Entendendo o padrão de segurança JWT

E seu uso em Python

Cleuton Sampaio, M.Sc.

Descrição

Toda aplicação web que retorna recursos dinâmicos deveria se proteger contra acessos anônimos. Há dois motivos para isto:

- Evitar roubo de informação (web scrapping);
- Evitar divulgação de informação corporativa;

Podemos proteger uma aplicação simplesmente adicionando um **CAPTCHA** (como o Google reCaptcha: https://www.google.com/recaptcha/about/)? Sim. Isso protege contra o roubo automático por scripts, mas não contra o acesso indevido manual, com uma pessoa navegando.

É preciso acabar com o acesso anônimo, protegendo rotas web que retornem recursos dinâmicos. Isso não só protege seus dados, como evita sobrecarregar seus servidores (de aplicação, de banco de dados etc) com acessos ilegítimos, o que é um tipo de ataque DOS (Denial of Service).

Existem vários esquemas de autenticação de usuário. Antigamente, quando utilizávamos o conceito de SESSÃO WEB, o cliente recebia um "cookie" identificador de sessão e tinha que sempre acessar o mesmo servidor.

Hoje, com o conceito de recursos **REST**, que são ***Stateless*** por natureza, este esquema não funciona mais, embora seja possível guardar o estado em um banco de dados no ***backend***.

Cookies não são uma forma muito segura de guardar identificações de estado, portanto, um novo esquema de autenticação é necessário.

JASON Web Token

Sempre que criamos um esquema particular de autenticação, ficamos sujeitos a incompatibilidades. O JWT (JASON Web Token) é um mecanismo para transitar informações entre ***frontend*** e ***backend*** de maneira padronizada.

Um token JWT é um conjunto de afirmações (“claims”) sobre um usuário remoto, por exemplo:

```
{
  "fresh": false,
  "iat": 1639743783,
  "jti": "72ad917e-e4f0-4efc-b797-9656e92b215c",
  "type": "access",
  "sub": "test",
  "nbf": 1639743783,
  "exp": 1639744683
}
```

Neste token temos alguns ***claims*** interessantes:

- sub: Subject ou o usuário que está enviando o token;
- exp: Expiration ou intervalo de tempo em que o token deixará de ser válido;

JWT e Sessões

Alguns utilizam tokens JWT para enviar estado de sessão mantido no ***frontend*** para o ***backend***. Para isso, utilizam ***claims*** particulares, fora do esquema padrão do JWT. Eu não considero uma boa prática, pois manter estado no ***frontend*** é inseguro e trafegar grandes quantidades de bytes por JWT é ruim.

Autenticação web

O padrão é utilizar tokens JWT para identificar um usuário autenticado. Você pode ter várias **rotas REST** em seu ***backend***, sendo que algumas exigem que o usuário esteja autenticado. Neste caso ele pode enviar um token JWT para provar isso.

Cabe ao ***backend*** validar o token JWT antes de retornar os recursos para o ***frontend***.

Há várias maneiras para o ***frontend*** enviar um token JWT para o ***backend***:

- Em um **cookie**: Tem a vantagem de ser automático e diminui o trabalho para o ***frontend***, mas há o risco de Cross Site Request Forgery (**CSRF**);
- Em um **header HTTP**: Uma das melhores opções, pois existe o header ***Authorization*** que é exatamente para isto;
- No corpo do **request**: Além de complicar o processamento do ***request***, também está sujeito a certos tipos de ataque;

O fluxo de autenticação seria algo assim:

1. **Frontend**: Tenta acessar uma rota protegida;
2. **Backend**: Retorna HTTP Status 401;
3. **Frontend**: Acessa uma rota de login, enviando credenciais;
4. **Backend**: Se tudo estiver correto, retorna um token JWT no corpo da mensagem;
5. **Frontend**: Acessa uma rota protegida, passando o header “Authorization: Bearer <token recebido>”;
6. **Backend**: Valida o token (vê se a assinatura está ok e se não expirou) e retorna o recurso protegido. Caso haja erro no token, pode retornar erro.

