

← [Back to Blog](#)

Monday, October 23rd 2023

How to Think About Security in Next.js

Posted by



Sebastian Markbåge
@sebmarkbage

React Server Components (RSC) in App Router is a novel paradigm that eliminates much of the redundancy and potential risks linked with conventional methods. Given the newness, developers and subsequently security teams may find it challenging to align their existing security protocols with this model.

This document is meant to highlight a few areas to look out for, what protections are built-in, and include a guide for auditing applications. We focus especially on the risks of accidental data exposure.

Choosing Your Data Handling Model

[React Server Components](#) blur the line between server and client. Data handling is paramount in understanding where information is processed and subsequently made available.

The first thing we need to do is pick what data handling approach is appropriate for our project.

- [HTTP APIs](#) (recommended for existing large projects / orgs)

- [Data Access Layer](#) (recommended for new projects)
- [Component Level Data Access](#) (recommended for prototyping and learning)

We recommend that you stick to one approach and don't mix and match too much. This makes it clear for both developers working in your code base and security auditors for what to expect. Exceptions pop out as suspicious.

HTTP APIs

If you're adopting Server Components in an existing project, the recommended approach is to handle Server Components at runtime as unsafe/untrusted by default like SSR or within the client. So there is no assumption of an internal network or zones of trust and engineers can apply the concept of Zero Trust. Instead, you only call custom API endpoints such as REST or GraphQL using `fetch()` from Server Components just like if it was executing on the client. Passing along any cookies.

If you had existing `getStaticProps` / `getServerSideProps` connecting to a database, you might want to consolidate the model and move these to API end points as well so you have one way to do things.

Look out for any access control that assumes fetches from the internal network are safe.

This approach lets you keep existing organizational structures where existing backend teams, specialized in security can apply existing security practices. If those teams use languages other than JavaScript, that works well in this approach.

It still takes advantage of many of the benefits of Server Components by sending less code to the client and inherent data waterfalls can execute with low latency.

Data Access Layer

Our recommended approach for new projects is to create a separate Data Access Layer inside your JavaScript codebase and consolidate all data access in there. This approach ensures consistent data access and reducing the chance of authorization bugs occurring. It's also easier to maintain given you're consolidating into a single library. Possibly providing better team cohesion with a single programming language. You also get to take advantage of better performance with lower runtime overhead, the ability to share an in-memory cache across different parts of a request.

You build an internal JavaScript library that provides custom data access checks before giving it to the caller. Similar to HTTP endpoints but in the same memory model. Every API should accept the current user and check if the user can see this data before returning it. The principle is that a Server Component function body should only see data that the current user issuing the request is authorized to have access to.

From this point, normal security practices for implementing APIs take over.

TS data/auth.tsx

```
1  import { cache } from 'react';
2  import { cookies } from 'next/headers';
3
4  // Cached helper methods makes it easy to get the same value in many places
5  // without manually passing it around. This discourages passing it from Server
6  // Component to Server Component which minimizes risk of passing it to a Client
7  // Component.
8  export const getCurrentUser = cache(async () => {
9    const token = cookies().get('AUTH_TOKEN');
10    const decodedToken = await decryptAndValidate(token);
11    // Don't include secret tokens or private information as public fields.
12    // Use classes to avoid accidentally passing the whole object to the client.
13    return new User(decodedToken.id);
14  });
```

TS data/user-dto.tsx

```
1  import 'server-only';
2  import { getCurrentUser } from './auth';
3
4  function canSeeUsername(viewer: User) {
5    // Public info for now, but can change
6    return true;
7  }
8
9  function canSeePhoneNumber(viewer: User, team: string) {
10    // Privacy rules
11    return viewer.isAdmin || team === viewer.team;
12  }
13
14  export async function getProfileDTO(slug: string) {
15    // Don't pass values, read back cached values, also solves context and easier t
```

```

16
17 // use a database API that supports safe templating of queries
18 const [rows] = await sql`SELECT * FROM user WHERE slug = ${slug}`;
19 const userData = rows[0];
20
21 const currentUser = await getCurrentUser();
22
23 // only return the data relevant for this query and not everything
24 // <https://www.w3.org/2001/tag/doc/APIMinimization>
25 return {
26   username: canSeeUsername(currentUser) ? userData.username : null,
27   phonenumber: canSeePhoneNumber(currentUser, userData.team)
28     ? userData.phonenumber
29     : null,
30 };
31 }

```

These methods should expose objects that are safe to be transferred to the client as is. We like to call these Data Transfer Objects (DTO) to clarify that they're ready to be consumed by the client.

