JWT vs cookies for token-based authentication

Asked 4 years, 9 months ago Active 1 month ago Viewed 56k times

137

80

I read some posts about **"JWT vs Cookie"** but they only made me more confused...

1. I want some **clarification**, when people talking about "token-based authentication vs cookies", **cookies** here merely refer to ***session cookies***? My understanding is that **cookie is like a medium**, it can be used to implement a token-based authentication(store something that can identify logged-in user on the **client side**) or a session-based authentication(store a constant on the client side that matches session information on the **server side**)
2. Why do we need **JSON web token**? I was using the standard cookie to implement token-based authentication(**not using session id, not use server memory or file storage**): `Set-Cookie: user=innocent; preferred-color=azure`, and the only difference that I observed is that JWT contains both **payload and signature**...whereas you can choose between **signed or plaintext** cookie for http header. In my opinion signed cookie (`cookie:'time=s%3A1464743488946.WvSJxbCspOG3aiGi4zCMMR9yBdvS%2B6Ob2f3OG6%2FYCJM'`) is more space efficient, the only drawback is that client cannot read the token, only the server can...but I think it's fine because just like *claim* in JWT is optional, it's not necessary for token to be meaningful

json authentication cookies jwt

Share Improve this question Follow

edited Sep 15 '18 at 13:10
Bergi
502k ● 104 ● 804 ● 1147

asked Jun 2 '16 at 4:02
watashiSHUN
6,976 ● 3 ● 30 ● 38

6 Answers

Active Oldest Votes

187

+50

The biggest difference between bearer tokens and cookies is that the browser will ***automatically send cookies***, where bearer tokens need to be added explicitly to the HTTP request.

This feature makes cookies a good way to secure websites, where a user logs in and navigates between pages using links.

The browser automatically sending cookies also has a big downside, which is [CSRF](https://en.wikipedia.org/wiki/Cross-site_request_forgery) attacks. In a CSRF attack, a malicious website takes advantage of the fact that your browser will automatically attach authentication cookies to requests to that domain and tricks your browser into executing a request.

Suppose the web site at https://www.example.com allows authenticated users to change their passwords by `POST`-ing the new password to https://www.example.com/changepassword without requiring the username or old password to be posted.

If you are still logged in to that website when you visit a malicious website which loads a page in your browser that triggers a POST to that address, your browser will faithfully attach the authentication cookies, allowing the attacker to change your password.

authentication cookies are set, as the same-origin policy won't send cookies to another domain.

Also, cookies make it more difficult for non-browser based applications (like mobile to tablet apps) to consume your API.

Share Improve this answer Follow

answered Jun 4 '16 at 23:23
MvdD
17.4k ● 5 ● 50 ● 81

8 "If you are still logged in to that website when you visit a malicious website which loads a page in your browser that triggers a POST to that address, your browser will faithfully attach the authentication cookies, allowing the attacker to change your password." Doesn't CORS prevent this? – kbuilds Mar 18 '17 at 23:34

21 @kbuilds Only is the malicious page is using AJAX to POST the form. If the attacker gets you to click the submit button on a regular form, CORS does not come into play. – MvdD Mar 18 '17 at 23:52

3 but doesn't this mean that the site would only be vulnerable if there were no CSRF tokens being used? – kbuilds Mar 19 '17 at 0:04

5 Right, you can mitigate CSRF attacks by using CSRF tokens. But this is something you have to do explicitly. – MvdD Mar 19 '17 at 0:27

3 It's worthwhile to mention that the Set-Cookie's SameSite attribute can effectively prevent CSRF attacks. developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie/… And since browsers now use `SameSite=Lax` as a default value and if you do not make changes on GET requests, you are protected by default. However only in case of modern browsers. `SameSite=Strict` is even a more robust variant. A good article to read: netsparker.com/blog/web-security/… – RussCoder Jun 7 '20 at 13:51

120

Overview

What you're asking for is the difference between cookies and bearer tokens for sending JSON Web Tokens (JWTs) from the client to the server.

Both cookies and bearer tokens send data.

One difference is that cookies are for sending and storing arbitrary data, whereas bearer tokens are specifically for sending authorization data.

That data is often encoded as a JWT.

Cookie

A cookie is a name-value pair, that is stored in a web browser, and that has an expiry date and associated domain.

