Eclipse on Next.js: Conditioned exploitation of an intended race-condition

Eclipse on Next.js: Conditioned exploitation of an intended race-condition - Author not stated, zhero_web_security.

- Published: 2025-05-06
- Original: <https://zhero-web-sec.github.io/research-and-things/eclipse-on-nextjs-conditioned-exploitation-of-an-intended-race-condition>
- Preserved from: https://zhero-web-sec.github.io/research-and-things/eclipse-on-nextjs-conditioned-exploitation-of-an-intended-race-condition (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

Back to the Next.js framework this time, with a modest but rather interesting piece of research born out of a need to bypass the patch implemented following one of my previous vulnerabilities, CVE-2024-46982, still my favorite to date on Next. I strongly recommend reading the previous paper on the subject for a better understanding, namely "Next.js, cache, and chains: the stale elixir". I won't go into detail about the previous CVE, nor revisit some basic concepts already covered earlier (SSG, SSR..).

It all started some time ago, when I was looking for vulnerable targets (BBP) to the famous stale elixir. I came across an asset whose version was higher than that of the patch and exhibited strange behavior, prompting me to take another look at the source code, which led to the discovery of race condition, which - when chainable with a cache poisoning - allows bypassing the previous patch, potentially leading to an SXSS in the best/worst case scenario, depending on the point of view.

Its exploitability stems from **misconfigurations/poor implementation practices** that are for the most part outside the scope of Next.js's responsibility, making the `AC: H` metric in the CVSS vector fully appropriate, as you'll soon see. I still found it worthwhile to document, especially since there are a few real-world targets out there that remain exposed, enough to be of interest to some fellow hunters.

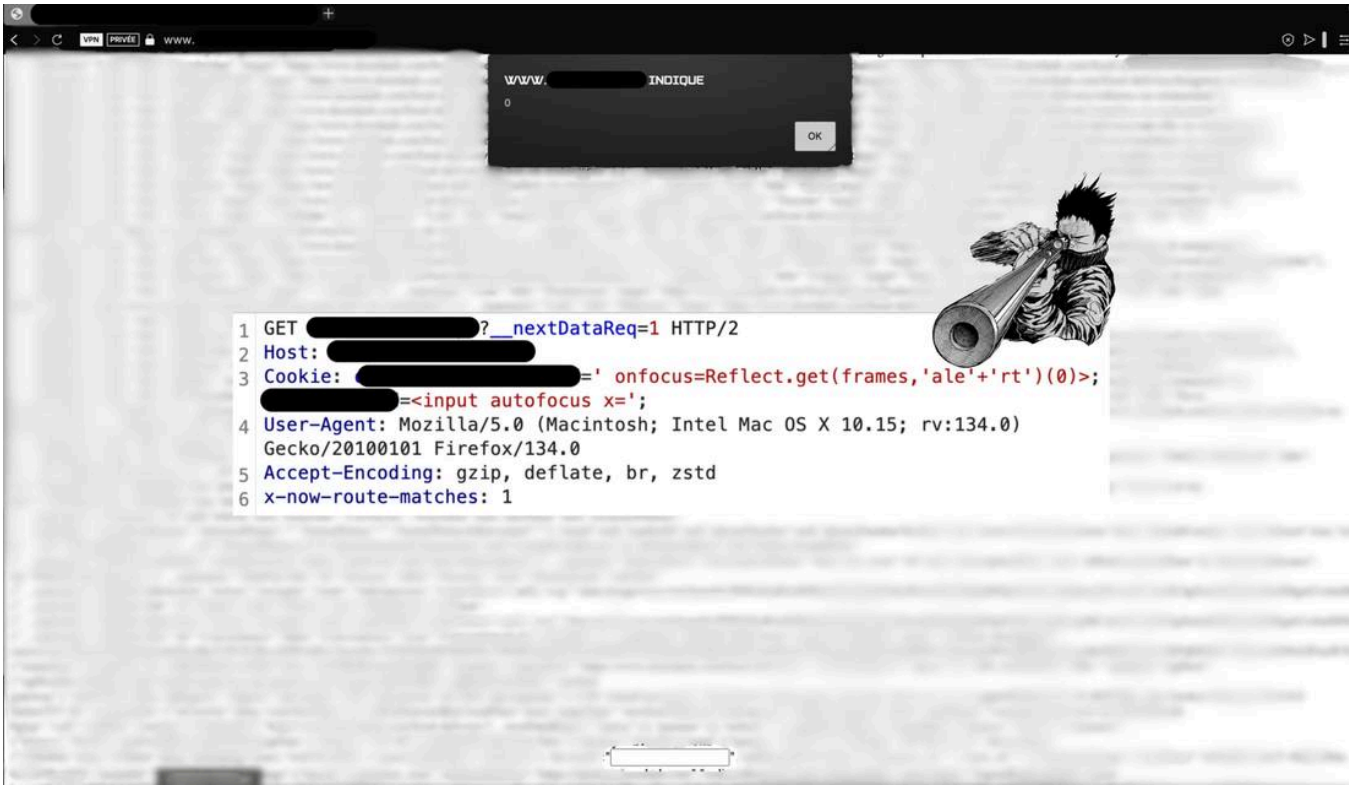
Index

Mission objective: "Bypass" CVE-2024-46982

The fix for CVE-2024-46982 was implemented starting from version **14.2.10**, after which the addition of the `x-now-route-matches` header no longer altered the `Cache-Control`. However, it was not stripped from the user request until version **15.1.6**, and would trigger a `500` error when added to a request for an SSR endpoint.

