Using JSON web tokens to secure Elasticsearch

Cliff Staley · 2018-03-19

Managing user credentials and access to data and services moves more and more to centralized identity providers (IdP) and Single-Sign-On (SSO) solutions. Over are the days where each service maintains its own list of users and permissions. SSO technologies like Kerberos and SAML have been around for quite a while, but the new kid on the block is JWT and OpenID. Easy, lightweight, yet secure. This article series explains what JWT is all about, and how you can use it to secure your Elasticsearch cluster.

Token-based access control

The main players in any token-based access control scenario are the user, the service the user wants to access, and the identity provider (IdP). To access the service, the user first needs to authenticate against the IdP. The IdP verifies the user credentials and hands out a signed token. The user then sends this token to the service with each request. The service uses the information in the token to verify the user's identity and to assign roles and permissions.

> Token-based authentication works by allowing users to provide their credentials to an identity provider in order to obtain a secure token. A service can validate this token and grant access to resources based on the information contained in it.

How does it work in detail?

- The user tries to access a service
- The initial request does not contain any token or other credentials, so the service redirects the user to the IdP
- The user authenticates against the IdP by providing credentials like username and password
- The IdP creates a token that contains information about the user and signs it

- The IdP redirects the user to the service and adds the signed token
- The service checks if the signature of the token is valid.
- If validation succeeds, the service trusts all the information the token contains.

The main benefits of using token-based access control are:

- **Tokens are stateless**. All information a service needs to verify a users identity is already contained in the token. This improves scalability since the service does not need to store session state, and does not need to communicate with any other service for authentication.
- **Tokens provide for SSO.** Once a token has been generated, it can be used to access many different resources and services. This means the user has to provide credentials only once.
- **Token generation can be centralized.** You only need to maintain a single identity provider to grant access to your services and resources. Likewise, user information is only stored in a single system, making user and security management more straightforward.

JSON web tokens

JSON web token (JWT) is a JSON based standard for creating identity tokens (sometimes also called access tokens) that contain many verified statements about a user. It is specified in RFC7519.

Structure of a token

Header

The **Header** field defines the type of the token and the cryptographic algorithm used to sign it:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

In this example we use JWT tokens that are signed with the HMAC-SHA256 symmetric hash function.

Payload

The **Payload** field contains all data necessary for authenticating a user and deciding which operations are allowed or not. The payload fields are called *JWT claims*. JWT provides some predefined claims like:

- iss - issuer of the token
- sub - subject to whom a token is issued to
- exp - expiration time of a token
- iat - time when the token was issued

and many more. It also allows custom claims that can be used for specifying roles and other information about the user. An example payload may be:

```
{
 "iss": "search-guard.com",
 "exp": 1300819380,
 "name": "John Doe",
 "roles": "admin, devops",
 "department": "IT"
}
```

Most identity provides, like Keycloak for example, let you configure what custom claims should be contained in the JWT. Besides using these claims to assign roles and permissions to a user, you can also use them in Document-level security queries and for defining index-level access control.

Signature

The **Signature** field guarantees token integrity and is used by the service to verify the token is valid. It contains a cryptographic hash of the header and the payload. All parts of the JWT are base64 encoded, and the final token is calculated like:

```
encoded = base64UrlEncode(header) + "." + base64UrlEncode(payload)
signature = HMACSHA256(encoded, 'secretkey')
jwt = encoded + "." + base64UrlEncode(signature)
```

We first base64 encode the header and the payload, and then apply the cryptographic hash function. The value of JWT contains the whole token with header, payload and signature, separated by a dot.

Validating the token

The crucial part in JWT is token validation. The service that receives a token needs to be able to validate that it has been created and signed by a trusted identity provider. There are two basic approaches: Symmetric hash functions and public/private key based hash functions.

In the example above we have used the symmetric HMAC-SHA256 function. Once a service receives a signed token it checks if the payload is valid by running the same hash function that the IdP used. If the hashes are equal, the token is deemed valid. This, however, requires that both the IdP and the service know the key (or shared secret) that was used for creating the hash. In our example above, the shared secret is "secretkey".

