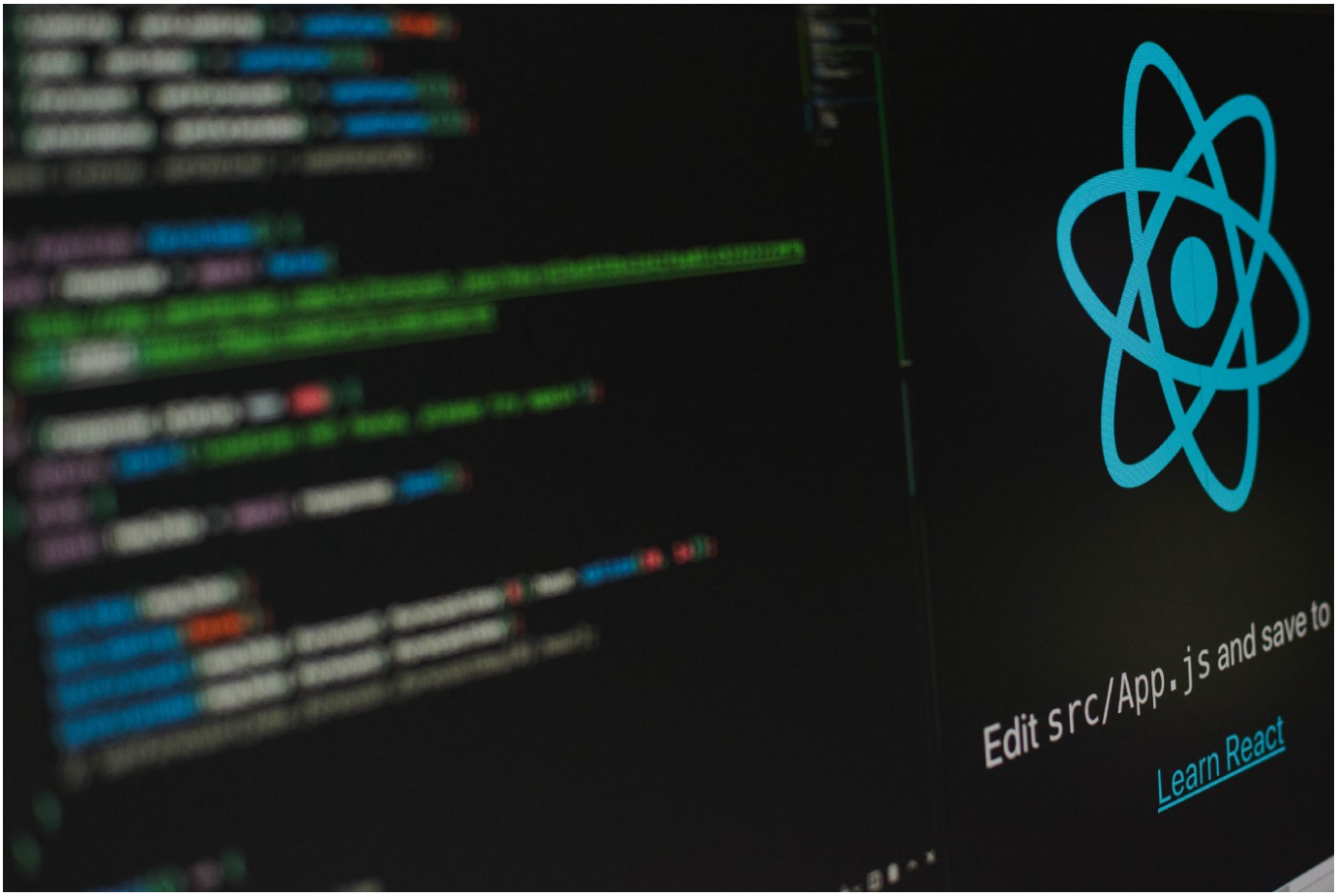
Angular + React: Vulnerability Cheatsheet

By Vickie Li March 1, 2022

Share



Securing applications is not the easiest thing to do. An application has many components: server-side logic, client-side logic, data storage, data transportation, API, and more. With all these components to secure, building a secure application can seem really daunting.

Thankfully, most real-life vulnerabilities share the same root causes. And by studying these common vulnerability types, why they happen, and how to spot them, you can learn to prevent them and secure your application.

The use of every language, framework, or environment exposes the application to a unique set of vulnerabilities. The first step to fixing vulnerabilities in your application is to know what to look for.

Today, let's take a look at six of the most common vulnerabilities that affect
how you can find and prevent them. The vulnerabilities I will cover in this po

- Authentication bypass
- Improper access control
- Open redirects
- Cross-site request forgery (CSRF)
- Template injection
- Cross-site script inclusion (XSSI)

Authentication Bypass

Authentication refers to proving one's identity before executing sensitive actions or accessing sensitive data. If authentication is not implemented correctly on an application, attackers can exploit these misconfigurations to gain access to functionalities they should not be able to.

