the kube-apiserver

Each plugin implements a specific authentication method. The incoming requests will be presented to each plugin one by one, until one of them can successfully verify the user identity. Then, the authentication stage finishes and the request proceeds to the subsequent authorization stage. If none of the authentication plugins can verify the user identity, the coming request is rejected with a `401 Unauthorized` HTTP status code.



The kube-apiserver authentication flow

How to configure the plugin

We need to first configure the three flags below to trigger using the webhook token authentication plugin.

- `--authentication-token-webhook-config-file`: This is the file that tells the `kube-apiserver` the place to verify authentication tokens. This file is in the same format as `kubeconfig`, just as the sample file below shows. Fields that contain the comment `[CHANGEME]` need to be updated to our environment.

```
apiVersion: v1
kind: Config
clusters:
  - name: remote-authn-service
  cluster:
    certificate-authority: /path/to/ca.pem        # [CHANGEME] CA for verifying the remote service.
    server: https://auth.webhook.com/authenticate # [CHANGEME] URL of remote service to query. 'https
users:
  - name: webhook-authn
  user:
    client-certificate: /path/to/cert.pem # [CHANGEME] cert for the webhook plugin to use
    client-key: /path/to/key.pem          # [CHANGEME] key matching the cert
current-context: webhook-authn
contexts:
- context:
```

```
    cluster: remote-authn-service
    user: webhook-authn
    name: webhook-authn
```

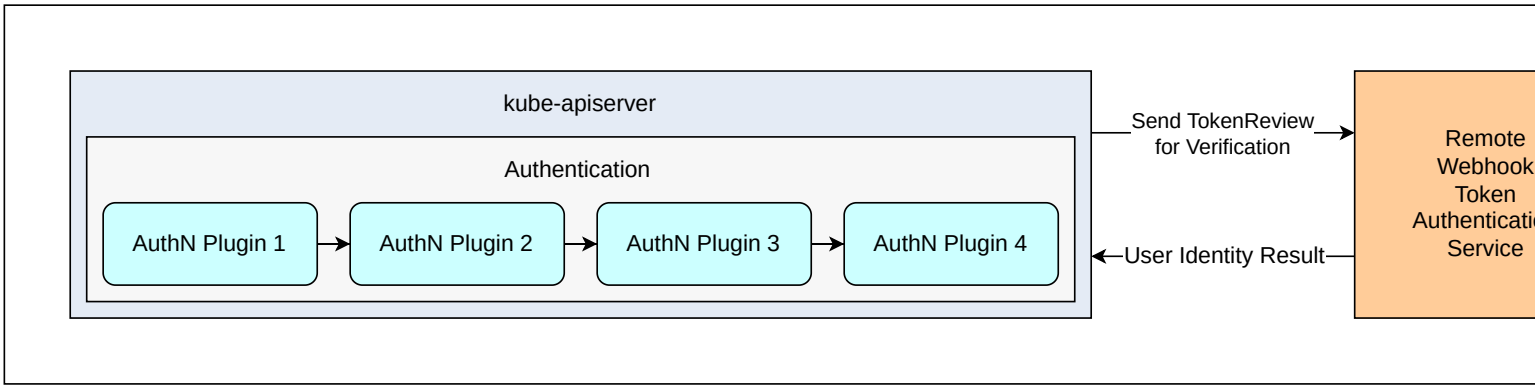
Authentication token webhook config file

- `--authentication-token-webhook-cache-ttl`: This flag specifies how long it takes to cache authentication decisions. It's set to `2m0s` by default. This helps improve the `kube-apiserver` performance by caching authentication results because every request needs passing authentication. In actual production environments, we can leave this with the default setting unless there's a need for adjustment.
- `--authentication-token-webhook-version`: This flag determines the API version of the `TokenReview` objects in the group `authentication.k8s.io` to send to and expect from the remote webhook authentication service. Currently, in Kubernetes v1.23, it remains `v1beta1` by default.

How it works

Now, let's see how this webhook token authentication plugin works.

The authentication webhook is an HTTP callback that will be called in the authentication stage.



How webhook token authentication works

During the authentication stage, if the preceding plugins in the union authentication chain fail to identify the user, the webhook plugin will be called to determine the legitimacy of the user. This webhook plugin extracts the HTTP bearer token from the original request. Then, it invokes an HTTP POST request to the remote authentication service specified in the flag `--authentication-token-webhook-config-file`.

```
{
  "apiVersion": "authentication.k8s.io/v1beta1",
  "kind": "TokenReview",
  "spec": {
    "token": "beaer-token-here-please-change-me"
  }
}
```

The TokenReview request payload

Here, the remote webhook token authentication service is entirely independent of Kubernetes. It can run anywhere as long as the `kube-apiserver` can access it. The implementation is entirely up to us, and we can use any language we're familiar with. All the communications are through HTTP.

The response from the remote authentication service should also be a valid `TokenReview` object like the one below. The `status` field contains the authenticated result and detailed user info, as shown below:

```
{
  "apiVersion": "authentication.k8s.io/v1beta1",
  "kind": "TokenReview",
  "spec": {
    "token": "beaer-token-here-please-change-me"
  },
  "status": {
    "authenticated": true,
    "user": {
      "username": "some-body",
      "uid": "a-valid-user-id",
      "group": [
        "group1"
      ]
    }
  }
}
```

The TokenReview response

Summary

With the authentication webhook, we can integrate Kubernetes with our internal user-management system.

Back

Introduction to AuthX

Next

Implementing a Webhook for Token Authentication