We store cookies in a web browser either with JavaScript or with an HTTP Response header.

```
document.cookie = 'my_cookie_name=my_cookie_value'   // JavaScript
Set-Cookie: my_cookie_name=my_cookie_value           // HTTP Response Header
```

The web browser automatically sends cookies with every request to the cookie's domain.

```
GET http://www.bigfont.ca
```

Bearer Token

A bearer token is a value that goes into the `Authorization` header of any HTTP Request. It is not automatically stored anywhere, it has no expiry date, and no associated domain. It's just a value. We manually store that value in our clients and manually add that value to the HTTP Authorization header.

```
GET http://www.bigfont.ca
Authorization: Bearer my_bearer_token_value          // HTTP Request Header
```

JWT and Token Based Authentication

When we do token-based authentication, such as OpenID, OAuth, or OpenID Connect, we receive an access_token (and sometimes id_token) from a trusted authority. Usually we want to store it and send it along with HTTP Requests for protected resources. How do we do that?

Option 1 is to store the token(s) in a cookie. This handles storage and also automatically sends the token(s) to the server in the `Cookie` header of each request. The server then parses the cookie, checks the token(s), and responds accordingly.

Option 2 is to store the token in local/session storage, and then manually set the `Authorization` header of each request. In this case, the server reads the header and proceeds just like with a cookie.

It's worth reading the linked RFCs to learn more.

Share Improve this answer Follow   edited Nov 12 '20 at 21:00 Community ♦ 1 ● 1   answered Jul 20 '16 at 0:45 Shaun Luttin 105k ● 63 ● 323 ● 404

is it safe to store sensitive data into local/session storage? or it's just whatever since tokens are ephemeral? – hyunchel Mar 6 at 2:01

27

In addition to what MvdD has said about cookies being automatically sent:

1. A cookie can be a medium, but its most significant function is how it interacts with the browser. Cookies are set by the server and sent in requests in very specific ways. JWT on the other hand is exclusively a medium, it is an assertion of some facts in a particular structure. If you were so inclined, you could put a JWT as your authentication cookie. When you read articles comparing them, they typically are talking about using a JWT sent as a bearer token by front end code vs an authentication cookie which corresponds to some cached session or user data on the back end.
2. JWT offers many features, and puts them in a standard so they can be used between parties. A JWT can act as a signed assertion of some facts in many different places. A cookie, no matter what data you put in it or if you sign it, only really makes sense to use between a browser and a specific back end. JWT can be used from browser to back end, between back ends controlled by different parties (OpenId Connect is an example), or within back end services of one party. Regarding your specific example of your signed cookies, you can probably achieve the same functions ("not using session id, not use server memory or file storage") as JWT in that use case, but you lose out on libraries and peer-review of the standard, in addition to the CSRF issues talked about in the other answer.

standardization and features for use outside the use case you're probably thinking of.

Share Improve this answer Follow

edited Jun 11 '16 at 5:46

answered Jun 11 '16 at 5:23
kag0
4,104 ● 5 ● 29 ● 55

5 Good job clarifying that the comparison is really between Bearer tokens and cookies. – Shaun Luttin Jul 19 '16 at 23:50

19

While cookies can increase the risk of CSRF attacks by virtue of them being sent *automatically* along with requests, they can decrease the risk of XSS attacks when the `HttpOnly` flag is set, because any script that is injected into the page won't be able to read the cookie.

CSRF: a user clicks on a link (or views images) on an attacker's site, which causes the browser to send a request to the victim's site. If the victim uses cookies, the browser will automatically include the cookie in the request, and if the GET request can cause any non-read-only actions, the victim site is vulnerable to the attack.

XSS: an attacker embeds a script in the victim site (the victim site is only vulnerable if inputs are not sanitized correctly), and the attacker's script can do anything JavaScript is allowed to do on the page. If you store JWT tokens in local storage, the attacker's script could read those tokens, and also send those tokens to a server they control. If you use cookies with the `HttpOnly` flag, the attacker's script won't be able to read your cookie to begin with. That said, the script they successfully injected will still be able to do anything JavaScript can do, so you're still hosed IMO (i.e., while they may not be able to read the cookie to send it off to their own server for use later, they *can* send requests to the victim site using XHR, which will include the cookie anyway).