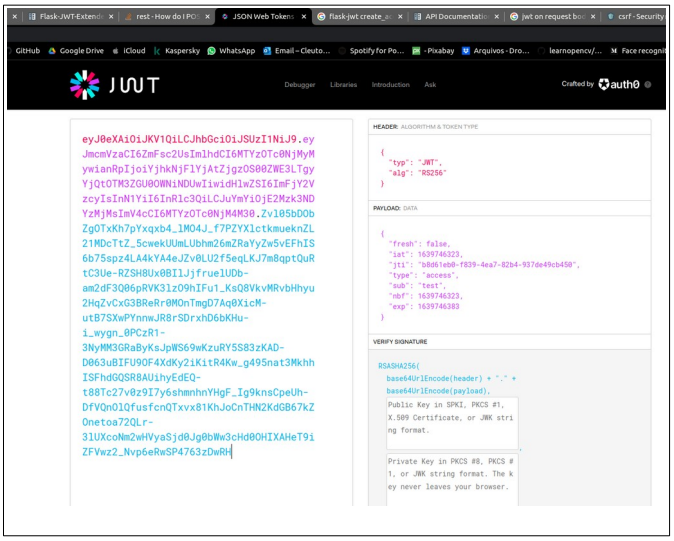
Há muitas variações, inclusive em esquemas de ***Single Sign-On*** e **autenticação federada**, mas este fluxo muda pouca coisa.

Componentes do token JWT

Um token JWT é algo assim:


```
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJmcmVzaCI6ZmFsc2UsImlhdCI6MTYzOTc0NjMyMywianRpIjoiYjhkNjFlYjAtZjgzOS00ZWE3LTgyYjQtOTM3ZGU0OWNiNDUwIiwidHlwZSI6ImFjY2VzcyIsInN1YiI6InRlc3QiLCJuYmYiOjE2Mzk3NDYzMjMsImV4cCI6MTYzOTc0NjM4M30.Zvl05bDObZgOTxKh7pYxqxb4_lMO4J_f7PZYXlctkmueknZL21MDcTtZ_5cwekUUmLUbhm26mZRaYyZw5vEFhIS6b75spz4LA4kYA4eJZv0LU2f5eqLKJ7m8qptQuRtC3Ue-RZSH8Ux0BIlJjfruelUDb-am2dF3Q06pRVK3lzO9hIFu1_KsQ8VkvMRvbHhyu2HqZvCxG3BReRr0MOnTmgD7Aq0XicM-utB7SXwPYnnwJR8rSDrxhD6bKHu-i_wygn_0PCzR1-3NyMM3GRaByKsJpWS69wKzuRY5S83zKAD-D063uBIFU9OF4XdKy2iKitR4Kw_g495nat3MkhhISFhdGQSR8AUihyEdEQ-t88Tc27v0z9I7y6shmnhnYHgF_Ig9knsCpeUh-DfVQnOlQfusfcnQTxvx81KhJoCnTHN2KdGB67kZOnetoa72QLr-3lUXcoNm2wHVyaSjd0Jg0bWw3cHd0OHIXAHeT9iZFVwz2_Nvp6eRwSP4763zDwRH
```


Há 3 partes separadas por pontos: Header, Payload e Assinatura digital. Se quiser conferir o token, pode ir no site http://jwt.io:



Podemos ver que este JWT contém um header:

```
{
  "typ": "JWT",
  "alg": "RS256"
}
```

E um payload:

```
{
  "fresh": false,
  "iat": 1639746323,
  "jti": "b8d61eb0-f839-4ea7-82b4-937de49cb450",
  "type": "access",
  "sub": "test",
  "nbf": 1639746323,
  "exp": 1639746383
}
```

A última parte é a assinatura digital.

O token não deve conter informações privilegiadas, como senhas ou níveis de autorização, ou mesmo dados de sessão. Se for necessário incluir isso, então é melhor encriptar o token. A própria proteção do HTTPS é suficiente para manter um token JWT protegido. Porém, é preciso verificar se a assinatura digital é válida e se o token não expirou.

Há duas maneiras de assinar um token JWT:

- Chave simétrica: Todas as partes conhecem a chave, que serve para encriptar e decriptar;
- Chave assimétrica: Há uma chave privada e uma chave pública;

A maneira mais segura é assinar um token utilizando a **chave** assimétrica **privada** no ***backend***, e validar com a **chave** assimétrica **pública**.

Refresh de token

O que acontece quando o token expira? Simples: Você precisa se autenticar novamente!