Clearly a public/private key based hash provides for much better security because it does not require any shared secret: The IdP signs the token with its private key, and the service only needs to know the public key to validate the token. Search Guard supports both symmetric and PKI based hashes, and you can use multiple different algorithms for the hash function (See section “Supported formats and algorithms” in our documentation). For security reasons, we recommend the public/private key approach.

JWT in action @ Search Guard

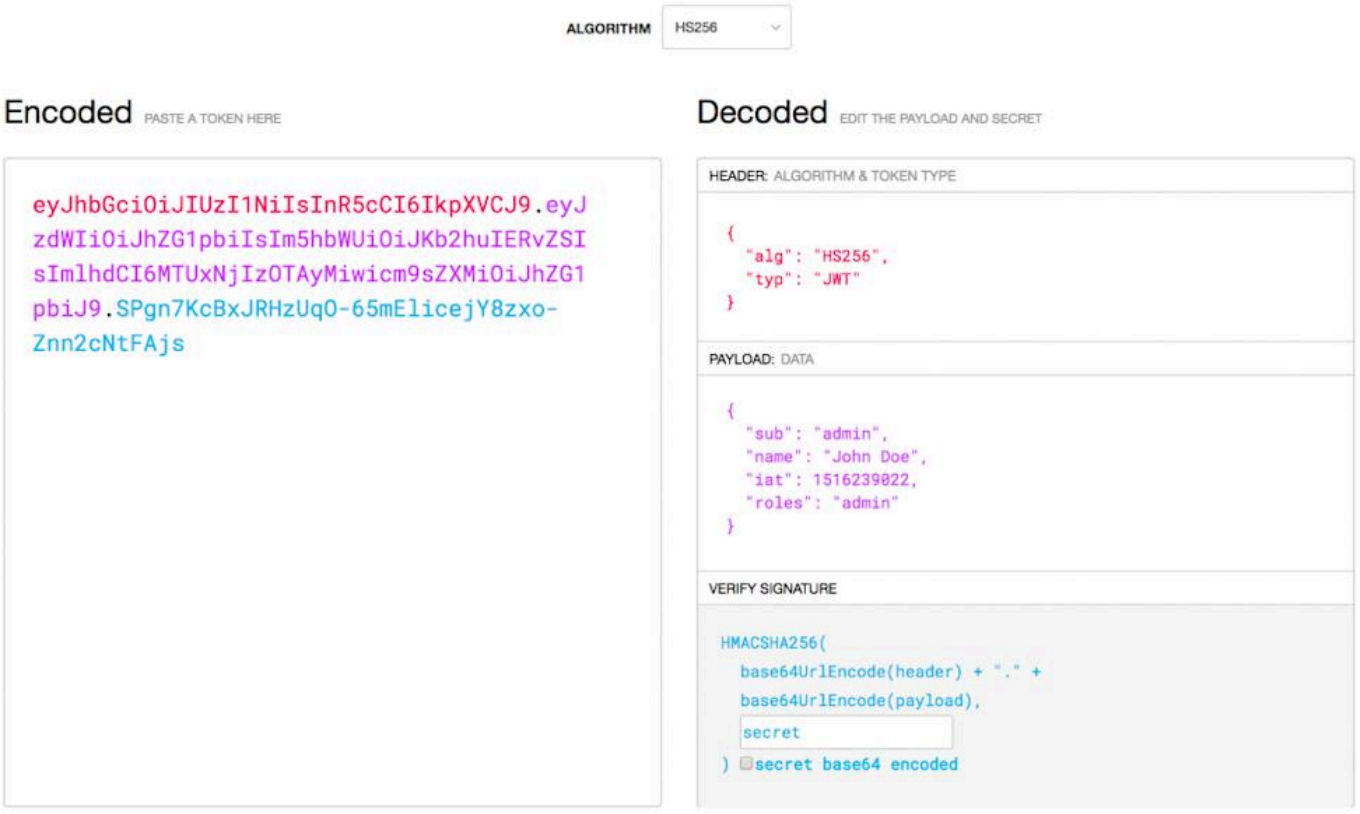
In this section we provide a step-by-step guide for using JWT with Search Guard. We assume you are already have an instance of Elasticsearch and Search Guard running,

