Angular, React and Front-end Security Follow Up

CL-ARF | Virtual classroom | 1 day

Audience: Frontend developers
Preparedness: General JS development
Exercises: Hands-on

This course is the next step for our participants, who completed either our OWASP Top 10, Java Secure Coding or C# Fundamentals course. This is a follow up training, meaning that in order to attend this, everyone must already have the knowledge that is covered in the Fundamentals.

This follow-up course is tailored to participants working as full-stack or frontend developers using Angular and React. The course dives into modern browser security features, as well as framework specific countermeasures and mitigation techniques.

At the end of the training everyone has the possibility to take an exam, where they are able to measure their level of the gained knowledge.

Outline:

- Client-side security
- Modern browser security features
- Introduction to Angular security
- Protection against XSS in Angular
- Protection against HTTP-level vulnerabilities
- Introduction to React security

Participants attending this course will:

- Learn client-side vulnerabilities and secure coding practices
- Understand Content Security Policy
- Explore the security features of Angular
- Understand Angular's countermeasures against XSS
- Understand Angular's countermeasures against HTTP-level vulnerabilities
- Learn about the security of ReactJS
- Understand React's countermeasures against XSS
- Learn about JSON security

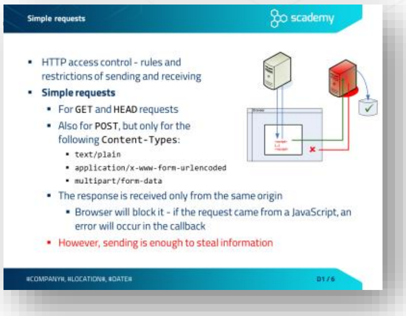
Related courses:

- CL-WSC - Web application security (Onsite / Virtual classroom, 3 days)
- CL-WSM - Web application security master course (Onsite / Virtual classroom, 5 days)
- CL-WTS - Web application security testing (Onsite / Virtual classroom, 3 days)
- CL-NJS - Node.js and Web application security (Onsite / Virtual classroom, 3 days)

Detailed table of contents

Client-side security

- JavaScript security
- Same Origin Policy
- Simple requests ....
- Preflight requests
- JavaScript usage
- JavaScript Global Object
- Dangers of JavaScript
- Clickjacking
  - Exercise – IFrame, Where is My Car?
  - Protection against Clickjacking
  - Anti frame-busting – dismissing protection scripts
  - Protection against busting frame busting
- AJAX security
  - XSS in AJAX
  - Script injection attack in AJAX
  - Exercise – XSS in AJAX
  - XSS protection in AJAX
  - iCloud worm
  - AJAX security guidelines



Simple requests

scademy

- HTTP access control - rules and restrictions of sending and receiving
- **Simple requests**
  - For **GET** and **HEAD** requests
  - Also for **POST**, but only for the following `Content-Types`:
    - `text/plain`
    - `application/x-www-form-urlencoded`
    - `multipart/form-data`
  - The response is received only from the same origin
    - Browser will block it - if the request came from a JavaScript, an error will occur in the callback
  - However, sending is enough to steal information

#COMPANY#, #LOCATION#, #DATE#

D1 / 6

Modern browser security features

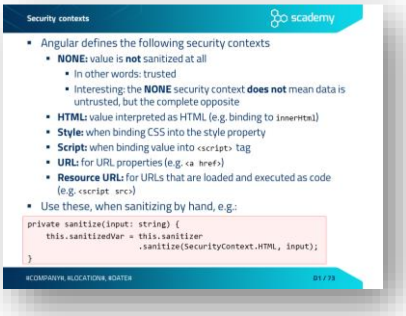
- SameSite Attribute
  - 3rd party cookies
- Certificate Transparency
  - Exercise – HTTP Response Headers
- Content Security Policy
  - Directives
  - Sources
  - Extensions
  - Exercise – CSP in Action

Introduction to Angular security

- Versions of Angular
- Data binding
- Templating
- Built-in security features
- Best practices by Angular

Protection against XSS in Angular

- XSS in a nutshell
- Trusted and untrusted values
  - Inserting values into the DOM
  - Handling of templates
  - AOT template compiler
  - Ahead-of-time compilation
  - Ahead-of-time compilation phases
- Sanitization and security contexts
  - Sanitization
  - Security contexts
  - Exercise: Security Contexts
  - Interacting with the DOM
  - Marking values as trusted
  - Exercise: Marking values as trusted
- Enforcing Trusted Types
  - Configuring HTTP headers
- Server-side XSS protection
  - Server-side template generation



Security contexts

- Angular defines the following security contexts
  - **NONE:** value is **not** sanitized at all
    - In other words: trusted
    - Interesting: the **NONE** security context **does not** mean data is untrusted, but the complete opposite
  - **HTML:** value interpreted as HTML (e.g. binding to `innerHtml`)
  - **Style:** when binding CSS into the style property
  - **Script:** when binding value into `<script>` tag
  - **URL:** for URL properties (e.g. `<a href>`)
  - **Resource URL:** for URLs that are loaded and executed as code (e.g. `<script src>`)
- Use these, when sanitizing by hand, e.g.:

```
private sanitize(input: string) {
    this.sanitizedVar = this.sanitizer
                        .sanitize(SecurityContext.HTML, input);
}
```

#COMPANY#, #LOCATION#, #DATE#

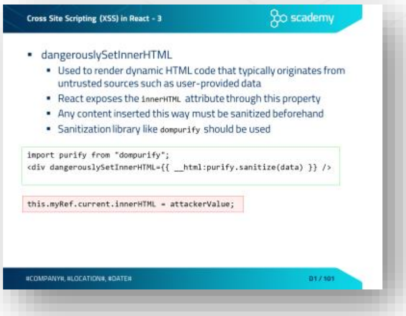
01 / 73

Protection against HTTP-level vulnerabilities

- Cross-site request forgery protection in Angular
- Angular's XSRF protection in practice
- XSSI protection in Angular
  - Cross-site script inclusion protection in Angular
  - Angular's XSSI protection in practice

Introduction to React security

- Introduction to ReactJS
- Protection against XSS in React
    - Cross Site Scripting (XSS) in React - 1
    - Cross Site Scripting (XSS) in React - 2
    - Cross Site Scripting (XSS) in React - 3 ....
    - Cross Site Scripting (XSS) in React - 4
    - Case study – XSS via spoofed JSON element
        - Advanced attack abusing dangerouslySetInnerHTML



Cross Site Scripting (XSS) in React - 3

- dangerouslySetInnerHTML
    - Used to render dynamic HTML code that typically originates from untrusted sources such as user-provided data
    - React exposes the `innerHTML` attribute through this property
    - Any content inserted this way must be sanitized beforehand
    - Sanitization library like `dompurify` should be used

```
import purify from "dompurify";
<div dangerouslySetInnerHTML={{ __html:purify.sanitize(data) }} />
```

```
this.myRef.current.innerHTML = attackerValue;
```

#COMPANY#, #LOCATION#, #DATE#

D1 / 101