Vários frameworks permitem ***refresh*** de token. Muitos geram novo token a cada request, ou quando o token está próximo de expirar. Também é possível colocar um prazo muito alto para expiração ou mesmo gerar token sem prazo de expiração, o que é muito arriscado.

Um esquema que eu gosto é NÃO FAZER REFRESH! Ao fazer ***refresh***, você revalida o token, permitindo que o usuário fique indefinidamente acessando com o mesmo token.

Você pode gerar um token com duração inicial alta, digamos: 1 dia, e, para operações realmente sensíveis (comprar, deletar etc) pode exigir que o usuário entre com a senha novamente, gerando novo token.

Muitas aplicações usam este esquema. Depois que o usuário se autentica, deixam acessar as rotas menos sensíveis mas, para acessar perfil ou realizar compras, exigem a senha novamente, gerando um token mais recente.

Cancelamento de token (logout)

Há certas histórias de usuário que podem requerer sair da aplicação, processo conhecido como ***logout***, ***logoff*** ou ***sign-off.*** Isso não é comum em aplicações web, mas existe em aplicações móveis. Neste caso é preciso invalidar o token, e não há uma maneira simples de fazer isso. Se o token não expirou, então pode ser usado novamente.

O que podemos fazer é criar um repositório no ***backend*** contendo os tokens cancelados e testar se um token recebido está entre eles.

Ataques de força bruta

Um dos tipos de ataque mais conhecidos (e que funciona até hoje) é tentar logar com diversas senhas conhecidas. A Wikipedia tem uma lista de passwords comuns, utilizadas até hoje: https://en.wikipedia.org/wiki/List_of_the_most_common_passwords

Veja só:

- password
- 123456
- qwerty

E no Brasil também temos uma lista (https://www.tecmundo.com.br/seguranca/229020-lista-mostra-senhas-comuns-vazamentos-brasil.htm):

- 123456
- 123456789
- Brasil
- 12345
- 102030
- senha
- 12345678
- 1234
- 10203
- 123123
- 123
- 1234567
- 654321
- 1234567890
- gabriel
- abc123
- q1w2e3r4t5y6
- 101010
- 159753
- 123321
- senha123
- mirantte
- flamengo

Se você repetir o login com algumas dessas senhas, há grande chance de sucesso. Os ***hackers*** utilizam bases de dados com usuários e senhas capturados em fraudes, criando ***scripts*** que ficam

tentando logar na sua aplicação o tempo todo. Eles podem até configurar tempos de espera aleatórios entre um login e outro, ou mudar de **zumbi** (máquinas infectadas) para te enganar.

Há algumas maneiras de se proteger disso:

1. No primeiro erro, enviar um **CAPTCHA** de volta para o ***frontend***;
2. Sempre exigir **CAPTCHA** para ***login***;
3. Contar a quantidade de tentativas de ***login*** e suspender (temporariamente ou não) aquele usuário;

Isso é responsabilidade do ***backend*** e não está previsto no esquema JWT.

Em resumo

1. Tokens JWT servem para identificar um usuário autenticado;
2. Devem ser enviados no header Authorization;
3. Podem expirar, requerendo geração de novo token;
4. Não devem ser utilizados para trafegar nada além dos ***claims*** padrões;
5. Devem ser assinados digitalmente com chaves assimétricas;
6. Cancelamento de tokens devem ser lidados pelo ***backend*** com listas de bloqueio;
7. Cabe ao ***backend*** se precaver de ataques de força bruta;

Uso em python

O uso de JWT em python é feito através da biblioteca ***pyjwt*** (em conjunto com a ***cryptography***).

Eu tenho um repositório com demonstração de uso de JWT com Python:

Primeiramente, crie um ambiente virtual:

```
python –m venv .
```

Depois, ative e instale as dependências:

```
source bin/activate
pip install –r requirements.txt
```

Há três exemplos de uso:

- Assinatura com chave simétrica: **first.py**;
- Assinatura com chave assimétrica: **asymetric.py**;
- Aplicação REST: **restsample.py**;

Uso básico

O script first.py demonstra o uso básico de token, gerando e verificando. Podemos gerar um token com a função **encode**:

```
payload = {
    "sub": "cleuton"
}

senha_simetrica = 'senha-secreta!!'
token = jwt.encode(
    payload=payload,
    key=senha_simetrica
)
```

Aqui utilizamos uma senha simétrica e um payload contendo apenas “sub”.

