U.P.B. Sci. Bull., Series C, Vol. 87, Iss. 3, 2025 ISSN 2286-3540

SECURITY CHALLENGES IN MICRO FRONTEND ARCHITECTURE WEB APPLICATIONS

Andrei TĂNASĂ[1], Liliana DOBRICĂ[2]

This paper examines security vulnerabilities inherent to micro frontend architecture, detailing how these weaknesses function and can be exploited within web applications. Our research provides an original analysis of specific attack vectors, demonstrating their operational mechanics and methods of triggering them from a unique attacker's perspective. This work contributes to a deeper understanding of the implications of security flaws and mitigation strategies.

Keywords: security vulnerabilities, micro frontend architecture, software architecture, web application.

1. Introduction

There has been approximately 35 years since Tim Berners-Lee created the first website by joining the hypertext with the Internet, resulting in more than 30 years of intensive web development and innovation in which the industry grew exponentially. Nowadays, when web applications are becoming an important component of the economy serving as an interface to various business-related domains, a better knowledge management in web development organizations could improve collaborative actions between various departments [16]. Similar to how a piece of machinery can become more complex by adding more moving parts, so is the web ecosystem in software development. When a lot of layers are involved and shared across different departments (software architects, developers, DevOps, quality assurance, security engineers), the flaws have more surface to manifest. Moreover, software developers are pressured to deliver quickly web applications, making security an afterthought, especially when tacit or explicit knowledge in security standards is not well-managed. As attack techniques constantly evolve, often one step ahead of defense, actions based on appropriate knowledge about security concerns become an important priority in prevention.

In recent years, web applications development industry started focusing on enhanced scalability, maintainability, and team autonomy, which guided the process towards a revolution. The adoption of the emerged architectural style,

[1] MSc. Eng., Dept. of Automation and Computer Science, National University of Science and Technology POLITEHNICA Bucharest, Romania, e-mail: andrei.tanasa3110@stud.electro.upb.ro
[2] Professor, Dept. of Automation and Computer Science, National University of Science and Technology POLITEHNICA Bucharest, Romania, e-mail: liliana.dobrica@upb.ro

namely, micro frontend (MFE), addresses all these benefits [13][15][19]. However, at the same time with the gain in architectural flexibility and modular nature it raises the probability of experiencing a larger set of security vulnerabilities along the way [14][21]. MFE implies decomposing the application frontend into smaller, independently deployable units. When more elements are involved, the complexity grows and the risk to experiment security issues increases.

The novelty of this software architectural style and the limited attention it has received in the current state-of-the-art motivated our research purpose to investigate its specific security vulnerabilities and demonstrate how these concerns can manifest in an implemented demonstrator of a vulnerable web application. To highlight the effects and push for awareness, a presumed attacker's perspective has been used extensively to expose the weaknesses of the experimental web application. In this article, we first elaborate on the background about the concepts of the MFE architecture. Next, we describe the most common vulnerabilities involving frontend development. Then, we present the design decisions and implementation of the practical MFE web application and we evaluate it considering some of the most common vulnerabilities and discuss about known solutions to mitigate them. Finally, we summarize our experiences in conclusions.

2. Background

2.1. Micro frontend architecture

A recent research study revealed that the earliest notable mentions of MFE architecture had been discussed in blogs and articles by companies' tech teams [20]. The rationale in the emergency of this architectural paradigm was the need for innovation in a context characterized by chaos and complexity in the frontend development of web applications with microservices in the backend.

Over the time software architecture design and analysis methods have facilitated enhanced quality of the software intensive systems in various domains spanning from embedded systems [12] to web applications [10]. Automatic transformation of software architecture models [17] or guided decisions based on catalogues of known architectural styles [10] empower developers to create well-designed applications. MFE architecture style offers a solution for technology independence, code isolation, team integration, preference for native browser APIs and resilience to faults [14]. Web application development with MFEs decomposes the frontend into smaller, independent, and self-contained modules. These modules, called micro frontends (MFEs) themselves, handle specific functionalities within the application. They rely on several key principles to achieve their benefits [19]. These are very similar to microservices' principles, which are separation of concerns, independent development, loose coupling, and self-contained. Separation of concerns ensures that each MFE focuses on a specific, well-defined functionality within the application. Independent development states that MFEs need to be developed, tested, and deployed independently of each other. Loose coupling means that components rely on minimal dependencies and well-defined interfaces to interact with each other. Lastly, self-contained refers to the idea that each module had its own shell, including its own HTML, CSS, and JavaScript code.