Although the `500` page triggered by the header can be cached on misconfigured CDNs, as I’ve been able to observe on some sites, the impact is limited to the DoS aspect, since the page itself isn’t altered, unlike the previous CVE.

As a reminder, CVE-2024-46982 directly impacted **the framework’s caching system**, allowing cache poisoning, regardless of the presence of a CDN system, which made the vulnerability particularly dangerous and its scope relatively broad. The impact was ranged from a DoS attack via page alteration, replacing the entire page with a `pageProps` object, to a stored XSS if a request element was passed through `getServerSideProps`, due to the content-type being `text/html` after response poisoning. *see more here*



Cache-Poisoning to Stored-XSS on a BBP (CVE-2024-46982)

The goal is therefore to find a way to obtain a complete alteration of the page again by pageProps under a text/htmt content-type, for versions between 14.2.10 and 15.1.6 (*the version from which the header is stripped*). Keeping in mind that the exploit this time will not impact the framework cache but an external one, poorly configured, thus reducing its scope significantly due to its very conditioned nature.

All local tests performed in this paper were done on version **15.0.4**, on the `pages` router (*router affected by CVE-2024-46982*).

Batcher and cache-key

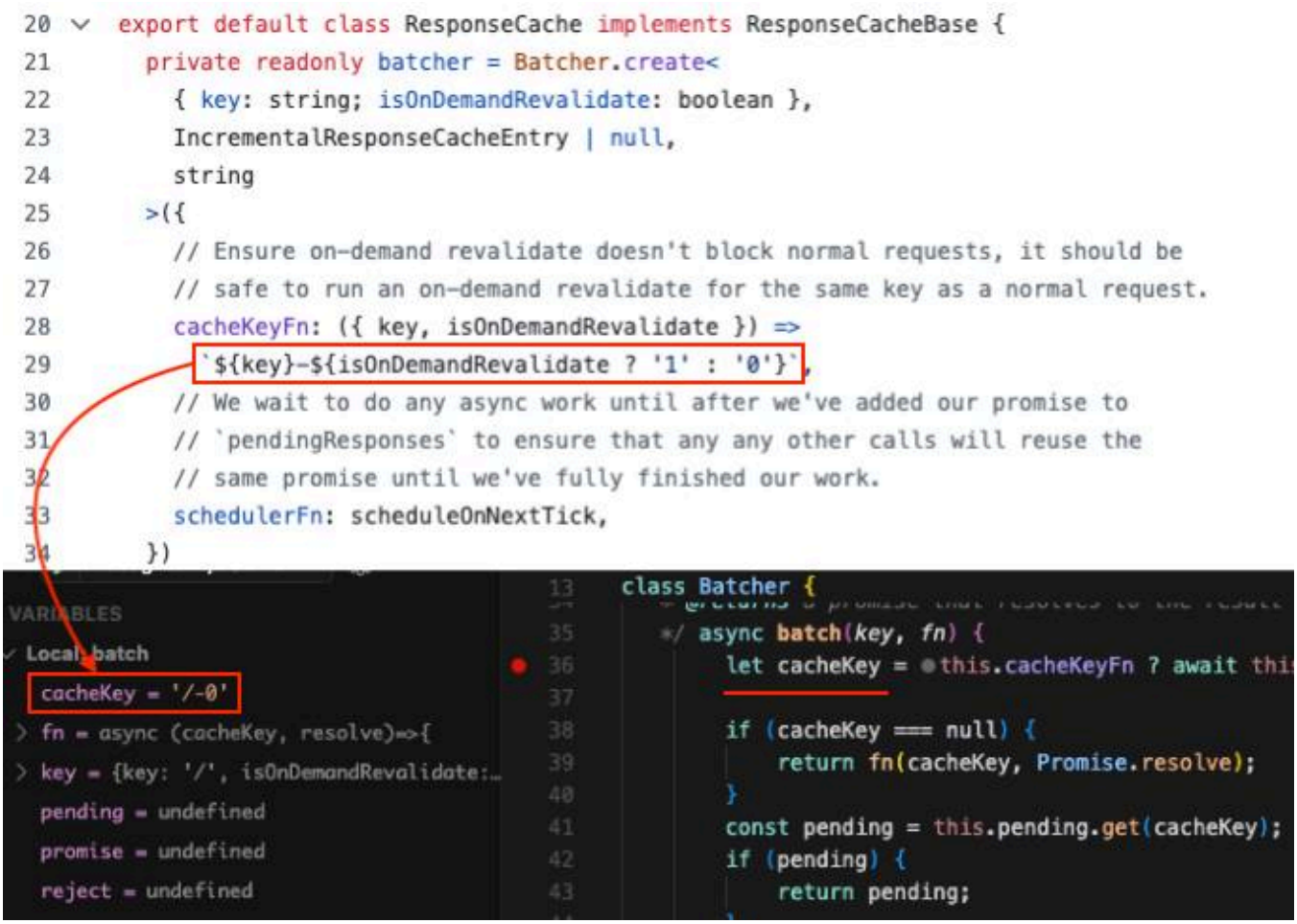
It all starts here, a batcher used as a promise anti-duplication mechanism: when several calls with the same key are made before the previous one completes, the batcher executes the `fn` function only once and shares the result with all callers.

```
64    public async batch(key: K, fn: WorkFn<V, C>): Promise<V> {
65      const cacheKey = (this.cacheKeyFn ? await this.cacheKeyFn(key) : key) as C
66      if (cacheKey === null) {
67        return fn(cacheKey, Promise.resolve)
68      }
69
70      const pending = this.pending.get(cacheKey)
71      if (pending) return pending
72
73      const { promise, resolve, reject } = new DetachedPromise<V>()
74      this.pending.set(cacheKey, promise)
75
76      this.schedulerFn(async () => {
77        try {
78          const result = await fn(cacheKey, resolve)
79
80          // Resolving a promise multiple times is a no-op, so we can safely
81          // resolve all pending promises with the same result.
82          resolve(result)
83        } catch (err) {
84          reject(err)
85        } finally {
86          this.pending.delete(cacheKey)
87        }
88      })
89
90      return promise
91    }
92  }
```