Podemos validar um token com a função **decode**:

```
print(jwt.decode(token, key=senha_simetrica,
algorithms=['HS256', ]))
```

O algoritmo “HS256” é de chave simétrica. Se tudo der certo, o resultado é o ***payload***:

```
{'sub': 'cleuton'}
```

E podemos usar os campos do ***payload*** em nosso programa.

Se o token estiver inválido (assinatura não confere) uma exception será retornada:

```
fake_token=token.replace('XA','X0')

print(fake_token)

try:
    print(jwt.decode(fake_token, key=senha_simetrica, algorithms=['HS256', ]))
except InvalidSignatureError as e:
    print(e)
```

Uso com assinatura de chave assimétrica

Esta é a maneira mais segura de assinar e validar um token. Podemos gerar chave assimétrica com o OpenSSH:

```
ssh-keygen -t rsa
```

Você pode gerar as chaves em qualquer pasta. Uma chave privada será criada e uma chave pública (extensão “.pub) também será criada.

Utilizando o pacote ***cryptography*** podemos ler e armazenar o texto das chaves:

```
private_key = open('.ssh/cleuton', 'r').read()

key = serialization.load_ssh_private_key(private_key.encode(),
password=b'teste')

…

public_key = open('.ssh/cleuton.pub', 'r').read()

pubKey = serialization.load_ssh_public_key(public_key.encode())
```

Se você usou uma ***passphrase*** para proteger sua chave privada, então tem que informá-la quando for carregar (argumento ***password***).

A única diferença no ***encode*** do token é a **chave privada** para assinar:

```
new_token = jwt.encode(
    payload=payload_data,
    key=key,
    algorithm='RS256'
)
```

A variável ***key*** é a chave privada que lemos e o algoritmo deve ser o “RS256”.

A verificação também é semelhante:

```
print(jwt.decode(jwt=new_token, key=public_key, algorithms=['RS256', ]))
```

Só que utilizamos a **chave pública** para validar a assinatura. Se der erro, a mesma exception será levantada.

Uso com serviços REST

O Flask tem o pacote **flask_jwt_extended** com uma série de utilitários para usar JWT em serviços REST.

Precisamos configurar nossa app Flask para trabalhar com JWT:

```
app.config["JWT_PRIVATE_KEY"] = prKey
app.config["JWT_PUBLIC_KEY"] = pubKey
app.config['JWT_ALGORITHM'] = 'RS256'
app.config['JWT_ACCESS_TOKEN_EXPIRES'] = datetime.timedelta(minutes=1)
```

Note que já carretamos nossas chaves privada e pública, também selecionamos o algoritmo e o valor padrão para a ***claim*** “exp”, que será em 1 minuto (só para exemplo).

São poucas diferenças para uma aplicação REST Flask comum. A primeira é a necessidade de adicionar o JWTManager:

```
jwt = JWTManager(app)
```

As rotas REST que queremos proteger temos que anotar com o decorator @jwt_required:

```
@app.route("/protected", methods=["GET"])
@jwt_required()
def protected():
```

Se o frontend tentar acessar uma dessas rotas, sem ter o ***header Authorization***, vai tomar um HTTP Status 401.

```
$ curl -i http://localhost:5000/protected
HTTP/1.0 401 UNAUTHORIZED
Content-Type: application/json
Content-Length: 39
Server: Werkzeug/2.0.2 Python/3.8.8
Date: Fri, 17 Dec 2021 14:41:35 GMT

{"msg":"Missing Authorization Header"}
```

