Protection against CSRF and XSS (Hashing + Encrypting)

Asked 4 years, 10 months ago Modified 4 years, 10 months ago Viewed 2k times

5

Security. Today, no application can survive the internet if it does not have proper security programmed into it - either by the framework used by the developer, or by the developer himself. I am currently developing a RESTful API to work using Bearer token authentication but have been reading about XSS and CSRF attacks.

Question 1) From what I've read, I see that applications consuming RESTful APIs that use token-based authentication are vulnerable to XSS and *not* CSRF if the token is stored in localStorage/sessionStorage of the browser instead of in cookies. This is because, for CSRF to work, the application must use cookies. Am I correct?

But now that the tokens are stored in the localStorage/sessionStorage, the application becomes vulnerable to XSS attacks. If there is any part of the application that does not sanitize inputs (E.g. Angular inputs are sanitized by the framework, but maybe a certain 3rd-party library I'm using is not sanitizing inputs by default), then an attacker can just inject malicious code to steal tokens of other users and then make authenticated requests by impersonating them.

Question 2) There is a way to protect against both of these attacks in your application that consumes a RESTful API. I came across **this post**. The gist of that article is that at the time of the user logging in and requesting a bearer token, have the server also return a `httpOnly` cookie that would act as, say, `CSRFProtectionCookie`. I believe the solution in the article is pretty robust and provides strong protection. Again, am I correct? What are your views?

My application and my version of the approach mentioned in the above post

The approach mentioned in the post above requires me to persist the `CSRFProtectionCookie` in a database of some sort. I do not want to do that. I do not want the database to be hit every time a authenticated request is made to the API. Instead, what I can do is this:

Login

1. User sends POST request to token endpoint with username and password
2. Authorization server validates user credentials and starts building JWT with certain claims.
3. As part of JWT-building, it also generates a random string, **hashes** it, and adds it as a, say, `csrf` claim to the JWT.
4. Also, next the server sets a `httpOnly` cookie named `XSRF-TOKEN` whose value is the same random string but **encrypted** this time.
5. Return the response to the browser. Browser gets both the bearer token as the response

1. Application calls authenticated endpoints by adding the JWT bearer token as the `Authorization` header. Browser automatically sends along the cookie.
2. Server decrypts the cookie to get plain text. Server then verifies this plain text with the hash present in the JWT.
3. If matched, proceed with request. Else, return Unauthorized response.

I think this gives protection against both CSRF and XSS. As both the token and cookie (httpOnly) is required to make authenticated requests.

Question Yes this eliminates the need to persist anything to the database (or does it? Am I missing some loophole?). But what is the overhead of decrypting and verifying hashes on every user request? Is it more overhead than database I/O? Also, any other suggestions to this approach or other approaches are welcome!

Thanks :)

rest xss csrf

Share Improve this question Follow

asked Apr 1, 2018 at 11:48

Anindit Karmakar
815 1 8 26

You don't have to persist anything to effectively protect against csrf, see double posting. XSS will not be fully mitigated by any framework, and you don't seem to understand XSS, so be careful with that one. Your method does not provide protection. Also Schneier's law applies. :) – Gabor Lengyel Apr 1, 2018 at 13:50

Schneier's law makes one feel like a fool xD But yes, I'll read up on double posting right away! Also, could you explain how/why this version does not provide protection? – Anindit Karmakar Apr 1, 2018 at 19:38

2 Answers

Sorted by: Highest score (default)

Answer to Question 1

2

From what I've read, I see that applications consuming RESTful APIs that use token-based authentication are vulnerable to XSS and not CSRF if the token is stored in localStorage/sessionStorage of the browser instead of in cookies. This is because, for CSRF to work, the application must use cookies. Am I correct?

More or less correct. An application that only uses a cookie for authentication and does not

> But now that the tokens are stored in the localStorage/sessionStorage, the application becomes vulnerable to XSS attacks.

It's not that the application becomes vulnerable to XSS, it's that the *authentication token* becomes vulnerable to XSS. If your authentication token is sent as `http-only`, then JavaScript (and thus XSS attacks) cannot read the value.

Answer to Question 2

I think the article does a pretty good job of explaining how to use a session cookie and CSRF token. Let me try to summarize the article, as it's a little long:

1. Use an `http-only` cookie (and ideally also `secure` — which the author did not mention — which prevents the browser from sending the cookie on an http request, only https requests) as the **authentication** token. In this way, the cookie cannot be read by JavaScript and thus cannot be stollen in an XSS attack.
2. Use session storage to store the **CSRF protection** token. The benefit here is that some bit of JavaScript code must read this value and put it into the request. Malicious code running on some 3rd party, compromised site will not be able to read the CSRF token from local storage, and thus will not be able to create a valid request.

