Research Document

Table of contents

Contents

Introduction

About the research

The goal of the research is to investigate how a content security policy can be implemented on a ReactJS frontend and the most efficient way possible. The attack surface of the website should be greatly reduced because the policies would only allow certain things that are necessary for the application to work which would also reduce the possibility of a successful cyber-attack.

Approach

The main question to be answered is “How do we implement content security policies?”

First, I would create a list of subquestions which would be helpful in finding an answer. Then, I would answer each subquestion by using different research methodologies. I will use the Library Research methods Best good and bad practices, Community research and Literature study, the Field Research methods Problem analysis and Task analysis to make sure I solve the correct problems and the Workshop method Prototyping to determine if the solutions I have found are actually viable. Then, I will write a conclusion to the findings and recommendations.

Research

First, a number of subquestions should be created:

1. What attacks can be prevented or mitigated with content security policies?
2. Where should the content security policies be placed?
3. How do we specify which sources are allowed and which aren't allowed?
4. What are some techniques to reduce the attack surface of the application as much as possible?
5. Are the content security policies going to cause potential issues for the performance of the application?

Plan

I am planning to use the Library Research methodology to answer to all the subquestions. More specifically, I am going to use the methods Literature Study, Best good and bad practices and Community Research to get more information about the areas that are of interest to me.

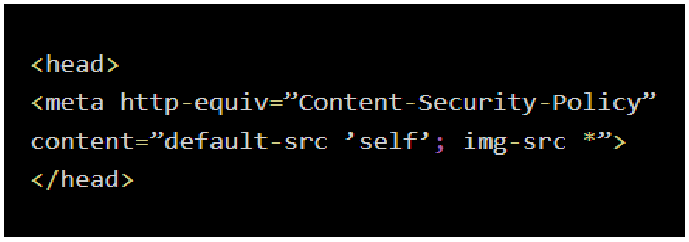
I am also going to use Field Research and its methods, Problem Analysis and Task Analysis to make sure that I understand the problem or task at hand before I try finding solution for it.

Answers

For Frontend Security Policies:

1. ***What attacks can be prevented or mitigated with content security policies:*** It helps to detect and mitigate certain types of attacks, including Cross-Site Scripting (XSS) which exploit the browser's trust of the source of the content, even when it's not coming from where it seems to be coming from. CSP reduces or eliminates the vectors by which XSS can occur by specifying the domains that the browser should consider to be valid sources of executable scripts. The browser will then only execute scripts loaded in source files received from allowed domains, ignoring all other scripts.
2. ***Where should the content security policies be placed:*** If you have access to your web server's configuration via HTTP Response Header. You should add the following line to your configuration file: ***Header always set Content-***

Security-Policy "default-src 'self'; script-src *; style-src *; font-src *;img-src *". For example, if the web server is Apache, the configuration file is ***/etc/apache2/sites-enabled/CSPexample.conf***. If you don’t have access to the configurations of your web server, you should add them to your code via a metatag:



```
<head>
<meta http-equiv=”Content-Security-Policy”
content=”default-src ’self’; img-src *”>
</head>
```

3. ***How do we specify which sources are allowed and which aren’t allowed:*** There are certain rules when writing content security policies.

Some of the directives which specify the allowed sources are:

- ***default-src***: Default policy, used in any case except if overridden by a more precise directive.
- ***connect-src***: Specifies a source with which a connection can be made.
- ***script-src***: Defines authorized sources for scripts
- ***style-src***: Defines authorized sources for stylesheets (CSS)
- ***object-src***: Defines authorized sources for plugins (ex: <object> or <embed>)
- ***img-src***: Defines authorized sources for images, or link element related to an image type (ex: rel=”icon”)
- ***mia-src***: Defines authorized sources for media elements (ex: <video>, <audio>)
- ***frame-src***: Defines authorized sources for loading frames (iframe or frame)
- ***font-src***: Defines authorized sources where fonts files can be loaded from
- ***connect-src***: Policy applies to connections from a XMLHttpRequest (AJAX) or a WebSocket

Common source values for source directives:

- ***: Allows any URL except data: blob: filesystem: schemes.

- ***‘none’***: Prevents loading resources from any source.
- ***‘self’***: Allows loading resources from the same origin (same scheme and domain name).
- ***domain.example.com***: Allows loading resources from the specified domain name.
- ****.example.com***: Allows loading resources from any subdomain under example.com
- ***https://cdn.com***: Allows loading resources only over HTTPS matching the given domain
- ***https:***: Allows loading resources only over HTTPS on any domain.

An example of a content for a security policy would be ***default-src ‘self’*** which means that the default is that the application only accepts sources from itself and everything else would be blocked unless there is another rule which, for example, allows images from a certain domain like ***img-src img.example.com*** which allows images from *img.example.com*.