They might only get consumed by Server Components in practice. This creates a layering where security audits can focus primarily on the Data Access Layer while the UI can rapidly iterate. Smaller surface area and less code to cover makes it easier to catch security issues.

```

1 import {getProfile} from '../..data/user'
2 export async function Page({ params: { slug } }) {
3   // This page can now safely pass around this profile knowing
4   // that it shouldn't contain anything sensitive.
5   const profile = await getProfile(slug);
6   ...
7 }

```

Secret keys can be stored in environment variables but only the data access layer should access `process.env` in this approach.

Component Level Data Access

Another approach is to just put your database queries directly in your Server Components. This approach is only appropriate for rapid iteration and prototyping. E.g. for a small product with a small team where everyone is aware of the risks and how to watch for them.

In this approach you'll want to audit your `"use client"` files carefully. While auditing and reviewing PRs, look at all the exported functions and if the type signature accepts overly broad objects like `user`, or contains props like `token` or `creditCard`. Even privacy sensitive fields like `phoneNumber` need extra scrutiny. A Client Component should not accept more data than the minimal data it needs to perform its job.

```
1 import Profile from './components/profile.tsx';
2
3 export async function Page({ params: { slug } }) {
4   const [rows] = await sql`SELECT * FROM user WHERE slug = ${slug}`;
5   const userData = rows[0];
6   // EXPOSED: This exposes all the fields in userData to the client because
7   // we are passing the data from the Server Component to the Client.
8   // This is similar to returning `userData` in `getServerSideProps`
9   return <Profile user={userData} />;
10 }
```

```
1 'use client';
2 // BAD: This is a bad props interface because it accepts way more data than the
3 // Client Component needs and it encourages server components to pass all that
4 // data down. A better solution would be to accept a limited object with just
5 // the fields necessary for rendering the profile.
6 export default async function Profile({ user }: { user: User }) {
7   return (
8     <div>
9       <h1>{user.name}</h1>
10      ...
11    </div>
12  );
13 }
```

Always use parameterized queries, or a db library that does it for you, to avoid SQL injection attacks.

Server Only

Code that should only ever execute on the server can be marked with:

```
import 'server-only';
```

This will cause the build to error if a Client Component tries to import this module. This can be used to ensure that proprietary/sensitive code or internal business logic doesn't accidentally leak to the client.

The primary way to transfer data is using the React Server Components protocol which happens automatically when passing props to the Client Components. This serialization supports a superset of JSON. Transferring custom classes is not supported and will result in an error.

Therefore, a nice trick to avoid too large objects being accidentally exposed to the client is to use `class` for your data access records.

In the upcoming Next.js 14 release, you can also try out the experimental [React Taint APIs](#) by enable the `taint` flag in `next.config.js`.

JS next.config.js

```
1 module.exports = {
2   experimental: {
3     taint: true,
4   },
5 };
```

This lets you mark an object that should not be allowed to be passed to the client as is.

TS app/data.ts

```
1 import { experimental_taintObjectReference } from 'react';
2
3 export async function getUserData(id) {
4   const data = ...;
5   experimental_taintObjectReference(
6     'Do not pass user data to the client',
7     data
8   );
9   return data;
10 }
```

TS app/page.tsx



```
1 import { getUserData } from './data';
2
3 export async function Page({ searchParams }) {
4   const userData = getUserData(searchParams.id);
5   return <ClientComponent user={userData} />; // error
6 }
```

This does not protect against extracting data fields out of this object and passing them along:

TS app/page.tsx



```
1 export async function Page({ searchParams }) {
2   const { name, phone } = getUserData(searchParams.id);
3   // Intentionally exposing personal data
4   return <ClientComponent name={name} phoneNumber={phone} />;
5 }
```

For unique strings such as tokens, the raw value can be blocked as well using

`taintUniqueValue` [↗](#).

TS app/data.ts



```
1 import { experimental_taintObjectReference, experimental_taintUniqueValue } from
2
3 export async function getUserData(id) {
4   const data = ...;
5   experimental_taintObjectReference(
6     'Do not pass user data to the client',
7     data
8   );
9   experimental_taintUniqueValue(
10    'Do not pass tokens to the client',
11    data,
12    data.token
13  );
14  return data;
15 }
```

However, even this doesn't block derived values.