Symmetric hash functions

To test JWT with a symmetric hash / shared secret, let's create a JWT first. We will use the jwt.io website which has some great tools to generate and debug JWTs. We first make sure the algorithm is set to HS256. After that, in the "verify signature" section, we set the shared secret to "secret". Finally, we add some claims to the JWT that we will use for assigning roles to the user:

```
{
 "sub": "admin",
 "name": "John Doe",
 "iat": 1516239022,
 "roles": "admin"
}
```

We will use the *sub* claim to store the username, and added the *roles* claim to store the roles. You can have more than one role in this field by separating the roles by a comma. The screen should now look like:



Next, we need to base64 encode our secret key. You can use a command line tool like base64 on Linux and Mac systems, or just use an online tool like base64encode.org. The base64 encoded secret is *c2VjcmV0*. Now we have everything ready for configuring Search Guard.

We need to enable JWT in *sg_config.yml*, provide the base64 encoded secret key and tell Search Guard in which claims the username and the roles are stored. Based on our token and the base64 encoded secret, the corresponding entry in *sg_config* looks like:

```
jwt_auth_domain:
 http_enabled: true
 transport_enabled: false
 order: 0
 http_authenticator:
   type: jwt
   challenge: false
   config:
     signing_key: "c2VjcmV0"
     jwt_header: "Authorization"
     roles_key: "roles"
     subject_key: "sub"
 authentication_backend:
   type: noop
```

We provided the signing key, roles key, and subject key. The additional *jwt_header* configuration tells Search Guard where to find the JSON web token in the HTTP headers. *Authorization* is the default, so we could have skipped this part.

The *authentication_backend* is set to *noop* - this is because the JSON web tokens are self-contained, and once their signature is validated, we already have everything we need to assign roles and permissions!

After enabling the configuration by running sgadmin, we can use the token generated by jwt.io to access Elasticsearch. Let's try the *authinfo* endpoint which will print out information about the user:

```
curl -i https://<ES-HOST>:9200/_searchguard/authinfo -H "Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJhZG1pbiIsIm5hbWUiOiJKb2huIERvZSIsImlhdCI6MTUxNjIzO
TAyMiwicm9sZXMiOiJhZG1pbiJ9.SPgn7KcBxJRHzUqO-65mElicejY8zxo-Znn2cNtFAjs"
```

Note the "Bearer" before the actual token: If you transmit a JWT in an HTTP request it has to be set as Bearer token.

PKI based JWT

PKI configuration can be set up almost the same way. The only difference is that we need to generate a private/public keypair, use the private key to sign the token, and put the value of the public key in *sg_config.yml*.