4. ***What are some techniques to reduce the attack surface of the application as much as possible:*** One technique to reduce the attack surface of the frontend as much as possible is to write ***default-src ‘self’*** as a content security policy which would automatically block anything outside of the application. Then, the developer can look at logs in the developer tools and determine which sources need to be allowed for which service (usually it is written in the warnings). This way the attack surface would be drastically decreased, and the application would be a lot more secure.
5. ***Are the content security policies going to cause potential issues for the performance of the application:*** They can cause issues only if they block some source which the application needs for some of its functionalities. This can be easily fixed by creating another rule which allows the source in question for the functionality that’s using it.

   For example, the application needs to load an image from ***img.example.com*** but the content security policies are blocking it. This can be fixed by adding ***img-src img.example.com*** which allows img.example.com to only be used for loading images and for nothing else.

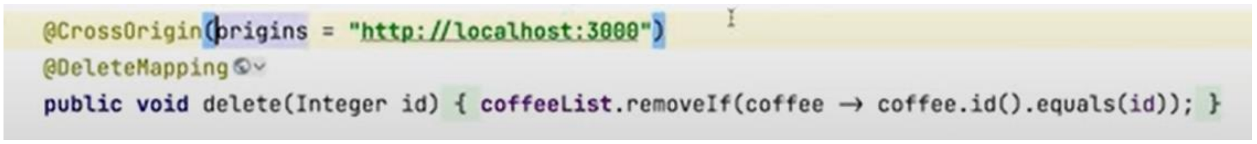
This ensures that the application would work, but the security wouldn't be compromised.

When something doesn't work, it's a good idea to check the developer tools because the warnings usually give tips on how to solve the issues.

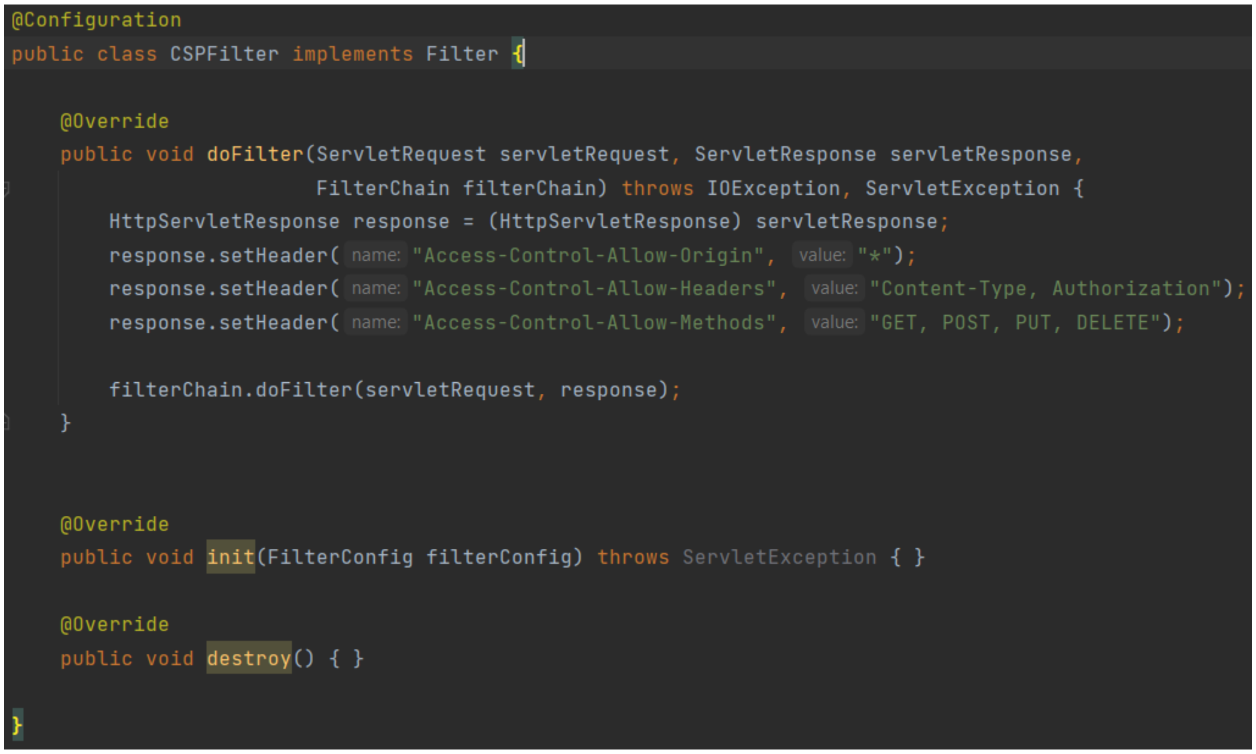
For Backend Security Policies:

1. ***What attacks can be prevented or mitigated with content security policies:*** They can limit the number of vectors through which the backend could be accessed. It can also determine what methods are allowed and what headers are allowed which makes it harder for hackers to access the backend.