Share Improve this answer Follow

edited Aug 25 '20 at 5:08
Pang
8,519 ● 144 ● 75 ● 113

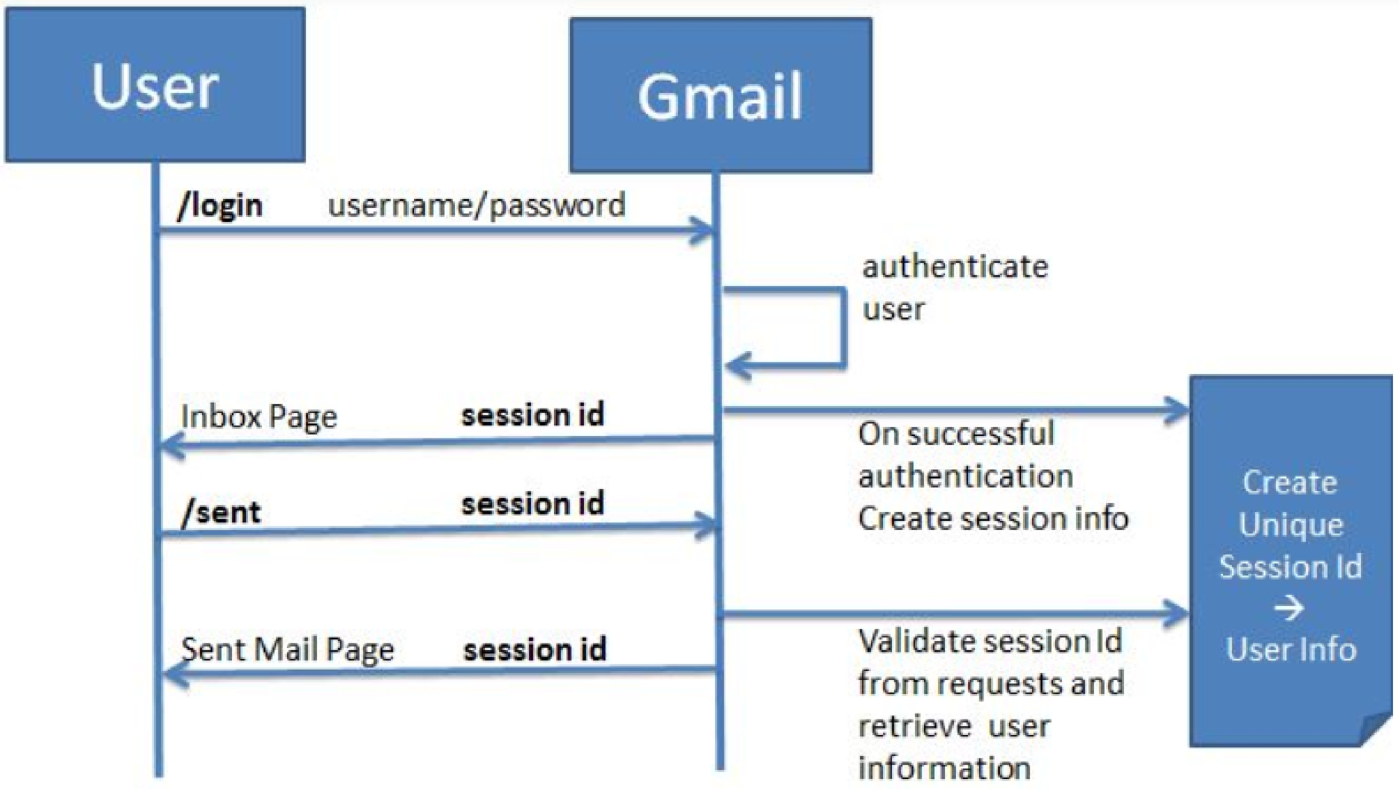
answered Oct 11 '19 at 0:09
cwa
842 ● 6 ● 12