Se ele usar a rota de ***login***, informando usuário e senha corretos, vai receber um token JWT:

```
$ curl -i --header "Content-Type: application/json" \
>   --request POST \
>   --data '{"username":"test","password":"test"}' \
>   http://localhost:5000/login
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 757
Server: Werkzeug/2.0.2 Python/3.8.8
Date: Fri, 17 Dec 2021 14:43:43 GMT

{"access_token":"eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJmcmVzaCI6ZmFsc2UsImlhdCI6MTYzOTc1MjIyMywianRpIjoiODQ4YWFmZTMtYzIwMS00NTU0LTg4ZjktMzJiYTRiZTRkMmNhIiwidHlwZSI6ImFjY2VzcyIsInN1YiI6InRlc3QiLCJuYmYiOjE2Mzk3NTIyMjMsImV4cCI6MTYzOTc1MjI4M30.ZBNcAPW84KuRmXxIzMTWeypjJn6jSPHatlqSYiP0_UY1E22KI1bEmFMhq5OjXKvlibcJh_p2a4sReJ6Yi8dEdIYD9G4Nn6-3tu-dHkwdStt9EMJwGB8GPAtMc31OGDJVT6MuOgtpnXqzqyODiDLP1FEPfcCeyqPHPf35vs44Ks1rznfodUd14r6KMi3N342_eomsv0itq1u30kaEH1rNWIMqPrXSka8VdA1Xv_UP0PTPirU89o4hpCQyPDRNI1rtFX6ReA1wzTqKMFrFq2Cmkrw3CSePptJ2SM5eNGLaZvv37tpk9dL9YgDfYjxOZWa2U2D1EpdccX6h18cdBIObByJgTds5Z_qNX-bkhnX2mH2YCpF6odV4Ru7e7JWkIlZWD-3iv8q_TRp4q3f19RSAUPlIOpZQn17gAomYHUFIVmdZYupNnr3Gbk0YzHj_6GdNdNUL3nQJAUA_c4-mtXak1Zrtjf58P_9Qe0laPAcXe6JodLFteKpxtqY0lPSgVb5"}
```

Então ele pode acessar a rota protegida:

```
$ curl --header "Content-Type: application/json" \
> --header "Authorization: Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJmcmVzaCI6ZmFsc2UsImlhdCI6MTYzOTc1MjM4MCwianRpIjoiOTExNTIyZTQtOGFmYy00ZWY1LTgxYjktNzBlN2Y5YTUyYzkyIiwidHlwZSI6ImFjY2VzcyIsInN1YiI6InRlc3QiLCJuYmYiOjE2Mzk3NTIzODAsImV4cCI6MTYzOTc1Mjk4MH0.WB9z23HENzk-14ndAaj5I7E7Sc7FUzPmZ8Vk6z-l3S6pey0qh69O8hjJG6tQc2myFdMy8fddKqrDEVYsfhujactbUqsAjBT6vDDMzStWJMCZzTcoZD4kRHHaYSaAjr4oUbj5B9CSJCnqfqHjNp4Ym-gYdrIYBRBy5Fdw4Wm8eEip_X83dQ-XP8T50DQiBteorBMT6Hi20Ktpvefxzlr7NwrK_uP27LKkL_b5nT__3VXdfcUnNYonjU_YV-QUZSRL7SHBiZMIvHmMHQ46-8NbMBguFCekqNZjdIDLD4YWXi7dgR-fN52mVsBgEFhFi6MNm5ENPx2V71_109v837OYSl5MufYWkCNP5X1wwW37uXrkkGYiSI2A0CJ689ZaMruYYSUSNt71A-OK1uJ7sbwdEiReM2HOR1oePiLc5Q3xWD9UwBrIfbYDsoX3SbvggtxizXRiEWpxi3fXG9Gk4u7oy6BQC7I9mwKoZdVLumLGFEfm0wgfjC-YXLRP0jZsJNKt" \
```

```
> http://localhost:5000/protected

{"logado":"test"}
```

Note que este esquema apenas diz se o usuário está autenticado. Se a rota exigir um **nível de autorização diferente,** então caberá ao ***backend*** decidir isso com base no “**sub**” do usuário! Eu não incluiria essa informação no token JWT, como alguns fazem (“nivel”: “admin”).

Se tentarmos acessar depois que o token expirar, receberemos um erro:

```
$ curl -i --header "Content-Type: application/json" \
> --header "Authorization: Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJmcmVzaCI6ZmFsc2UsImlhdCI6MTYzOTc1MjU
...
u2o2xcUYzAFL6FDQKcbN-_C36uXG4rcbKOlVINfD2ZhEk654" \
> http://localhost:5000/protected
HTTP/1.0 401 UNAUTHORIZED
Content-Type: application/json
Content-Length: 28
Server: Werkzeug/2.0.2 Python/3.8.8
Date: Fri, 17 Dec 2021 14:51:23 GMT

{"msg":"Token has expired"}
```