For instance, routing in Angular is usually done with the `AppRoutingModule`. Before directing users to sensitive routes in the application, you should check whether the user has been authenticated and authorized to access that resource.

```
@NgModule({
    imports: [RouterModule.forRoot([

        // These paths are available to all users.
        { path: '', component: HomeComponent },
        { path: 'features', component: FeaturesComponent },
        { path: 'login', component: LoginComponent },

        // These routes are only available to users after logging in.
        { path: 'feed', component: FeedComponent, canActivate: [ AuthGuard ]},
        { path: 'profile', component: ProfileComponent, canActivate: [ AuthGuard ]},

        // This is the fall-through component when the route is not recognized.
        { path: '**', component: PageNotFoundComponent}

])],

    exports: [RouterModule]

})

export class AppRoutingModule {}
```

For more details about how you can configure authentication properly in Angular and React, read this tutorial.

Improper Access Control

Improper access control occurs anytime when access control in an application is improperly implemented and can be bypassed by an attacker. Authentication bypass issues are essentially a type of improper access control. However, access control comprises of more than authentication. While authentication asks a user to prove their identity: "Who are you?", authorization asks the application "What is this user allowed to do?". Proper authentication and authorization together ensure that users cannot access functionalities outside of their permissions.

There are several ways of configuring authorization for users: role-based a
access control, access control lists, and more. A good post to reference fo

Angular and React is here.

One common mistake that developers make is to perform authorization checks on the client side. This is not secure since client-side checks can be overridden by an attacker. These authorization checks must be performed using server-side code instead:

```
    export class AdministratorGuard implements CanActivate {
      constructor(private authService: AuthService, private router: Router) {}

      canActivate(next: ActivatedRouteSnapshot, state: RouterStateSnapshot):
 Observable<true | UrlTree> {

        // Check whether this user is an administratoor.
        return this.authService.isAdmin().pipe(
          map(isAdmin => {
            if (!isAdmin) {
              return this.router.parseUrl('/')
            }

            return isAdmin
          })
        )
      }
    }

    export class AuthService {
      constructor(private http: HttpClient) {}

      // Whether the user is currently logged in.
      loggedIn: boolean | null = null

      // The user object object encompassing the user's name and role. Will be set
      user: User | null = null

      // Check whether the current user is an administrator.
      isAdmin(): Observable<boolean> {
        return this.getCurrentUser().pipe(map(user => {
          return user != null && user.role === 'admin'
        }))
      }

      // Get the user definition from local state, or the server (if this is the first time
 we are checking).
      getCurrentUser(): Observable<User | null> {
        if (this.loggedIn !== null) {
          return of(this.user)
        }

        return this.http.get<User>('/api/auth', {
          responseType: 'json'
        }).pipe(
          tap({
            next: user => {
              // If we get a user definition from the server i
```

```
logged in.
              this.user     = user
              this.loggedIn = true
            },
            error: error => {
              // A 401 response from the server indicates this user is not logged in.
              this.user     = null
              this.loggedIn = false
            }
          }),
          catchError(() => {
            return of(null)
          })
        )
      }
    }

    export interface User {
        username: string
        role:     string
    }
```

Open Redirects

Websites often need to automatically redirect their users. For example, this scenario happens when unauthenticated users try to access a page that requires logging in. The website will usually redirect those users to the login page, and then return them to their original location after they are authenticated.

During an open-redirect attack, an attacker tricks the user into visiting an external site by providing them with a URL from the legitimate site that redirects somewhere else. This can lead users to believe that they are still on the original site, and help scammers build a more believable phishing campaign.

To prevent open redirects, you need to make sure the application doesn't redirect users to malicious locations. For instance, you can disallow offsite redirects completely by validating redirect URLs:

```
    export class LoginComponent {

      // The username and password entered by the user in the login form.
      username = '';
      password = '';

      // The destination URL to redirect the user to once they log in successfully.
      destinationURL = '/feed'

      constructor(private authService : AuthService,
                  private route       : ActivatedRoute,
                  private router      : Router) { }

      ngOnInit() {
        this.destinationURL = this.route.snapshot.queryParams['destination'] || '/feed';
      }

      onSubmit() {
```

```
        this.authService.login(this.username, this.password)
          .subscribe(
            () => {
              // After the user has lgged in, redirect them to their desired destination.
              let url = this.destinationURL

              // Confirm that the URL is a relative path - i.e. starting with a single '/'
characters.
              if (!url.match(/^\/[^\/\\]/)) {
                url = '/feed'
              }

              this.router.navigate([ url ])
            })
      }
    }
```

There are many other ways of preventing open redirects, like checking the referrer of requests, or using page indexes for redirects. But because it's difficult to validate URLs, open redirects remain a prevalent issue in modern web applications.