It's better to avoid data getting into the Server Components in the first place - using a Data Access Layer. Taint checking provides an additional layer of protection against mistakes by specifying the value, please be mindful that functions and classes are already blocked from being passed to Client Components. More layers the minimize risk of something slipping through.

By default, environment variables are only available on the Server. By convention, Next.js also exposes any environment variable prefixed with `NEXT_PUBLIC_` to the client. This lets you expose certain explicit configuration that should be available to the client.

SSR vs RSC

For initial load Next.js will run both the Server Components and the Client Components on the server to produce HTML.

Server Components (RSC) execute in a separate module system from the Client Components to avoid accidentally exposing information between the two modules.

Client Components that render through Server-side Rendering (SSR) should be considered as the same security policy as the browser client. It should not gain access to any privileged data or private APIs. It's highly discouraged to use hacks to try to circumvent this protection (such as stashing data on the global object). The principle is that this code should be able to execute the same on the server as the client. In alignment with secure by default practices, Next.js will fail the build if `server-only` modules are imported from a Client Component.

Read

In Next.js App Router, reading data from a database or API is implemented by rendering Server Component pages.

The input to pages are searchParams in the URL, dynamic params mapped from the URL and

headers. These can be abused to be different values by the Client. They should not be trusted and needs to be re-verified each time they are read. E.g. a `searchParam` should not be used to track things like `?isAdmin=true`. Just because the user is on `/[team]/` doesn't mean that they have access to that team, that needs to be verified when reading data. The principle is to always re-read access control and `cookies()` whenever reading data. Don't pass it as props or params.

Rendering a Server Component should never perform side-effects like mutations. This is not unique to Server Components. React naturally discourages side-effects even when rendering Client Components (outside `useEffect`), by doing things like double-rendering.

Additionally, in Next.js there's no way to set cookies or trigger revalidation of caches during rendering. This also discourages the use of renders for mutations.


E.g. `searchParams` should not be used to perform side-effects like saving changes or logging out. Server Actions should be used for this instead.

This means that the Next.js model never uses GET requests for side-effects when used as intended. This helps avoid a large source of CSRF issues.

Next.js does have support for Custom Route Handlers (`route.tsx`), which can set cookies on GET. It's considered an escape hatch and not part of the general model. These have to explicitly opt-in to accepting GET requests. There's no catch-all handler that might accidentally receive GET requests. If you do decide to create a custom GET handler, these might need extra auditing.

Write

The idiomatic way to perform writes or mutations in Next.js App Router is using [Server Actions](#).

 actions.ts

```
1  'use server';
2
3  export function logout() {
```

```
4     cookies().delete('AUTH_TOKEN');
5 }
```

The `"use server"` annotation exposes an end point that makes all exported functions invocable by the client. The identifier is currently a hash of the source code location. As long as a user gets the handle to the id of an action, it can invoke it with any arguments.

As a result, those functions should always start by validating that the current user is allowed to invoke this action. Functions should also validate the integrity of each argument. This can be done manually or with a tool like `zod`.

 actions.ts

```
1  "use server";
2
3  export async function deletePost(id: number) {
4      if (typeof id !== 'number') {
5          // The TypeScript annotations are not enforced so
6          // we might need to check that the id is what we
7          // think it is.
8          throw new Error();
9      }
10     const user = await getCurrentUser();
11     if (!canDeletePost(user, id)) {
12         throw new Error();
13     }
14     ...
15 }
```

Closures

Server Actions can also be encoded in [closures](#)[↗]. This lets the action be associated with a snapshot of data used at the time of rendering so that you can use this when the action is invoked:

 app/page.tsx

```
1  export default function Page() {
2      const publishVersion = await getLatestVersion();
3      async function publish() {
4          "use server";
```

```

5     if (publishVersion !== await getLatestVersion()) {
6         throw new Error('The version has changed since pressing publish');
7     }
8     ...
9 }
10 return <button action={publish}>Publish</button>;
11 }
12

```

The snapshot of the closure must be sent to the client and back when the server is invoked.

In Next.js 14, the closed over variables are encrypted with the action ID before sent to the client. By default a private key is generated automatically during the build of a Next.js project. Each rebuild generates a new private key which means that each Server Action can only be invoked for a specific build. You might want to use [Skew Protection](#) to ensure that you always invoke the correction version during redeploy.

If you need a key that rotates more frequently or is persistent across multiple builds, you can configure it manually using `NEXT_SERVER_ACTIONS_ENCRYPTION_KEY` environment variable.

By encrypting all closed over variables, you don't accidentally expose any secrets in them. By signing it, it makes it harder for an attacker to mess with the input to the action.

