Next.js and cache poisoning: a quest for the black hole

Next.js and cache poisoning: a quest for the black hole - Author not stated, zhero_web_security.

- Published: 2024-06-24
- Original: <https://zhero-web-sec.github.io/research-and-things/nextjs-and-cache-poisoning-a-quest-for-the-black-hole>
- Preserved from: https://zhero-web-sec.github.io/research-and-things/nextjs-and-cache-poisoning-a-quest-for-the-black-hole (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.

Introduction

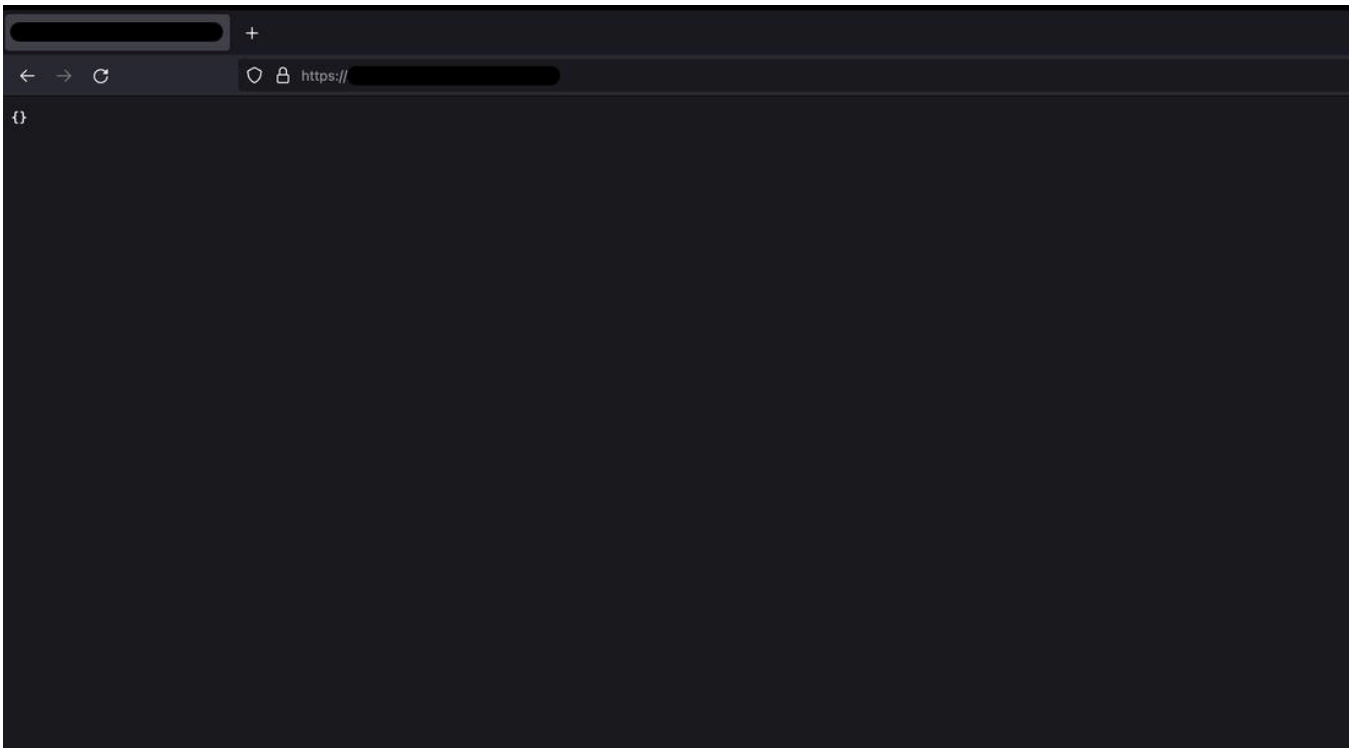
In search of challenges, zero-days, and bounties, I focused on widely used software to find interesting cases of cache-poisoning. My attention quickly turned to **Next.js**, an open-source JavaScript framework based on React, developed and maintained by Vercel. The Next package is downloaded more than 6 million times per week, indicating its extensive use. A discovery in this software could potentially affect many users, which was all the motivation I needed to roll up my sleeves.

This article will focus on cache-poisoning. If you are unfamiliar with this vulnerability, I highly recommend starting by checking out my write-up titled “DOS via cache poisoning on Mozilla”.

Index

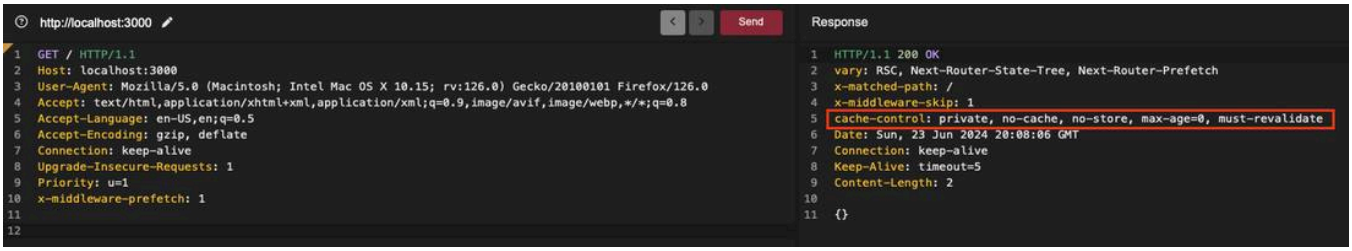
First vulnerability: Start of research, disillusionment and n-day

I quickly discovered an interesting behavior **related to the Next.js middleware**. When prefetching data for SSR (server-side rendering) pages, adding the `x-middleware-prefetch` header results in an empty JSON object `{}` as a response. If a CDN or caching system is present, this empty response can potentially be cached —depending on the cache rules configuration— rendering the targeted page **impractical and its content inaccessible**.