source: next.js/packages/next/src/lib/batcher.ts

New promises are created and stored with their `cacheKey` in a Map object named `pending`. When a call arrives, a check is performed to see if a promise already exists for this key in the Map, if so, the existing promise is reused avoiding recreating a new promise for the same operation. It serves as an optimization process that avoids repeating costly operations for identical concurrent requests and is **particularly used during cache revalidation and static page generation**.

The `cacheKey` used by the batcher is defined in response-cache/index.ts. It consists of the **requested path**, followed by a **dash** and either **0** or **1**, depending on whether it's an on-demand r evalidation request:



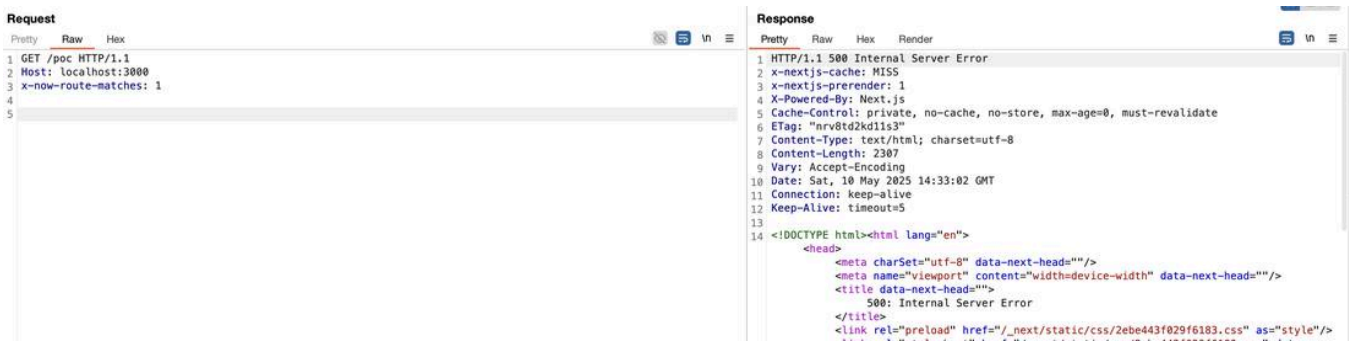
sources: /src/server/response-cache/index.ts and /src/lib/batcher.ts

As you may have noticed, a simple request to the root page - `/` - (*without on-demand revalidation*) is assigned the following key : `/-0`. Note that URL parameters are therefore **not taken into account** in the `cacheKey`.

Let's dissect

Once the promise is created, the `fn` function is executed; a callback responsible for generating/retrieving the expected data, whose result is then used to resolve the promise shared among all calls with the same key, as previously explained. A classic SSR page will therefore not trigger this function, the latter being - by default - neither generated nor revalidated/cached.