   The origins of the frontend could also be determined through the @CrossOrigin tag.

   ```
   @CrossOrigin(origins = "http://localhost:3000")
   @DeleteMapping
   public void delete(Integer id) { coffeeList.removeIf(coffee → coffee.id().equals(id)); }
   ```
2. ***Where should the content security policies be placed:*** A class called CSPFilter which implements Filter. The policies should be written in the doFilter method:



```
@Configuration
public class CSPFilter implements Filter {

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse,
                         FilterChain filterChain) throws IOException, ServletException {
        HttpServletResponse response = (HttpServletResponse) servletResponse;
        response.setHeader( name: "Access-Control-Allow-Origin",  value: "*");
        response.setHeader( name: "Access-Control-Allow-Headers",  value: "Content-Type, Authorization");
        response.setHeader( name: "Access-Control-Allow-Methods",  value: "GET, POST, PUT, DELETE");

        filterChain.doFilter(servletRequest, response);
    }


    @Override
    public void init(FilterConfig filterConfig) throws ServletException { }

    @Override
    public void destroy() { }

}
```

3. ***How do we specify which sources are allowed and which aren't allowed:*** Already answered.
4. ***What are some techniques to reduce the attack surface of the application as much as possible:*** The developer can specify which methods and which headers are allowed when the frontend tries to make connection to the backend. If the request uses a header or a method which isn't allowed, it would be blocked. This can protect the backend from malicious attacks.
5. ***Are the content security policies going to cause potential issues for the performance of the application:*** They can cause issues if they block the source which the frontend needs for its connection to the backend. They could also block some header or method which the frontend uses. This can be easily fixed by creating rules which make sure the source, the methods and the headers are allowed.

Practice

At first, I thought about putting the content security policies on every page of my React application, but then realized this would be redundant and instead decided to put it on only the App page because this way, the policies would encompass all the pages of my application.

I tested to see if it works by first declaring only *"default-src 'self'* without specifying *connect-src* to check if the policies would block the traffic to the backend. It worked because the policies blocked my attempts to log in. Then I added *connect-src* *http://localhost:8080/* to specify that the application needs to allow connections to localhost 8080 and this time my login attempt worked with no issues.

This means that I have implemented the policies for the frontend correctly and I can put more rules in the future if I need.

For the backend, I created a CSPFilter class where I set three headers in the doFilter method - Access-Control-Allow-Origin, Access-Control-Allow-Headers, Access-Control-Allow-Methods. When I tried to put a rule for Access-Control-Allow-Origin, it didn't work, so in the end I decided to leave it as a * sign because, either way, I have @CrossOrigin which specify that only localhost 3000 is allowed. However, I specified that only headers *Content-Type* and *Authorization* are allowed and specified that only methods *GET*, *POST*, *PUT* and *DELETE* are allowed which increases the security and makes it harder for attackers to gain access to the backend layer.

Conclusions

For the frontend, all outside sources should be blocked except for the ones the application needs to function and they should be specified with the policies. The rules should be placed in the configuration file of the web server, or if you don't have access to it, in a metatag in the head of the code.

For the backend, the user can block all other sources by specifying the source which they want to be allowed in @CrossOrigins above the controllers. The content security policies should be placed in CSPFilter class where the user can declare which sources, headers and methods should be allowed from the frontend for extra security.

Recommendations

If there is a problem, the user can always look at the Developer Tools to see the warnings which usually give clues about what rules need to be included to solve the issues.

Bibliography

1. MacLeod, R. (2023, January 9). How to Set Up a Content Security Policy (CSP) in 3 Steps. *Sucuri Blog*. https://blog.sucuri.net/2021/10/how-to-set-up-a-content-security-policy-csp-in-3-steps.html

2. *Content Security Policy (CSP) - HTTP | MDN*. (2023, April 10). https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

3. Web Dev Solutions. (2023, January 4). *Content Security Policy | How to solve content security policy error | #html #vuejs #angular #react* [Video]. YouTube. https://www.youtube.com/watch?v=hUDUqyy0jPM

4. Foundeo Inc. (n.d.). *Content-Security-Policy Java Examples*. https://content-security-policy.com/examples/java/

5. *Shall I use the Content-Security-Policy HTTP header for a backend API?* (n.d.). Stack Overflow. https://stackoverflow.com/questions/45630376/shall-i-use-the-content-security-policy-http-header-for-a-backend-api

6. Dan Vega. (2022, September 27). *Spring Security CORS: How to configure CORS in Spring Boot & Spring Security* [Video]. YouTube. https://www.youtube.com/watch?v=HRwlT_etr60