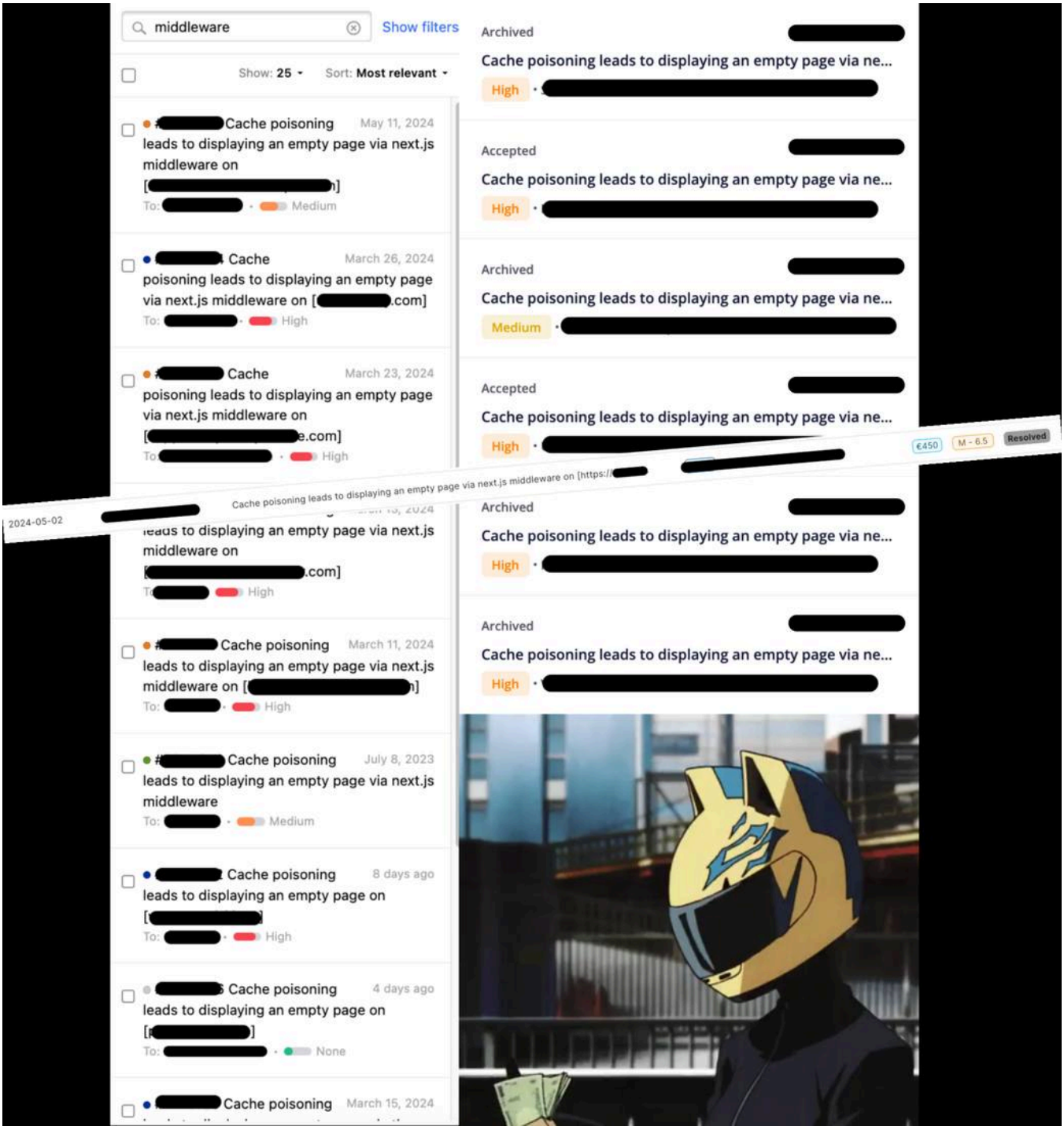


The primary condition for reproducibility/exploitation is the presence of a caching system. Without it, the response would not be cached, and the behavior resulting from adding the header would be harmless.

I won't delve into the technical details of the `getServerSideProps` and `getStaticProps` functions here, but they are used to retrieve server-side data and generate static pages, respectively. Since version 13.4.20-canary.13, Next.js has added **cache-control** to SSR responses to prevent them from being cached.



Excited by the idea of having discovered a fresh zero-day vulnerability, I quickly realized that this vulnerability already had a CVE assigned (**CVE-2023-46298**). However, since the proof of concept (POC) had not yet been shared, I was still able to exploit this vulnerability on a large scale. Several dozen bug bounty programs were affected, resulting in numerous bounties, which was highly motivating and encouraged me to continue my research.



flex info: non-exhaustive list

Impact and severity

This is a type of DOS via cache-poisoning with a CVSS score of 7.5:
`CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:U/C:N /I:N/A:H/CR:X/IR:X/AR:X`

Second vulnerability: React Server Component and CDNs, a food poisoning

To begin with, what is React Component Server (RSC)? Answer from the official next.js documentation:

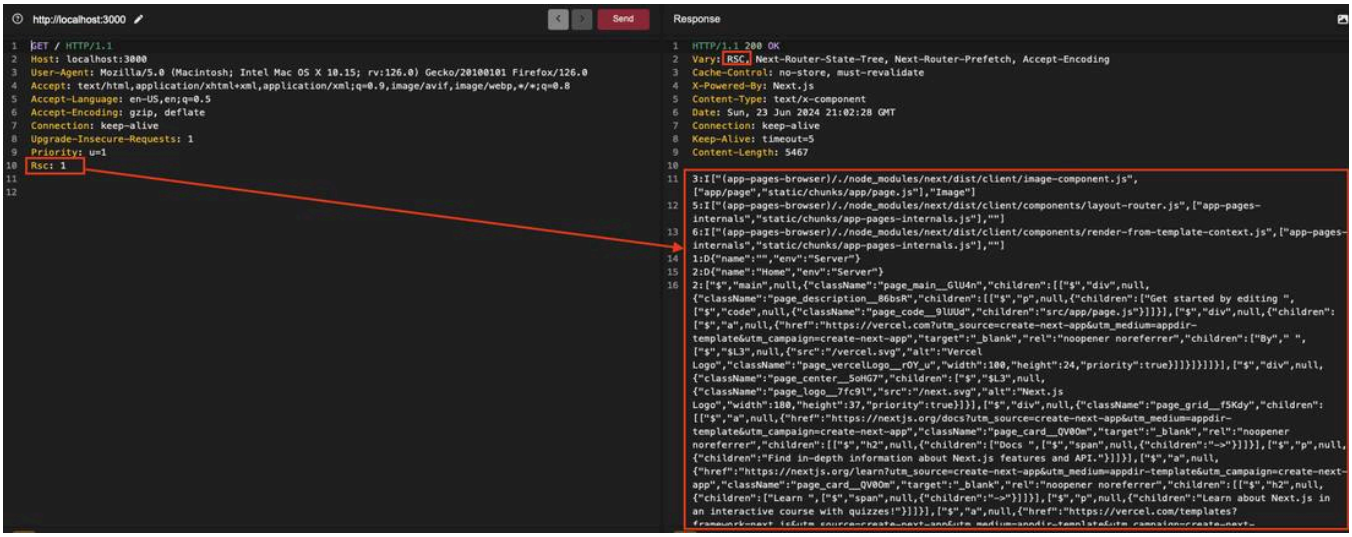
>

The RSC Payload is a compact binary representation of the rendered React Server Components tree. It's used by React on the client to update the browser's DOM. The RSC Payload contains:

- The rendered result of Server Components
- Placeholders for where Client Components should be rendered and references to their JavaScript files
- Any props passed from a Server Component to a Client Component

To put it simply, this feature allows you to render React components directly on the server before sending them to the client. This improves performance by reducing the client-side load and allows direct access to server-side resources.

During my research on Next.js applications, I often encountered requests retrieving the RSC payload. Initially, nothing about this aroused my curiosity. The header responsible for this behavior — `Rsc: 1` — was added to the cache-key via the `Vary` response header, **theoretically** preventing cache-poisoning.



By inspecting my HTTP history on my proxy, I noticed that the requests fetching the RSC payload included a **URL parameter** that was systematically added: `_rsc=randomValue`.

I quickly understood that this was a **cache-buster used to prevent caching an "unwanted" response**, which was surprising given that, as mentioned previously, the header was added to the cache-key via `vary`. After some research, I came across a GitHub ticket from a few months ago where a user complained about seeing the RSC payload cached, which prevented access to the content. Tim Neutkens, one of the framework maintainers, responded (on Apr 20, 2023):

>

To clarify App Router adds a few headers to the fetch when navigating client-side to get the RSC payload instead of HTML. Next.js on the server sets the Vary header to ensure that browsers /

CDNs can cache based on the headers passed to the fetch: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Vary

Going to close this issue as the Vary header is provided so this is a CDN configuration problem.

So it was clear, **the framework was cache-poisoning itself and the addition of the URL parameter was to prevent this behavior**. After examining the commit history, I noticed that the semblance of a "fix" was pushed on June 12, 2023, 2 months after timneutkens's response on the github ticket.

[image: image] [image: image]

Quick reminder about the function of the `vary` response header from the Mozilla documentation:

>

The Vary HTTP response header describes the parts of the request message aside from the method and URL that influenced the content of the response it occurs in. Most often, this is used to create a cache key when content negotiation is in use.

The maintainers of next.js therefore believe that it is not their responsibility given that the framework systematically adds the header -among others- in the `vary` header. Non-compliance with this HTTP standard is therefore not their responsibility but rather the responsibility of CDNs that do not adhere to these requirements.

Do CDNs really not respect `vary` values?

As surprising as it may be, after digging into the documentation of various CDNs, it turns out to be true under certain conditions. Here's what I found with some popular CDNs:

- At **Cloudflare** it is categorical (from official documentation) :

>

`vary` — Cloudflare does not consider vary values in caching decisions. Nevertheless, vary values are respected when Vary for images is configured and when the vary header is vary: accept-encoding.

- **Cloudfront** removes or replaces some headers, including `vary` :

>

If you configure your origin to include any other values in the Vary header, CloudFront removes the values before returning the response to the viewer. (from official documentation)

- At **Akamai** there is a behavior called **Remove Vary Header** allowing, as its name suggests, to remove the `vary` header. Akamai edge servers don't cache responses that include the `Vary` header, even if the content is cacheable by definition but there are "exceptions" (from official d ocumentation) :

Implementation

To enable caching of these responses via Akamai edge servers, you must remove the `Vary` header. You can accomplish this in one of two ways:

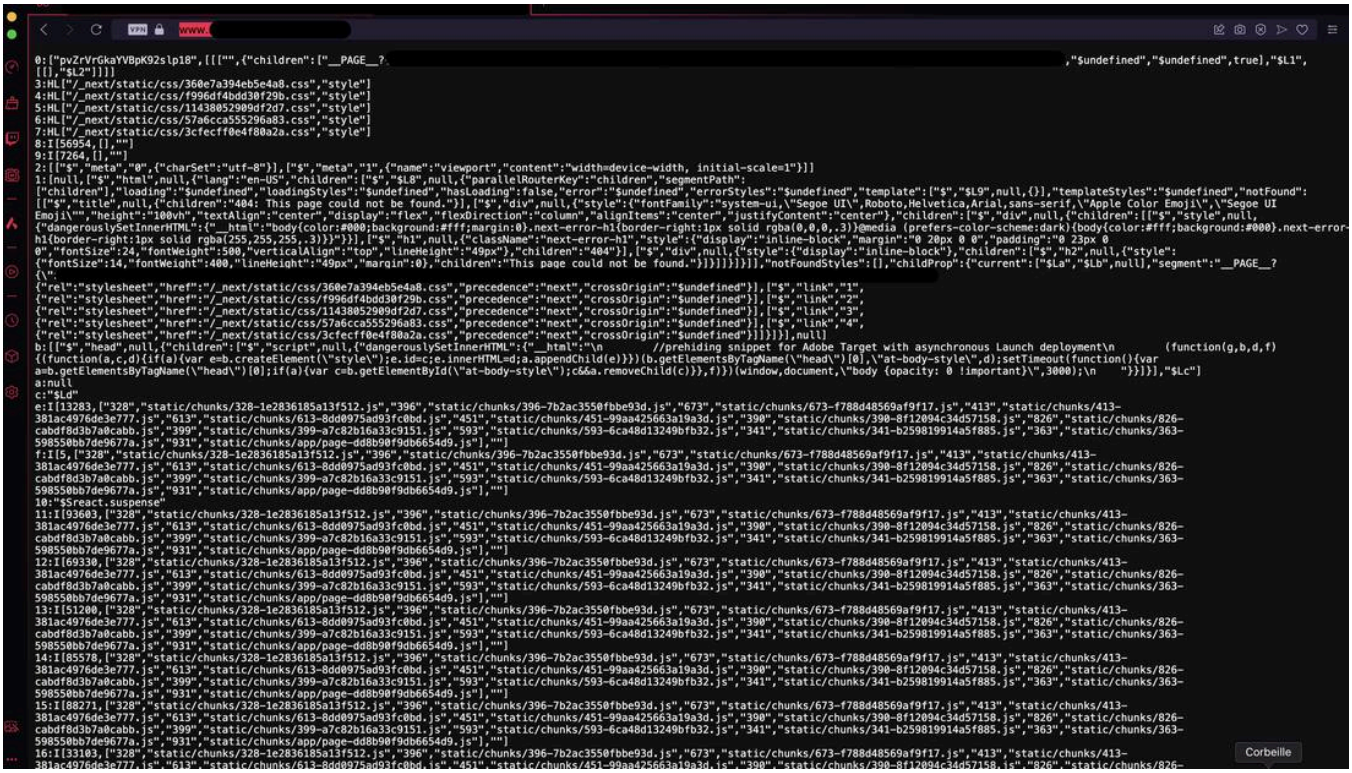
- **Remove the `Vary` header at the origin server**. It isn't needed here. (This is the preferred method.)
- **Include the Remove Vary Header behavior in your property and set it to ON**. This will remove Vary headers other than "Vary: Accept-Encoding" from the response.
- **This behavior and Adaptive Media Delivery, Download Delivery, and Object Delivery**. By design, these products assume that all content is cacheable. So, the **Remove Vary Header** behavior is automatically included in the background, and it's set to **On**. You can override this default by adding this behavior to the Default Rule and setting it to **Off**. But, you'll see a warning message stating that you're deviating from the best practices established for these products. This warning is informational, only.

The default behavior of the "**Adaptive Media Delivery**", "**Download Delivery**" and "**Object Delivery**" products is therefore to remove the `vary` response header.

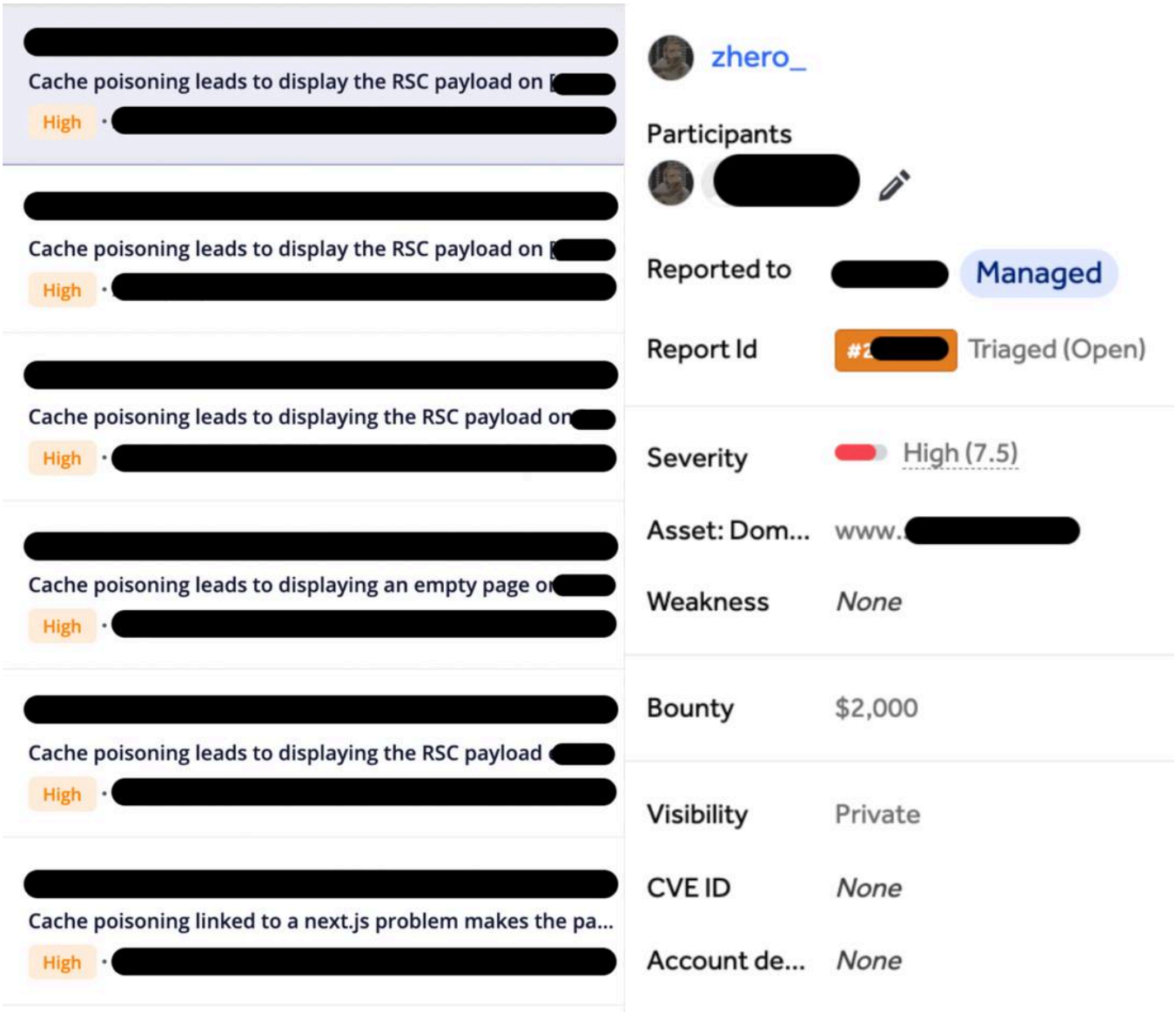
So, is this exploitable?