Another alternative to using closures is to use the `.bind(...)` function in JavaScript. **These are NOT encrypted.** This provides an opt-out for performance and is also consistent with `.bind()` on the client.

 app/page.tsx



```

1  async function deletePost(id: number) {
2      "use server";
3      // verify id and that you can still delete it
4      ...
5  }
6
7  export async function Page({ slug }) {
8      const post = await getPost(slug);
9      return <button action={deletePost.bind(null, post.id)}>
10         Delete
11     </button>;
12 }

```

The principle is that the argument list to Server Actions (`"use server"`) must always be treated as hostile and the input has to be verified.

CSRF

All Server Actions can be invoked by plain `<form>`, which could open them up to CSRF attacks. Behind the scenes, Server Actions are always implemented using POST and only this HTTP method is allowed to invoke them. This alone prevents most CSRF vulnerabilities in modern browsers, particularly due to Same-Site cookies being the default.

As an additional protection Server Actions in Next.js 14 also compares the `origin` header to the `Host` header (or `X-Forwarded-Host`). If they don't match, the Action will be rejected. In other words, Server Actions can only be invoked on the same host as the page that hosts it. Very old unsupported and outdated browsers that don't support the `origin` header could be at risk.

Server Actions doesn't use CSRF tokens, therefore HTML sanitization is crucial.

When Custom Route Handlers (`route.tsx`) are used instead, extra auditing can be necessary since CSRF protection has to be done manually there. The traditional rules apply there.

Error Handling

Bugs happen. When errors are thrown on the Server they are eventually rethrown in Client code to be handled in the UI. The error messages and stack traces might end up containing sensitive information. E.g. `[credit card number] is not a valid phone number`.

In production mode, React doesn't emit errors or rejected promises to the client. Instead a hash is sent representing the error. This hash can be used to associate multiple of the same errors together and associate the error with server logs. React replaces the error message with its own generic one.

In development mode, server errors are still sent in plain text to the client to help with debugging.

It's important to always run in Next.js in production mode for production workloads. Development mode does not optimize for security and performance.

Custom Routes and Middleware

[Custom Route Handlers](#) and [Middleware](#) are considered low level escape hatches for features that cannot be implemented using any other built-in functionality. This also opens up potential footguns that the framework otherwise protects against. With great power comes great responsibility.

As mentioned above, `route.tsx` routes can implement custom GET and POST handlers which may suffer from CSRF issues if not done correctly.

Middleware can be used to limit access to certain pages. Usually it's best to do this with an allow list rather than a deny list. That's because it can be tricky to know all the different ways there is to get access to data, such as if there's a rewrite or client request.

For example, it's common to only think about the HTML page. Next.js also supports client navigation that can load RSC/JSON payloads. In Pages Router, this used to be in a custom URL.

To make writing matchers easier Next.js App Router always uses the page's plain URL for both initial HTML, client navigations and Server Actions. Client navigations use `?_rsc=...` search param as a cache breaker.

Server Actions live on the page they're used on and as such inherit the same access control. If Middleware allows reading a page, you can also invoke actions on that page. To limit access to Server Actions on a page, you can ban the POST HTTP method on that page.

Audit

If you're doing an audit of a Next.js App Router project here are a few things we recommend looking extra at:

- **Data Access Layer.** Is there an established practice for an isolated Data Access Layer? Verify that database packages and environment variables are not imported outside the Data Access Layer.
- `"use client"` files. Are the Component props expecting private data? Are the type signatures overly broad?
- `"use server"` files. Are the Action arguments validated in the action or inside the Data Access Layer? Is the user re-authorized inside the action?
- `/[param]/`. Folders with brackets are user input. Are params validated?
- `middleware.tsx` and `route.tsx` have a lot of power. Spend extra time auditing these using traditional techniques. Perform Penetration Testing or Vulnerability Scanning regularly or in alignment with your team's software development lifecycle.



Resources

[Docs](#)

[Learn](#)

[Showcase](#)

[Blog](#)

[Analytics](#)

[Next.js Conf](#)

[DX Platform](#)

More

[Next.js Commerce](#)

[Contact Sales](#)

[GitHub](#)

[Releases](#)

[Telemetry](#)

[Governance](#)

About Vercel

[Next.js + Vercel](#)

[Open Source
Software](#)

[GitHub](#)

[X](#)

Legal

[Privacy Policy](#)

Subscribe to our newsletter

Stay updated on new releases and features, guides, and case studies.