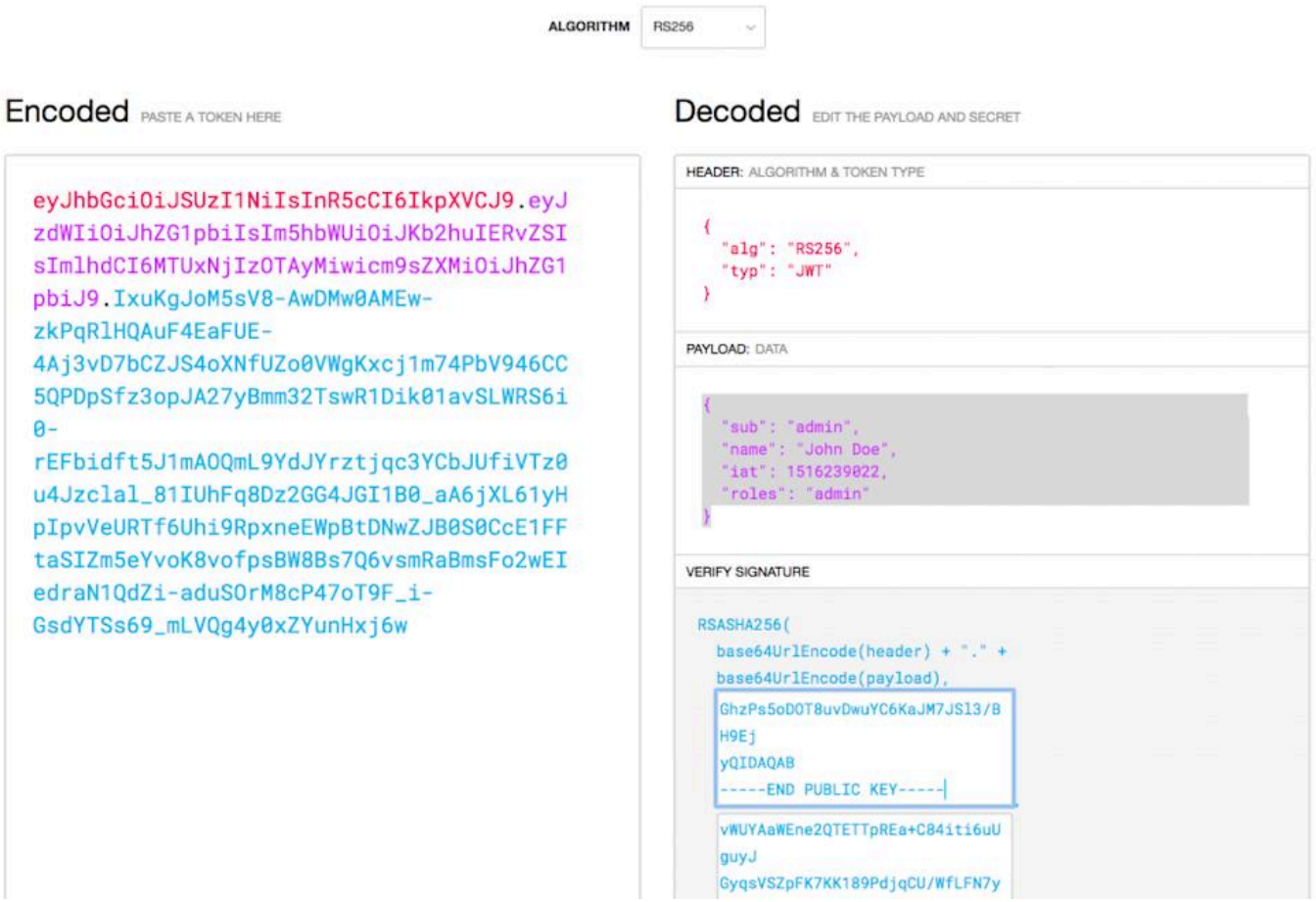
For testing purpose, private and public keys can be generated with commands:

```
openssl genrsa -out privatekey.pem 2048
```

Generates keypair and saves it in keypair.pem file.

```
openssl rsa -in privatekey.pem -pubout -out publickey.crt
```

This saves public key in the file *publickey.crt*. Going back to jwt.io, we change the algorithm to RS256 and paste the public and the private key into the respective text fields. Your screen should look something like this:



We simply put the public key in *sg_config.yml*, as a multiline value:

```
signing_key: |-
  -----BEGIN PUBLIC KEY-----
  MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA2GppeaZpIEtMIEu736gh
  rLn8Ti47Gk5uJm5UTHkQOtieKUC/EzPZI0HePQ7vfbMYPAO8ybP4GV5iR3kXvBVb
  ...
  -----END PUBLIC KEY-----
```

That's it. Search Guard will automatically detect that we now use PKI based hashed and also what hash algorithm was used to sign the token!

Resources

JWT.io website contains several useful information on JSON web tokens. It also links to several client libraries that help you to generate the tokens in various languages. Additionally, it has an online debugger which generates tokens, based on data you provide. This allows you to check if the tokens you have generated are valid.

JWT and OpenID

In this article, we have explained how JSON Web Tokens work and provided a step-by-step guide to test JWT configuration with Search Guard. In the next articles in this series we will look into OpenID and

explain how you can use IdPs like Keycloak or Google to fully automate the configuration and authentication process.

References:

1. https://scotch.io/tutorials/the-ins-and-outs-of-token-based-authentication

Image: shutterstock / ImageGentle

This article was published on the Search Guard blog: https://search-guard.com/blog/jwt-secure-elasticsearch/