However, we know that the `x-now-route-matches` header allows an SSR request to be passed as a static request (`SSG`). Since the fix, simply adding the header results in a `500` error in the form of an HTML page (*without modifying the cache control*):



request containing the x-now-route-matches header to an SSR endpoint

A little debugging reveals that when the header is added to our request to the SSR page: `/poc`, the `fn` function is triggered twice, with two different `cacheKeys`:

- `/poc-0` -> page initially requested
- `/_error-0` -> error page was requested due to confusion caused by the header; although the SSR page may appear to be an SSG page because of the header, its nature remains unchanged, and it therefore lacks revalidation, which causes Next.js to throw an error in *src/server/base-server.ts*:

```
if (cacheEntry.revalidate < 1) {
  throw new Error(`Invalid revalidate configuration provided: ${cacheEntry.revalidate} < 1`);
}
```

The initial exploit for CVE-2024-46982 combined the `x-now-route-matches` header for the SSG and the `__nextDataReq` URL parameter to make it a data request. Now (*after the fix*), by sending a request combining these two elements, we still get a `500` status code, but with a different body/content-type this time:

Another look at the debugger, the `fn` function is now called 3 times, with the same two `cacheKeys` as before:

- `/poc-0` -> page initially requested
- `/_error-0` -> error page was requested due to confusion caused by the header, *as seen earlier*
- `/_error-0` -> a second error, this time caused by the addition of the `__nextDataReq` parameter which raised an error in stream handling within the pipe-readable.js file

The second error led to a fork in code execution, landing in a base-server.ts catch block, which resulted in the body being rewritten:

[image: image]

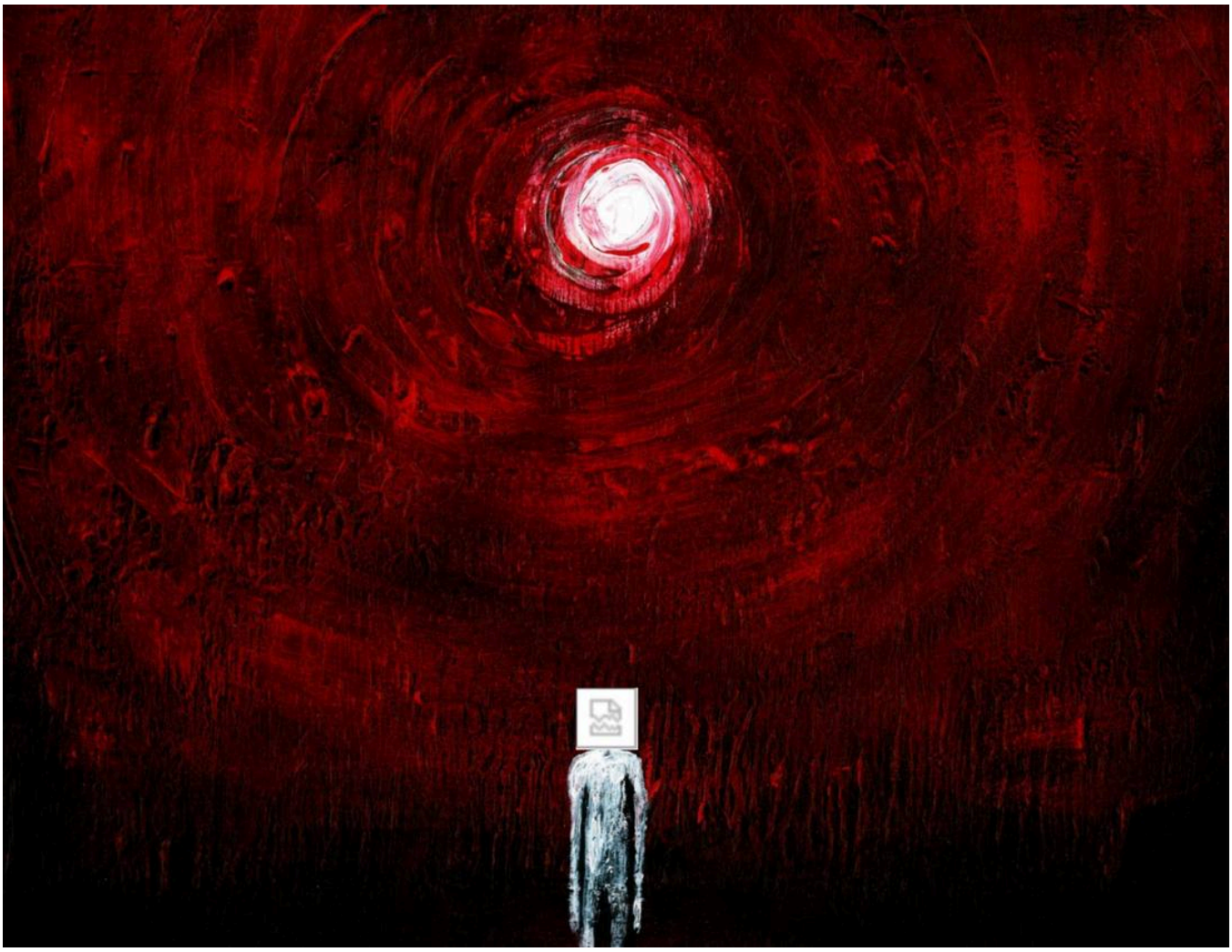
What's interesting is that despite the error, the HTML prepared by the `fn` function for this call - *the second error* - was indeed the `pageProps`, it was just rewritten a little further down the execution pipeline. Check this with my super `console.log` (*yea, long live console.log*):

```
batcher -> CACHE KEY /_error-0 ; RESOLVED :  {
  value: {
    kind: 'PAGES',
    html: '{"pageProps":{"statusCode":500}}',
    pageData: undefined,
    headers: {},
    status: undefined
  },
  revalidate: undefined,
  isFallback: false,
  isMiss: true
}
```