We discussed earlier the self-cache-poisoning issue caused by the framework, prompting Next.js developers to implement a "fix": **adding a URL parameter during fetch operations to act as a cache-buster and prevent accidental caching of the RSC payload**. However, this measure does not prevent attackers from exploiting the vulnerability by sending a request with the `Rsc` header without using a cache-buster, which mimics typical cache-poisoning attacks. Whether this attack succeeds naturally depends on **the CDN and its cache-rules**.

In theory it was exploitable, I had to be able to confirm all that with a real target, and after some research I got my first hit:



The cache was **poisoned** and the main/root page returned the react component server instead of the initial content, resulting in a **bounty of $2000**. From there and as always, pattern extractions, template creation and mass scan of my database containing my bug bounty assets then sending reports to vulnerable programs, here are some of them:



flex info: non-exhaustive list

Impact and severity

This is a type of DOS via cache-poisoning with a CVSS score of 7.5:
`CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:U/C:N /I:N/A:H/CR:X/IR:X/AR:X`

Escalation of the vulnerability to the security team

I contacted the Next.js team and discussed the issue with them. They concluded that **there was nothing more they could do at the framework level**. While I understand it's not entirely their fault/responsability, I believe leaving it unresolved without informing users so they can implement temporary solutions to mitigate the risk is a concern.

Timeline:

- Vulnerability reported on March 23, 2024
- Vercel first response on April 3, 2024
- Vercel asks for additional information on April 17, 2024
- Vercel final response April 29, 2024

Third vulnerability: Internal header, HTTP status and error page

After taking a short break from my research, I decided to dive back in. I had the Next.js source code displayed on each of my screens, a cup of coffee in hand, and armed with my trusty companion for debugging, good old `console.log` (yes?). BismiLlah, I was ready for some intensive code review.

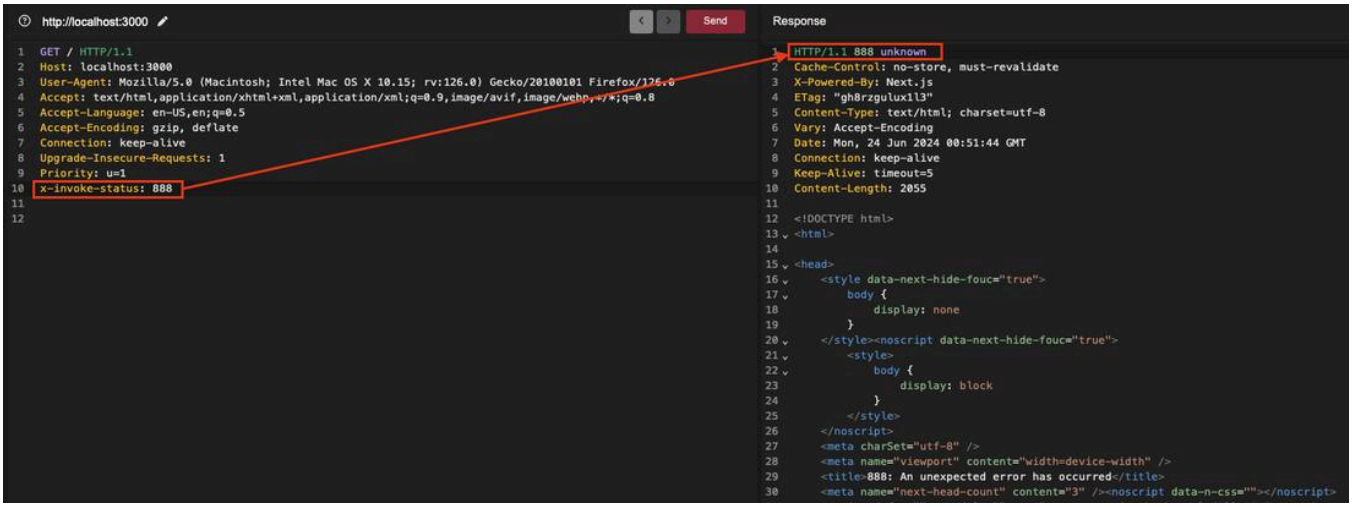
A little further into my review, I stumbled upon a piece of code that immediately piqued my curiosity:

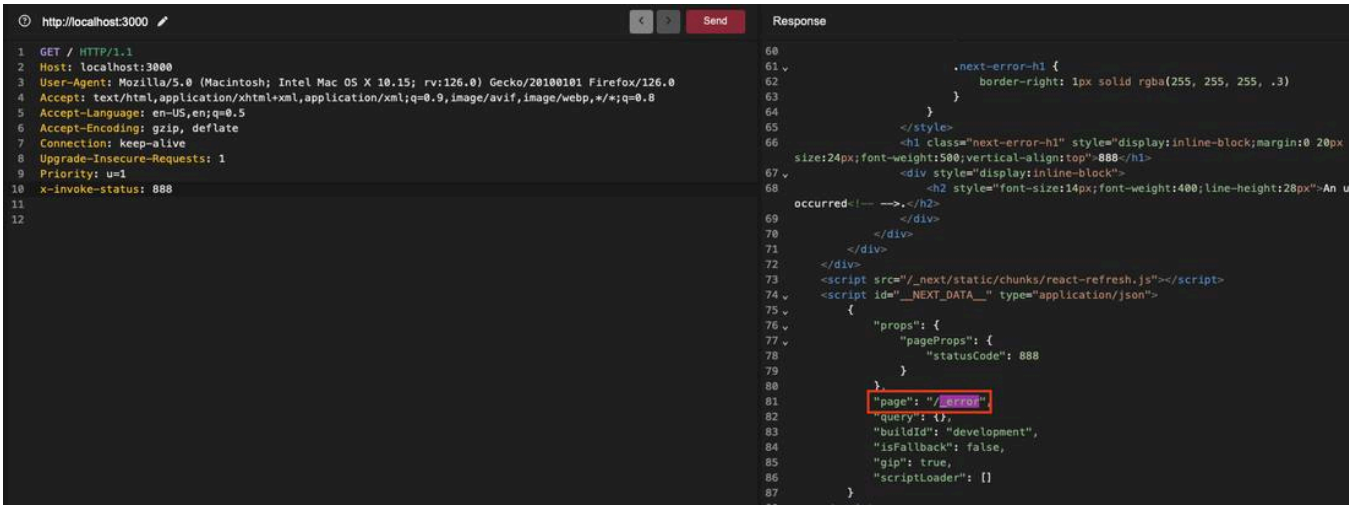
```
1260          // when x-invoke-path is specified we can short short circuit resolving
1261          // we only honor this header if we are inside of a render worker to
1262          // prevent external users coercing the routing path
1263          const invokePath = req.headers['x-invoke-path'] as string
1264          const useInvokePath =
1265            !useMatchedPathHeader &&
1266            process.env.NEXT_RUNTIME !== 'edge' &&
1267            invokePath
1268
1269          if (useInvokePath) {
1270            if (req.headers['x-invoke-status']) {
1271              const invokeQuery = req.headers['x-invoke-query']
1272
1273              if (typeof invokeQuery === 'string') {
1274                Object.assign(
1275                  parsedUrl.query,
1276                  JSON.parse(decodeURIComponent(invokeQuery))
1277                )
1278              }
1279
1280              res.statusCode = Number(req.headers['x-invoke-status'])
1281              let err: Error | null = null
1282
1283              if (typeof req.headers['x-invoke-error'] === 'string') {
1284                const invokeError = JSON.parse(
1285                  req.headers['x-invoke-error'] || '{}'
1286                )
1287                err = new Error(invokeError.message)
1288              }
1289
1290              return this.renderError(err, req, res, '/_error', parsedUrl.query)
1291            }
1292
```

So, under certain conditions - *which we will see later* - it is possible to **overwrite the status code** via the value of the `x-invoke-status` header directly provided in the request (`req.header[]`) and **invoke/return the error page** (source). Very interesting, I keep digging and I see that `x-invoke-status` is an **"internal" header**, so they are normally stripped when they come from the client:

```
13    /**
14     * Headers that are set by the Next.js server and should be stripped from the
15     * request headers going to the user's application.
16     */
17    export const INTERNAL_HEADERS = [
18      'x-invoke-error',
19      'x-invoke-output',
20      'x-invoke-path',
21      'x-invoke-query',
22      'x-invoke-status',
23      'x-middleware-invoke',
24    ] as const
25
```

A small local test was obviously necessary to verify if this was indeed the case, or at least to explore the possibility under certain conditions. Initially, I tried with the default configuration and included `x-invoke-status: 888` as a header, and on my first try:





At this point, I have no doubts about the exploitability of this vector for cache-poisoning. By specifying the HTTP status code `200`, it allows compliance with most caching rules (*which typically do not initially allow caching of error codes*). This enables caching the contents of the error page, which can then be served as a response instead of the main page.

Note: The other two vulnerabilities discussed above are particularly interesting for the same reason: the behaviors induced by adding their respective headers result in a `200` response, **which aligns with the majority of cache-rules**.

How the code works

As specified in the code, under certain conditions, the value of the `x-invoke-status` header overwrites the value of the `statusCode` property of the response object. Then the `/_error` page is returned (*whatever the HTTP code specified*):

```
res.statusCode = Number(req.headers['x-invoke-status'])

(...)

return this.renderError(err, req, res, '/_error', parsedUrl.query)
```

It is therefore possible to specify any HTTP code to **alter the response code and return the error page** (`/_error`). Generally, CDNs and caching systems are configured not to cache HTTP error codes. However, an attacker can specify the code `200` to align with cache rules and effectively "force" the caching of the error page, as demonstrated earlier.

Three conditions are necessary for this behavior to be possible:

```
const useInvokePath =
    !useMatchedPathHeader &&
    process.env.NEXT_RUNTIME !== 'edge' &&
    invokePath
```

- `useMatchedPathHeader` must return `false` , this is the case if:

```
const useMatchedPathHeader =
    this.minimalMode && typeof req.headers['x-matched-path'] === 'string'
```

>

Minimal mode is an experimental config being tested in fully featured Next.js environments like Vercel that have an edge network to handle specific parts of requests that next-server would normally handle. (source)

-

Next.js has two server runtimes, the nextjs runtime used must not be `edge` . The runtime used by default is `Node.js`