Cross-Site Request Forgery

Cross-site request forgery (CSRF) is a client-side technique used to attack other users of a web application. Using CSRF, attackers can send HTTP requests that pretend to come from the victim, carrying out unwanted actions on a victim's behalf. For example, an attacker could change your password or transfer money from your bank account without your permission.

Unlike open redirects, there is a surefire way of preventing CSRF: using a combination of CSRF tokens and SameSite cookies, and avoiding using GET requests for state-changing actions. As an example, Angular allows you to add anti-forgery tokens to HTTP requests using the `HttpClientXsrfModule` module:

```
    @NgModule({
      declarations: [],
      imports: [
        BrowserModule,
        HttpClientModule,
        HttpClientXsrfModule.withOptions({
          cookieName: 'XSRF-TOKEN',
          headerName: 'X-CSRF-TOKEN'
        }),
      ],
      providers: [],
      bootstrap: [AppComponent]
    })
    export class AppModule {
    }
```

Template Injection

Web templates are HTML-like files that provide developers with a way to sp
by combining application data with static templates. This functionality allo

content retrieved from a database or from an HTTP request into web pages.

Template injection refers to injection into web templates. Depending on the permissions of the compromised application, attackers might be able to use the template injection vulnerability to read sensitive files, execute code, or escalate their privileges on the system. For instance, here is an unsafe usage of an Angular template that allows attackers to inject code via URL hashes:

```
@Component({
  selector: 'app-header',
  template: '<h1>' + (window.location.hash || 'Home') + '</h1>'
})
export class HeaderComponent {}
```

You should never directly concatenate user-provided input into templates. Instead, use the template engine's built-in substitution mechanism to safely embed dynamic input:

```
@Component({
  selector: 'app-header',
  template: '<h1>{{ title }}</h1>'
})
export class HeaderComponent {
  title = ''

  ngOnInit() {
    this.title = window.location.hash || 'Home';
  }
}
```

Learn more about how template injection work and how to prevent them in Angular and React in this post.

Cross-Site Script Inclusion

Cross-site script inclusion attacks are also referred to as XSSI. These attacks happen when a malicious site includes Javascript from a victim site to extract sensitive info from the script.

The same-origin policy (SOP) usually controls data access cross-origins. But the SOP does not limit javascript code, and the HTML `<script>` tag is allowed to load Javascript code from any origin. This is an extremely convenient feature that allows JavaScript files to be reused across domains. But this feature also poses a security risk: attackers might be able to steal data written in JavaScript files by loading the JS files of their victims.

For example, imagine that a website stores and transports sensitive data for logged-in users via Javascript files. If a user visits a malicious site in the same browser, the malicious site can import that JavaScript file and gain access to sensitive information associated with the user's session, all thanks to the user's cookies stored in the browser. See an example of this vulnerability and learn how to prevent them in Angular and React here.

To avoid XSSI attacks, don't transport sensitive data within JavaScript files. Here's an example of how to safely load an API token in Angular using JSON files (which are limited by SOP):

```
// The configuration information we will retrieve from the server.
export interface Config {
  username     : string
  accessToken  : string
  role         : string
```

```
}

@Injectable()
export class ConfigService {
  constructor(private http: HttpClient) {}

  // Retrieve configuration information from the server.
  getConfig() {
    return this.http.get<Config>('api/config')
      .pipe(
        catchError(this.handleError)
      )
  }

  private handleError(error: HttpErrorResponse) {
    if (error.status === 0) {
      // A client-side or network error occurred. Handle it accordingly.
      log.error('An error occurred:', error.error)
    } else {
      // The server returned an unsuccessful response code.
      log.error(`Backend returned code ${error.status}, body was: `, error.error)
    }

    return throwError('An unexpected error occurred loading configuration.')
  }
}
```

What other security concepts do you want to learn about? I'd love to know. Feel free to connect on Twitter @vickieli7.

Now that you know how to fix these vulnerabilities, secure your Javascript application by scanning for these vulnerabilities! ShiftLeft CORE can find these vulnerabilities in your application, show you how to fix these bugs, and protect you from Javascript security issues.

Share your email to get the latest ShiftLeft updates, resources, and more.

Enter your email address

Platform

Overview

Next Gen Static Analysis

Intelligent SCA

ShiftLeft Educate

ShiftLeft Illuminate

Pricing

Why ShiftLeft

How it Works

Our Technology

See It For Yourself

Learn

Resource Hub

Blog

Community & Training

Open Source Resources

Documentation

Company

About Us

Press & News

Careers

Contact Us

Hi there, interested in SAST/SCA?