Let's pause for a moment and list the ingredients:

- One request containing the HTML of the `/_error` page in its body, with the actual cacheKey: `/_error-0`
- A second request containing the much-desired `pageProps` in its body, having been overwritten further down the execution pipeline with "**InternalServerError**", also using the cacheKey: `/_error-0`
- A batcher acting as a promise anti-duplication mechanism based on the `cacheKey`

Race Condition - It's eclipse time



By sending **both requests simultaneously**, we can **hijack the promise anti-duplication mechanism** to get the `pageProps` in a completely altered `text/html` response:

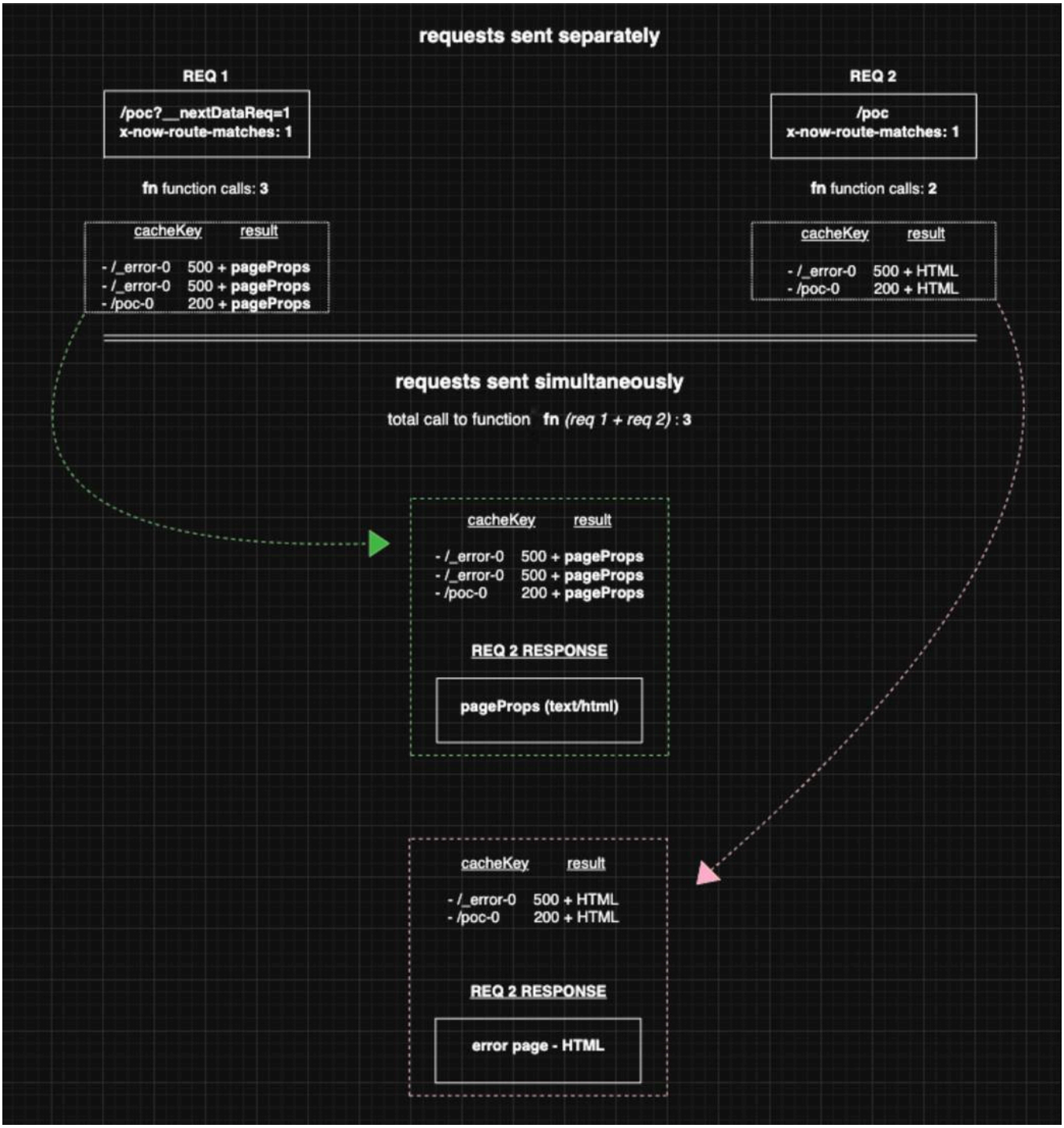
pageProps rebirth!