-

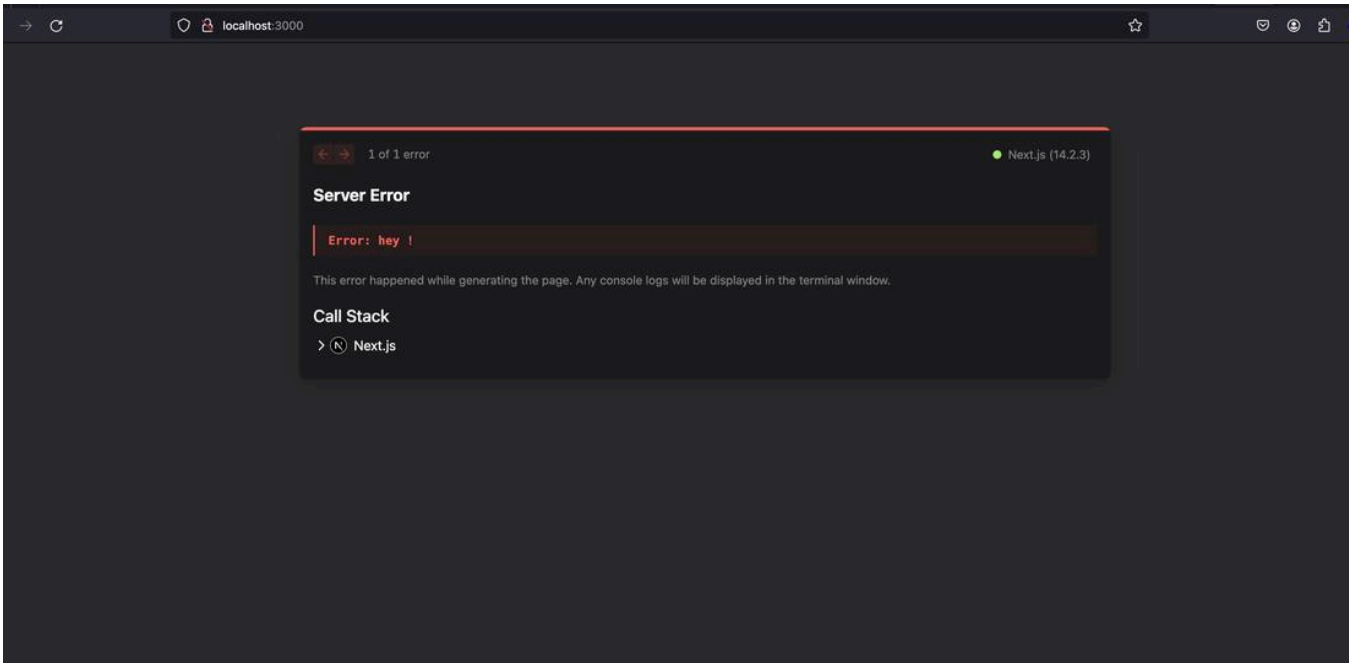
`invokePath` should return `true` :

```
const invokePath = req.headers['x-invoke-path'] as string
```

After conducting local tests, I found that this header is not considered when provided in the request, and its value does not appear to be retrieved. However, the header's value is automatically added and corresponds to the current path. Therefore, the constant will always return `true`.

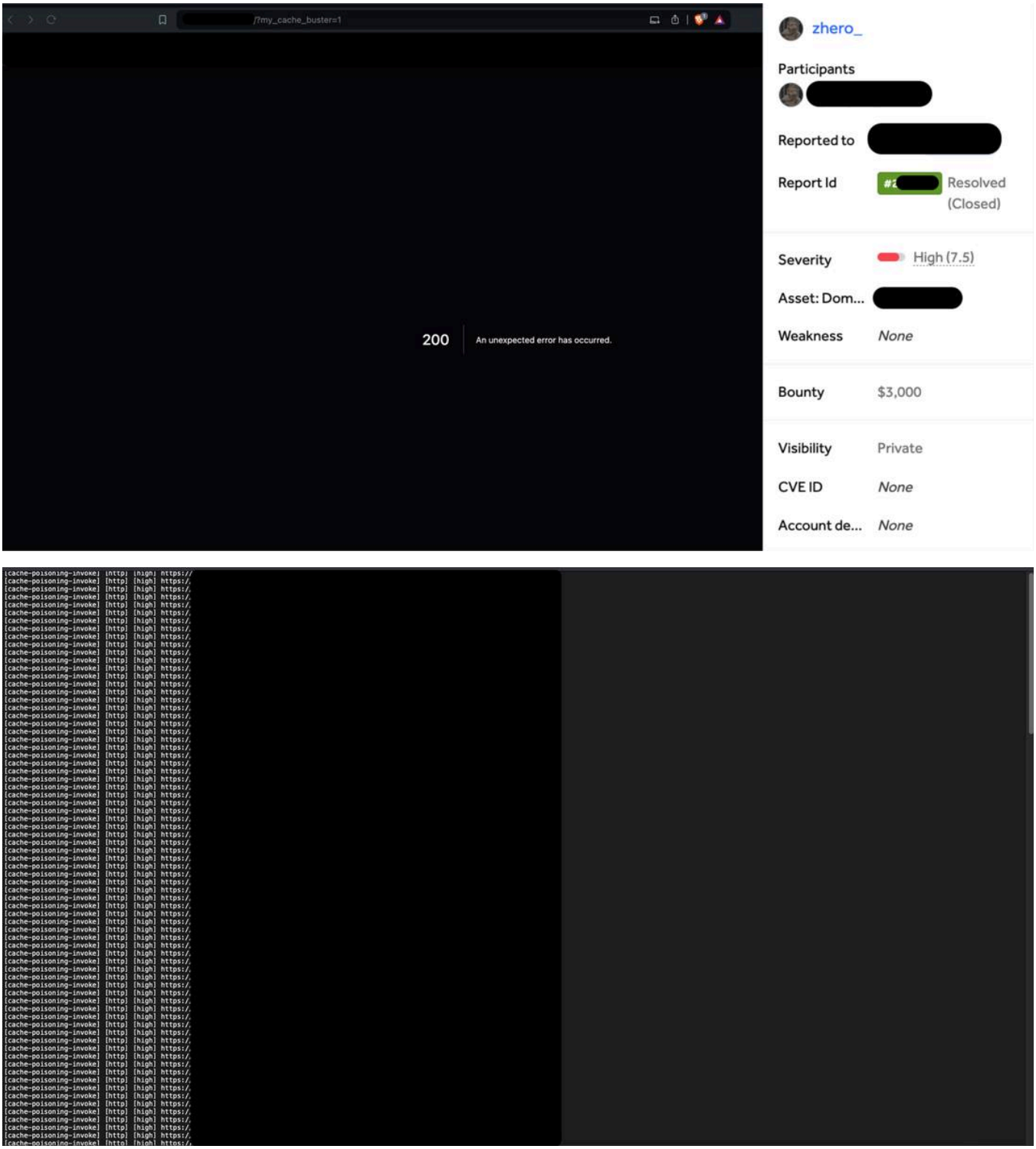
If the `x-invoke-error` header is provided and that the error page template expected it, it is also possible to display a custom error message (*the value must be provided in json format* `{"message":"<>"}`):

```
if (typeof req.headers['x-invoke-error'] === 'string') {
    const invokeError = JSON.parse(
      req.headers['x-invoke-error'] || '{}'
    )
    err = new Error(invokeError.message)
  }
```



Real-world exploitation

With the vulnerability confirmed locally, I swiftly moved to validate it on real targets, bug bounty programs. Once again, I promptly achieved my first successful exploit, which was followed by many others, one of them resulting in a **bounty of $3000**.