9

Ref - Need for JSON Web Token

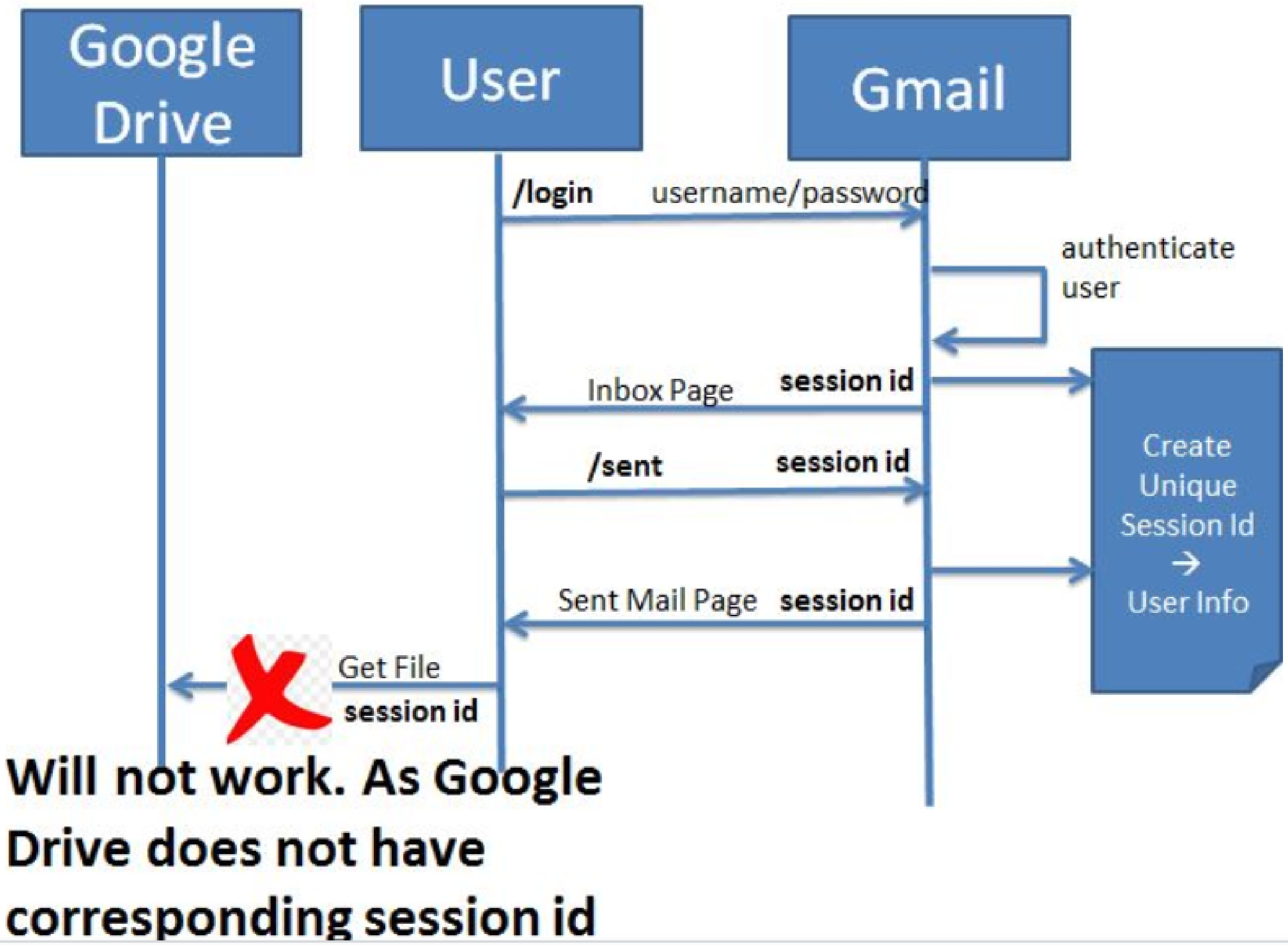
Cookies

In case of cookies, once the user has been authenticated then the Gmail Server will create a unique session Id. Corresponding to this session id it will store in memory all the user information that is needed by the Gmail server for recognizing the user and allowing it perform operations.
Also then for all subsequent requests and response, this session id will also be passed. So now when the server receives a request it will check the session id. Using this session id will check if there is any corresponding information. It will then allow the user to access the resource and return back the response along with the session id.