This is possible because the calls generated by the first request - *containing header + parameter* - share the same `cacheKey` as those triggered by the second (`/_error-0`). The batcher will therefore **only create promises for the initial calls**, and any subsequent ones arriving during their execution window will not trigger new promises, they will simply **receive the result of the initial requests once resolved**.

Let's ignore the calls using `/poc-0` as a `cacheKey` ; they're not relevant in our case since an error is thrown later in the execution, preventing their results from being rendered. We therefore have two calls with `/error-0` as a `cacheKey` generated by request 1 (*header + parameter*), and one call with `/error-0` as a cacheKey generated by request 2 (*header only*).

Regardless of the result, the response from request 1 will remain unchanged, since -as we saw earlier- its body is rewritten later in the execution pipeline. However, the response from request 2 has two possible outcomes (*when both requests are sent simultaneously*):

- The display of `pageProps` with `text/html` as a content-type, **fulfilling the initial objective**. This occurs if the calls from request 1 reach the batcher first
- The display of a 500 error page (HTML). This occurs if the calls from request 2 reach the batcher first



When the calls from request 1 reach the batcher first, this makes it possible to retrieve the result of the `fn` function **initially "hidden" by the body overwrite** (*caused by the second error as seen previously*) and share it with the response of request 2. The latter will display it without issue, **since no body overwrite is performed**: request 2 doesn't include the problematic URL parameter that originally caused the (*second*) error.

It's worth noting that, since the cacheKey is `/_error-0` - and not the path of the originally requested page - the pages called don't have to be the same. As long as they trigger an error and therefore result in a call with the `/_error-0` cacheKey, **it's possible to send multiple requests simultaneously to different paths and still reproduce the race-condition**, provided each page throws an error. This works because all the resulting calls share the same `cacheKey`, meaning the promise won't be duplicated and the response will be shared across all of them.

Exotic configurations as pillars of exploitation

I was able to get the `pageProps` back in an altered `text/html` response, which is nice, but in its current state, it's not truly exploitable. Two points:

- The `pageProps` displayed is that of the error page and therefore does not contain any elements of the request being reflected
- The `Cache-Control` is strict: `private, no-cache, no-store, max-age=0, must-revalidate` so the response is non-cacheable

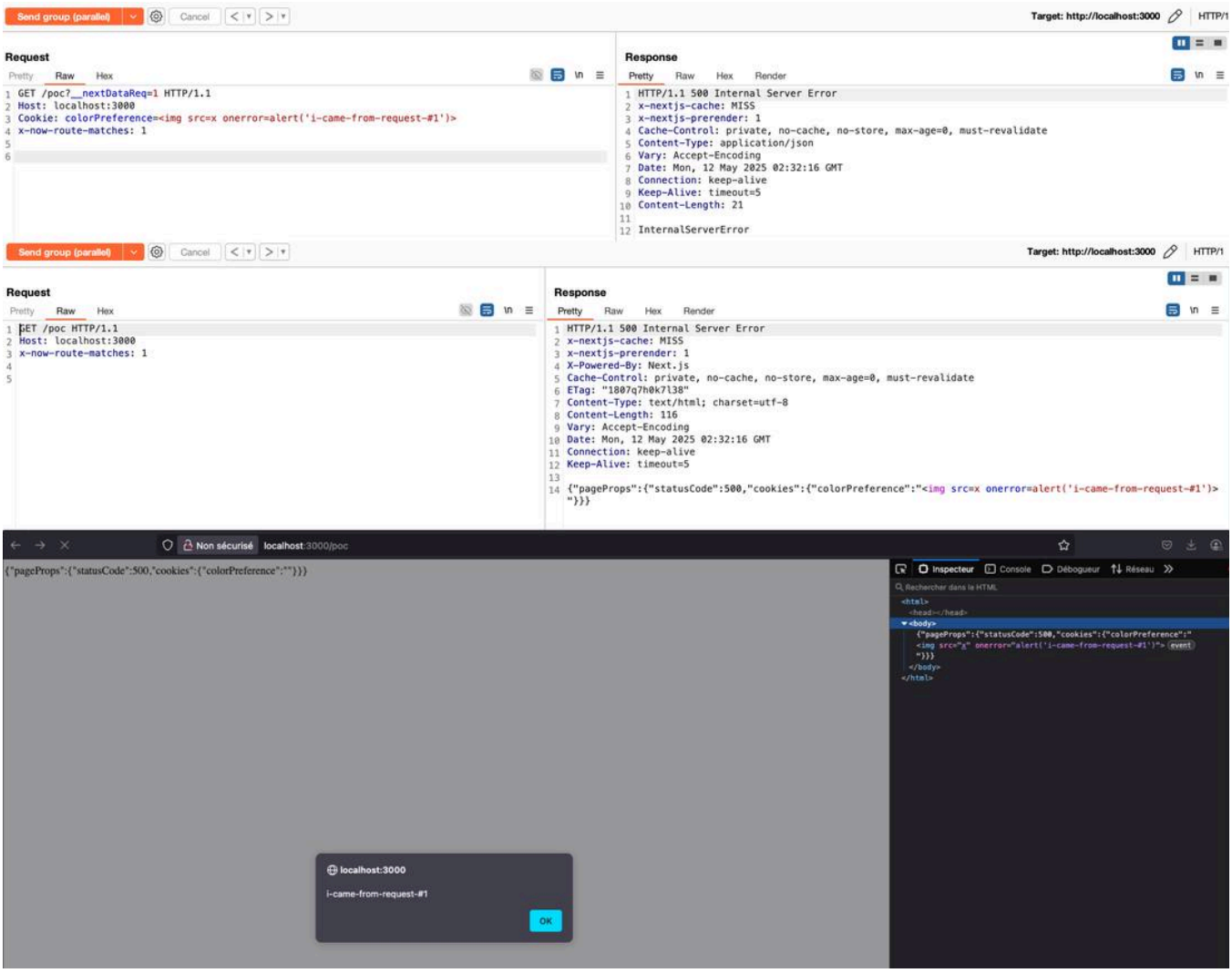
From there, the exploitability of this attack will depend on the development team's ability to deploy poor and/or risky configurations.

