Token Up

Keeping Hands out of the Cookie Jar

Erin Browning
Slides are available at:

https://www.frowning.wtf/token-up
$whoami
Erin Browning
Senior Security Engineer
a.k.a., I'm a hacker
You can contact me at:

erin@frowning.wtf

@efrowning
Slack is hiring!

Slack is used by millions of people every day – we need engineers who want to make that experience as secure and enjoyable as possible.

slack.com/jobs
All of the code-blocked frowning.wtf URLs are fake.
Please don't attack my website.
DON'T DO CRIMES
The Problem
I'm an attacker.

I want to take over accounts on your website.
Your site probably looks like this:

API: www.frowing.wtf/api

Frontend client: www.frowing.wtf

Mobile clients
Your sessions probably look like one or more of these:

- JWTs
- Randomized session token
- API tokens
As you break your website out from a monolith to microservices, how do you store sessions/tokens between an API and your browser-based frontend client?
Don't store tokens in local storage

Browser local storage (or session storage) is not a secure place to store sensitive information. Any data stored there:

- Can be accessed through JavaScript.
- May be vulnerable to cross-site scripting.

If an attacker steals a token, they can gain access to and make requests to your API. Treat tokens like credit card numbers or passwords: don't store them in local storage.
Is storing your auth in local storage that bad?

Let's find out.
In this talk:

- Common vulnerabilities that execute in a browser
- Modern application structures
- How to take advantage of browser-based protections
1UP
Two common attacks
XSS
What is XSS?
Cross site scripting
Attacker created javascript is executed in the user's browser in the context of site the user visited

More at: owasp.org/index.php/Cross-site_Scripting_(XSS)
How?
Injection!
Cannonical testing example:

```html
<script>alert(1);</script>
```

this is a homepage.
What can you do with XSS?

- Steal cookies
- Take actions as the user
- Like changing passwords
- Change page content
CSRF
What is CSRF?

Cross-Site Request Forgery

More at: owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)
Forces a user to perform actions they didn't intend on a website to which they're authenticated
How?

By default, cookies are included in requests sent cross domain.
user's browser

TAB 1  x  TAB 2

frowning.wtf - x

malicious script

TAB 1  x  TAB 2

bank.example - x

user is logged in, checking their account

sends request to bank.example + includes the cookie stored in the browser!

bank.example's servers

no CSRF protection
money is transferred
What do these attacks have in common?
All of these attacks execute in the user's browser
How are these attacks different?
XSS > CSRF
ANYTHING YOU CAN DO
I CAN DO BETTER
How can we reduce the impact of these attacks?
ARCHITECTURE
First, let's talk about a typical application structure:

- **www.frowning.wtf** - contains your frontend + any monolith code
- **www.frowning.wtf/api** - api
- **www.frowning.wtf/admin** - administrator site
Where is your authentication stored in the browser?
Probably in a cookie

That cookie is probably scoped to *.frowning.wtf

If not, it'll be in local storage, placed there by your javascript
Interactions are going through XHR to /api

For those of you who don't do frontend work:
XHR is an API called XMLHttpRequest.
It lets you transfer data between a web browser running JS
and a server without reloading the page.
Traditional CSRF protection stores a random token in a form in an HTML page. That token gets stored on the server as well. When the form is submitted, the token is sent with the form data and validated on the server.
Your API may be using a CSRF token, or it may just be relying on monolith form CSRF protection--aka, your api may be vulnerable.
Improvement: use subdomains
Now you have:

- api.frowning.wtf
- www.frowning.wtf
- admin.frowning.wtf
You can scope cookies to www, admin and api instead of using *.

The API cookie can have the secure and HTTPOnly flags set.

Secure means that cookie will only be sent over HTTPS

HTTPOnly means js can't touch it

Yes, the names are confusing, so remember: for HTTPOnly, only HTTP requests can access the cookie.
You XHR your requests to api from www.
How do you even do CSRF protection to your API?
 Depends on ~content types~

- `multipart/form-data`
- `text/plain`
- `application/x-www-url-form-encoded`
- `application/json`
- `application/xml`
• multipart/form-data, can go cross origin
• text/plain, can go cross origin
• application/x-www-url-form-encoded, can go cross origin
• application/json, can't go cross origin without CORS
• application/xml, can't go cross origin without CORS
What is CORS?
We care about CORS because of the protection offered by the Same Origin Policy (SOP).
What is the Same Origin Policy?

Lots of requests can't be made from URL1 to URL2 if they differ on the following things:

- Protocol (e.g., HTTP vs HTTPS)
- Port
- Host
CORS must be set on the assets you are accessing.
How do you reduce the impact of XSS?

API isn't running js.

www could still be vulnerable, and the site could send requests through XHR.
Improvement: use iFrames
Instead of using CORS, create an iFrame on www to api.

Use the `window.postMessage` API

docs at developer.mozilla.org/en-US/docs/Web/API/Window/postMessage
window.postMessage() enables cross-origin communication through DOM-based events. Windows can send and receive messages from each other through events.
window.postMessage()
ALWAYS ALWAYS ALWAYS validate your origin

if (event.origin !== www.frowning.wtf)
    return;

// .. otherwise do some stuff
Improvement: use websockets
Verify your Origin header.
The attacker would need to fake the origin header in the victim's browser.

Modern browsers don't let you set your own origin header.
Common problems:

- Sites don't check auth before upgrading the connection. The protocol upgrade request will have access to the browser's cookies. Therefore, check auth when upgrading.
Similarities

iFrames and websockets both have trustworthy origins in the browser.
Ultimately, how can you avoid CSRF hitting your API?

By dropping all requests to API that aren't `application/json`
How can you avoid XSS?
You can't completely.
Should you store auth in local storage?
Not unless you're sure you can prevent XSS.
How do you know you have a good understanding of all this?

Tell me why XSS is worse in all these cases.
Special Thanks

lvh @latacora
Leigh Honeywell @tallpoppy
The latacora team
The Product Security teams @Slack
see you on the internet bb