Impact and severity

This is a type of DOS via cache-poisoning with a CVSS score of 7.5:
`CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:U/C:N /I:N/A:H/CR:X/IR:X/AR:X`

Escalation of the vulnerability to the security team

Upon reporting the vulnerability to the Next.js security team, **they initially acknowledged it as a valid bug** related to the experimental minimal mode functionality. They mentioned they had potential fix options.

During this period, due to my busy schedule, I didn't delve deeper to verify whether the issue indeed originated from minimal mode, trusting that they were addressing it on their end. However, I found it weird compared to the earlier conditions (*number 1*) I had observed.

Then comes the time for the response, I won't go into certain confidential details but to summarize, **they think that ultimately this vulnerability is the result of CDN caching rather than a vulnerability present in Next.js** and that the conditions for exploitability are limited, based on factors beyond Next.js (**WAF/CDN configuration**).

What was my surprise to see that their response was preceded a few hours earlier by a beautiful commit, implementing a fix for my discovery:

Why doesn't this make sense?

Whether or not the source of the problem came from the `minimalMode` does not change the fact that the value provided by the client via the internal `x-invoke-status` header was taken into account and had an impact on the response in overwriting the HTTP code and invoking the error page and **that's why they implemented a fix**.

The use of the internal header `x-invoke-status` is possible with the minimum configuration (default) to **override the status code and invoke the error page**. Now regarding caching, from the moment we were able to use this header - *normally **internal** as specified in the code (see above)* - without the framework adding it in `vary` (as was the case for the RSC header for example) to avoid potential cache poisoning, their responsibility is directly incurred, regardless of the caching system used (*or not*) by a development team.

As explained earlier, being able to invoke the error page by specifying a status `200` will **satisfy most cache-rules** (*most of which do not allow the caching of responses with an error status code*), and will allow the caching of this error page if a caching system is implemented.

- The possible use of an internal header having an influence on the response
- The fact that no system is put in place to avoid the use of the header and/or the caching of a response resulting from its use

The two points above directly **involve the framework**, and it is necessary for users to be aware of this vulnerability and to be able to protect themselves against it. Reason why I think a CVE is more than essential as was the case for `x-middleware-prefetch` -*although different*- implies the responsibility of the framework in the same way (also requiring a cache system for its exploitability...).

Timeline:

- Vulnerability reported on May 22, 2024
- Vercel first response on June 5, 2024
- Vercel validate the bug on June 13, 2024
- CVE proposal -from me- on June 13, 2024
- Vercel pushes a fix commit on June 18, 2024
- Vercel changes its mind regarding their responsability on June 18, 2024
- Vercel final response and case considered resolved on June 21, 2024

Conclusion

You will have understood, the Vercel security team considering the problem as resolved and "not considering that it is the result of a direct vulnerability in next.js" (*despite the commit made*) I am sharing this vulnerability to inform the community. Based on my limited experience, every time I have reported vulnerabilities/zero days to companies, I have always - *without exception* - received fairly disappointing responses. I obviously wouldn't want to make my case a generality, but that could discourage more than one researcher from taking the time to report vulnerabilities responsibly, which is rather regrettable. I can understand that some companies see CVE as a "badge of shame", but when software is widely used it is their responsibility to assume and demonstrate exemplary transparency.

For my part, I was able to send **several dozen reports** (*BBP only*) related to the vulnerabilities cited in this article and I was able to find **more than 200 vulnerable assets** (*maybe a lot more I'm not sure*).

Kudos to my friend Jayesh Madnani, with whom I was able to systematically scan my discoveries on his gigantic database—mine being less than half its size. It's always a pleasure to work with him, benefiting from his automation expertise to target as many sites as possible.

Bonus: responses to certain allegations aimed at lowering the severity of specific cache-poisoning reports

I'm pretty experienced at the exercise, I've often changed the severity of my CP reports from `Medium` to `High` or even from `Informative` to `Triaged`, sometimes going from $0 to $3000 with a few arguments. Take into account the fact that the concept of cache-poisoning is not always well understood, be as clear and explicit as possible, assuming that your interlocutor knows nothing about the subject.

Claim number 1: "The impact is not significant, caching only lasts XXXX seconds"

Response: The duration of the cache does not matter, an attacker only has to make a small script to send a request each X seconds (=caching-duration) so that the cache is permanently poisoned making the site completely unavailable. (*feel free to attach the script to the POC*)

Claim number 2: "The impact is not significant, cache poisoning is only effective on a predefined area, the response is normal when I access the link containing your cache-buster"

Response: As you probably know, cache/CDN systems are often configured to operate by zone for performance reasons. Despite this, an attacker in a real situation only has to send the request from IPs coming from different zones (*via a VPN*) in order to impact all users.

Claim number 3: "We calculated using CVSS and the rating is 5.3 `CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:L`"

Response: The CVSS score was not correctly set, the vulnerability allowing the page to be completely unavailable/impracticable for an indefinite period of time, the Availability parameter is set to High, which gives a score of 7.5 (high): `CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H/CR:X/IR:X/AR:X`

>

High (H): There is total loss of availability, resulting in the attacker being able to fully deny access to resources in the impacted component; this loss is either sustained (while the attacker continues to deliver the attack) or persistent (the condition persists even after the attack has completed). Alternatively, the attacker has the ability to deny some availability, but the loss of availability presents a direct, serious consequence to the impacted component (...)

Published in June 2024.