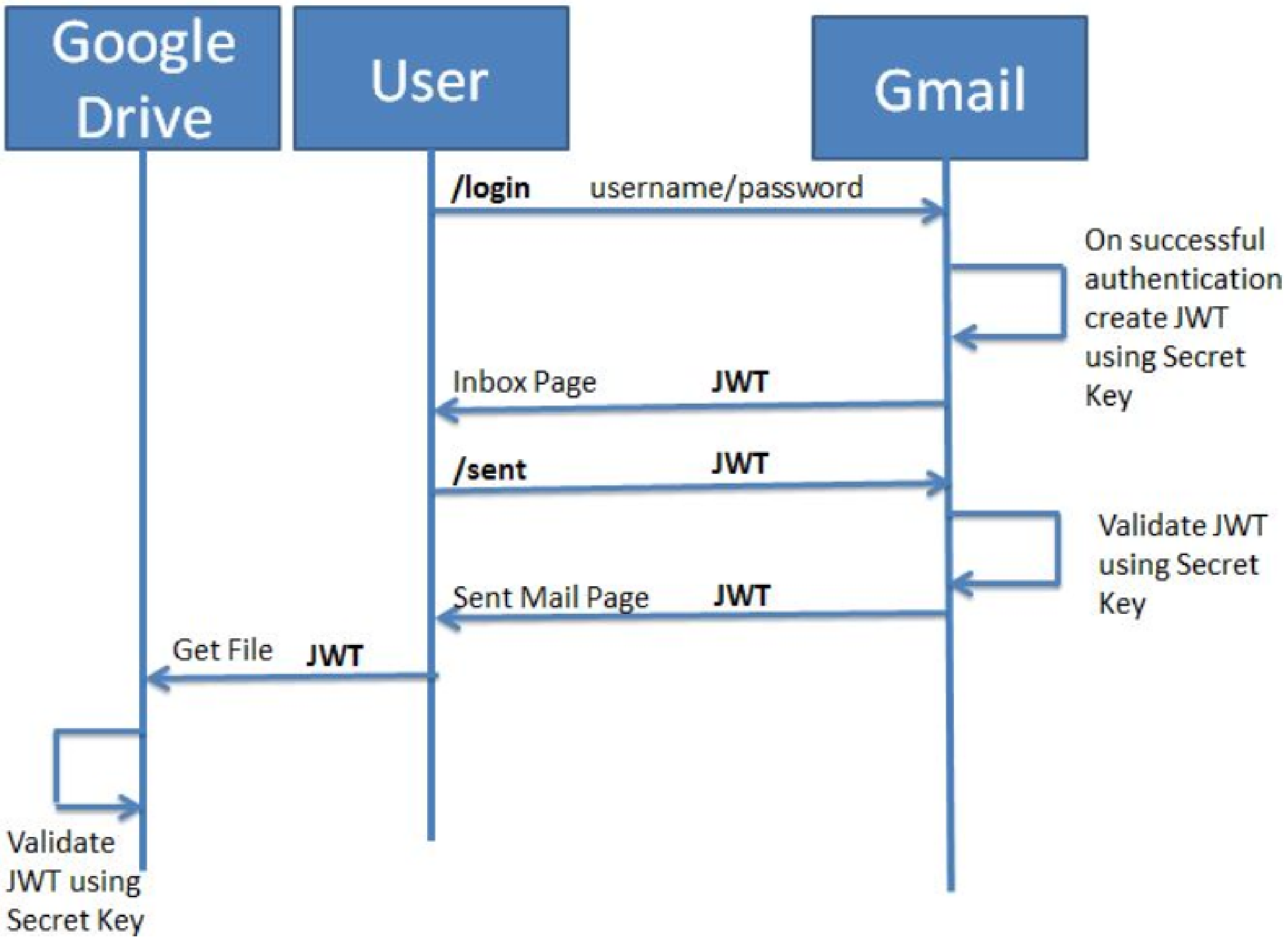

User
Gmail
/login
username/password
authenticate user
Inbox Page
session id
On successful authentication Create session info
/sent
session id
Create Unique Session Id → User Info
Sent Mail Page
session id
Validate session Id from requests and retrieve user information

Drawbacks of Cookies

- Cookies/session id is not self contained. It is a reference token. During each validation the Gmail server needs to fetch the information corresponding to it.
- Not suitable for microservices architecture involving multiple API's and servers



- JWT is self contained. It is a value token. So during each validation the Gmail server does not needs to fetch the information corresponding to it.
- It is digitally signed so if any one modifies it the server will know about it
- It is most suitable for Microservices Architecture
- It has other advantages like specifying the expiration time.


Google Drive
User
Gmail
/login username/password
On successful authentication create JWT using Secret Key
Inbox Page JWT
/sent JWT
Validate JWT using Secret Key
Sent Mail Page JWT
Get File JWT
Validate JWT using Secret Key

Share Improve this answer Follow

answered Aug 5 '20 at 15:37

B Batman Rises
161 ● 1 ● 3

If the user clicks "log me out of all sessions", then with each request the token must be validated by a database call - so the idea that it is self-contained doesn't hold. Short expiries might help but it's not perfect. – vaughan Sep 29 '20 at 19:57

0

A **JSON Web Token** is a safe, compact, and self-contained way of transmitting information between multiple parties in the form of a JSON object. It contains

- Header
- Payload
- Signature

The **header** Contains two parts **type** and **algorithm** that is to sign JWT Token

```
"typ": "JWT"
}
```

Payload Contains **actual data** that we want to send

```
{
"sub": "22222",
"name": "test",
"email": "test@xyz.com"
}
```

The **signature** is used to verify that the data was not altered before reaching its destination. We can achieve this functionality by using private keys.

As **JWT token** contains data but a **a cookie based token** may or may not contain data. The token is mainly generated by the server, which only recognizes any particular user. The token is used to protect a particular resource from unauthorized access.

If you want to have more details on the JWT Signing algorithm, LoginRadius can help you with easy implementation. Refer to the engineering blog article here.

Hope this answers the query !!

Share Improve this answer Follow

answered Feb 4 at 13:00

vaibhav jain
1