The article also goes into details about how to set cross domain headers if your static resources are on a different domain than your REST endpoints.

However, your approach is backwards to what the article said. You're putting the authentication token in localStorage and putting the CSRF token in a cookie. I would suggest flipping it back around, as the authentication token is more important than the CSRF token, and an `http-only secure` cookie is harder for a bad guy to get at than localStorage.

Comments on your approach

> The approach mentioned in the post above requires me to persist the `CSRFProtectionCookie` in a database of some sort. I do not want to do that. I do not want the database to be hit every time a authenticated request is made to the API.

You are thinking along a good path, here. What you're describing would be called a "stateless CSRF token".

Your approach to "Authenticated Requests" looks good. I'm not entirely sure you need to encrypt the CSRF token value, but it doesn't hurt.

> this eliminates the need to persist anything to the database (or does it? Am I missing some loophole?)

> But what is the overhead of decrypting and verifying hashes on every user request? Is it more overhead than database I/O?

I/O is usually much slower than a CPU bound calculation, even for something like encryption. That said, if you're really worried about it, you'd have to measure (which is easier said than done, of course).

Final thoughts...

Keep in mind that by storing the CSRF token in a JWT claim, the CSRF token will be valid for as long as the JWT token is. Even if you send the client a new JWT token, that old token and CSRF token will still be valid. This isn't such a problem for short lived tokens (say 10-60 minutes), but one of the benefits of storing a CSRF token on the server-side somewhere means that you can have one-time tokens for super sensitive operations. (Also, if you need to revoke a not-yet-expired JWT token, you'd need to store that state somewhere server-side, too. But now we're falling down a different rabbit hole...)

There's really no particular framework that's going to completely prevent XSS vulnerabilities simply because they can manifest in so many different ways. That said, your web app can use the `Content-Security-Policy` header to help prevent XSS attacks. The CSP header can be used to tell the browser which domains it's allowed to load resources from (such as JavaScript files). It can even be used to prevent in-line JavaScript from running (which is one of the major attack vectors for XSS).

Share Improve this answer

Follow

edited Jun 20, 2020 at 9:12

Community Bot

1 1

answered Apr 2, 2018 at 19:38

kuporific

9,933 3 42 46

> I had so much fun reading this answer. You carefully dissected the question and answered the relevant bits. Thanks alot! As for flipping the auth token and cookie around - there's a reason I didn't. If I need to consume this application in a native application (e.g. android), then I can only consider the auth token for the requests (and I plan to distinguish the clients using the User-Agent header of each request. A browser-based application cannot modify the User-Agent header but a native app can set it to something like "User-Agent: NativeApp" - in which case, ignore absence of cookie.
> – Anindit Karmakar Apr 4, 2018 at 10:29

2

If you are using bearer authentication then CSRF is not possible by any attacker as they don't have the token value. It won't automatically be added by the browser to requests, therefore it is inherently secure against CSRF. See here for an explanation of CSRF and how a custom header can mitigate it. Bearer authentication solves the problem in the same way as the custom header cannot be added without CORS being enabled, or if CORS is active (which is common for an API), then as authentication against the API is achieved with the token value itself, any attacker will not know the value to provide (if they did they would simply use the value directly without CSRF), and the browser will not automatically supply the authentication

For protection against XSS, which is a different issue, ensure all HTML output is HTML encoded (`>` becomes `>`). This could be done in the API itself if it outputs formatted HTML, or in JavaScript if it is up to the consumer to format output appropriately. There are functions built into JavaScript and frameworks like JQuery that can also help (e.g. `textContent` and `text()`).

Share Improve this answer Follow

edited Apr 3, 2018 at 13:46

answered Apr 1, 2018 at 19:09

SilverlightFox
31.9k 11 74 146

Thank you! The first paragraph about CSRF completely makes sense. But my bigger concern is XSS because it seems easy for such a vulnerability to creep into your code if anytime the developer is not conscious about HTML encoding - maybe as a result of human error. – Anindit Karmakar Apr 2, 2018 at 6:37

@AninditKarmakar unfortunately that's just the nature of the beast. XSS vulnerabilities are easy to create. – kuporific Apr 2, 2018 at 19:39