For the first point, it's not uncommon to come across applications with enriched `pageProps` on their error pages, whether for monitoring purposes (*like Sentry*) or to preserve user preference context.

Among the ways to achieve this, Next.js has an asynchronous method - a legacy API - called `getInitialProps`, which allows fetching dynamic data before the page is rendered in order to hydrate it. This method can be used in the `_app.tsx` page (*the component within which pages are rendered*) to fetch global data or context that should be available across all pages before the initial render.

Although this pattern is officially documented, it is not recommended.

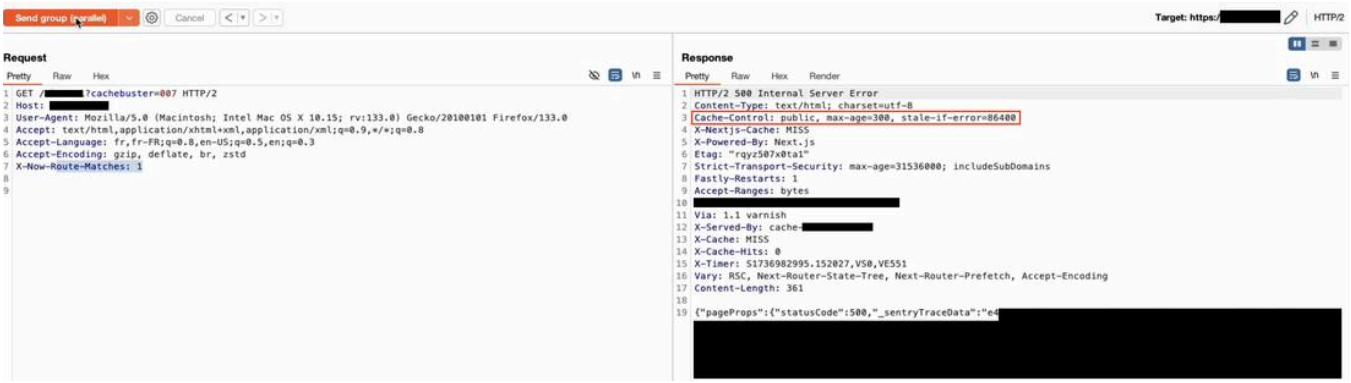
Here is a reproduction of a case encountered on a very large platform, which shared certain properties across all its pages, including static data and a cookie related to user preferences for the application's theme color:



based on a true story

Note that as soon as any element of the request is passed through `getServerSideProps` (`getInitialProps` *for _app.tsx*) - whose original purpose is to provide data that isn't available at build time, like the cookie here - it becomes a potential attack vector, regardless of its initial use on the client side, since we're merely rendering the props in a `text/html` type response.

For the second point, regarding the `cache-control` restriction, it's much rarer but I came across some applications that had configured surprising cache rules, which seemed to override the cache-control coming from the origin when it wasn't a JSON response. I don't know the reason, and to be honest, I didn't really look for it, but oddly enough, all of them used Fastly:



asset of a bug bounty program - race condition to cache-poisoning

When these two points are combined, it becomes possible to **chain the race-condition with a cache-poisoning attack to ultimately bypass the patch**, with the key difference being that the affected caching system is no longer the one provided by the framework.

Although I often used custom (*and dirty*) scripts for my exploits on real targets, I can't close this chapter without a shout-out to James Kettle for his exceptional work on the single-packet attack, particularly the Burp functionality that allows sending a group of parallel requests while neutralizing interference from network jitter making execution much simpler.

Bonus - Another way: never-say-never

There is a second way to partially bypass the fix implemented for CVE-2024-46982 that does not include the "race-condition". When the `x-now-route-matches` header is added to a request to an SSR endpoint, it is considered SSG, as explained earlier. This causes an HTML page to be generated and placed in the `.next/server/pages` build directory. This will happen every time an SSR request is sent with the header, and if it's the same endpoint, the page will simply be overwritten.

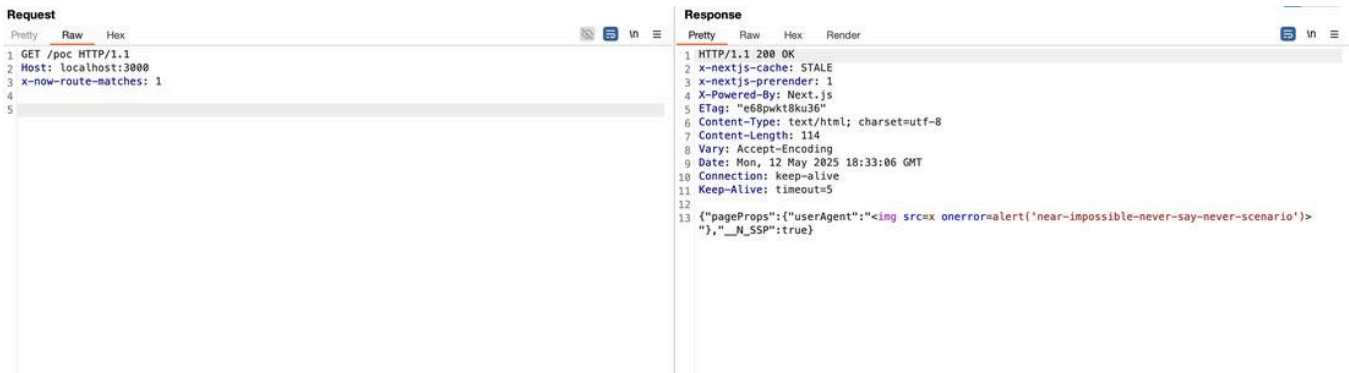
If the server is restarted, the first request to the SSR endpoint in question containing `x-now-route-matches` (*normal calls, without the header, are naturally not concerned*) will cause the static file to be searched in the `.next/server/pages` directory. If it exists, it will be retrieved without the cache's `revalidate` property being set (*not even to 0 as was the case previously*). This will allow the execution pipeline to proceed without the error discussed earlier in this paper being thrown:

[image: image] *src/server/base-server.ts*

This is due to the type check, previously, the value was a number - 0 - and the latter being less than 1 the error was triggered, which is not the case when starting the server, the value being `undefined` consequently bypassing the checks.

The near-impossible, never-say-never scenario is as follows:

- An attacker sends a request to the targeted SSR endpoint containing the `x-now-route-matches` header + the `__nextDataReq` URL parameter + injects the payload (if applicable) into the affected request element
- Returns to hibernate, hoping something happens (*excluding a nextjs update since there's a rebuild*) OR finds a way to bring down the targeted server (*maybe there's a way..?*)
- Finally, send a request to the targeted SSR endpoint with only the `x-now-route-matches` header:



huss

Notice the differences when the exploit is delivered this way, a `200` response (*because no errors*), **without** `cache-control`, which despite the existing constraints to achieve this state, makes it easier to save this pretty response in some CDN's that cache responses without *cache-freshness indicators*. Some examples with Fastly and Cloudflare:

- Fastly :

>

For example, an HTTP 200 (OK) response with no cache-freshness indicators in the response headers is cacheable and will have a TTL of 2 minutes. (documentation)

- Cloudflare :

>

By default, Cloudflare caches certain HTTP response codes with the following Edge Cache TTL when a cache-control directive or expires response header are not present (...) documentation

Security Advisory - CVE-2025-32421

Affected versions : <14.2.24, >15.0.0 and <15.1.6

Patched versions : ≥14.2.24 <15.0.0, ≥15.1.6

https://github.com/vercel/next.js/security/advisories/GHSA-qpjv-v59x-3qc4

Vercel changelog : https://vercel.com/changelog/cve-2025-32421

Conclusion

As a result, bypassing the previous CVE is partially possible via two different techniques, sometimes depending on certain factors beyond the control of Next.js, partly due to poor configuration choices. Despite this, the planets align far more often than one might think, and when they do, the damage can be significant, ranging from a simple DoS to stored XSS via (Race-Condition +) Cache-Poisoning. I've already identified and reported several vulnerable assets through bug bounty programs, a crucial step in funding these research efforts.

Note: Going forward, only research that leads to findings differing in exploitation methods or outcomes from those already published will be shared on this blog (*like this one*), regardless of their severity or impact on the ecosystem -*except under exceptional circumstances*- in order to avoid redundancy.

Thank you for reading.

Al hamduliLlah;

zhero;

Published in May 2025