While conceptually, MFE architecture can be applied independently of microservices, this is not a feasible option, mainly because it diminishes the benefits of a modular design and goes against its core principles. Using both architectures, each module in frontend would communicate with the counterpart in the backend using application programming interfaces (APIs) and other methods, which brings the idea of a micro application, each half with its own codebase. Every micro application would become a full-stack entity, achieving perfect vertical slicing. Each structure individually developed and maintained is responsible for a specific functionality in the application.

Currently there are few research studies on specific vulnerabilities of applications using MFE paradigm. Existing literature examines benefits, challenges and performance analysis of a scalable architecture based on MFE [13] or have realized a catalog of MFEs anti-patterns [10]. Another quality evaluation at the architectural level using ATAM illustrates scenarios elicitation for a defined quality tree to uncover risks in the design decisions. A performance comparison between the monolithic and micro frontend user interfaces using a browser integrated tool describes specific tests and measurements [14].

2.2. Security challenges involving MFEs architecture development

Web applications with the advancement to MFEs architecture challenges mostly an increased exposure to cross-site scripting attacks and potential misconfigurations of cross-origin resource sharing [14]. Other security weaknesses might include insecure client-side storage, insecure development practices, component isolation issues and cross-site request forgery [3][22].

Cross-site scripting (XSS). Represents a security vulnerability which allows an attacker to inject malicious scripts into a web application, which are then executed in the browsers. These scripts could hijack user session, break websites, or redirect users to malicious sites as it is mentioned in a recent survey [5]. There are few methods the XSS can exploit to happen. The most common way is for the script to be rendered unsanitized (without protection routines) into the document object model (DOM) [18]. The script can be injected in several ways including via direct user input, stored in another module, received via a custom event or from a third-party integration. According to the one of the most used sources for retrieving malicious XSS payloads, the code injected can have different forms, most common being JavaScript, but it can also be HTML, CSS, or others [1]. Vulnerabilities in shared libraries or frameworks can also introduce XSS risks.

The frequency of XSS in applications is still alarmingly high, even with the development of IT industry. A parallel can be drawn between the top 10 security issues [2]. In 2017 and 2021 the XSS was in the ranking, being in a higher spot in the later year. A new standard awareness document for developers and web application security, OWASP Top Ten 2025, is expected to be released soon.

Since MFEs involve multiple independently developed components, the risk of XSS is heightened due to inconsistent security practices across teams. One compromised MFE could impact others, leading to a cascading effect.

Cross-origin resource sharing (CORS) misconfiguration. CORS is essentially a security feature part of web browsers that controls how resources on one side can be accessed by scripts from another. When these policies are misconfigured headers, it can expose sensitive APIs or promote malicious interactions between modules. These misconfigurations happen when developers use overly permissive settings, without verifying where the requests come from, as well as improper credentials handling. An attacker could exploit these weaknesses to access protected resources, cause a data leak in the application or a way for an attacker to overload the system.

Compared to monolithic structures, MFEs are deployed on different subdomains or domains and communicate with backend services, relying on those systems to receive information. This functionality introduces the necessity for CORS policies that allow safe communication between the elements involved.

Insecure client-side storage. Client-side storage refers to ways of depositing data locally on the user's device via web browsers. These include local storage, session storage, indexed databases and cookies. In OWASP top 10 client-side security risks, the use of client-side storage for any type of sensitive data is a high-risk practice [4]. Session management flaws are prevalent vulnerability [7].

In MFE architecture, each module can independently decide how to handle and store sensitive information like user preferences and authentication tokens. Compared to a monolithic approach, the chances for one module to use problematic storage options are higher, as more elements are involved. The problem with client-side storage comes from how easily it's accessible. When insecure client-side methods of storing data like local storage or session storage are used, the data is exposed to JavaScript and unauthorized access made by another MFE. If data gets extracted it can directly harm the user (for example, impersonating the user using authentication token) or it can affect the application by overridden or leaking data due to naming conflicts or global access.

Component isolation issues. Component isolation refers to several types of isolation. While not all can pose a security "hazard", it's important that each aspect is taken into consideration in MFE development. There are multiple types of isolation including runtime isolation. This refers mainly to the logical separation between MFEs when code is executed. Without it, a compromised component could affect the behavior or integrity of other independently developed modules. By default, CSS styling is global in browsers, so careless frontend development may unintentionally target multiple components, creating inconsistencies. Each component maintains its own internal states and should not be allowed to share them or manipulate others directly. And lastly, security isolation should enforce tight access controls and privilege boundaries to prevent MFEs becoming out of scope. Failing this could lead to data leaks and compromised logic.

Component isolation is a foundational security concern for MFE architecture. In monolithic frontends, the application is handled and deployed as a single unit. In the opposite corner, the MFEs are fragmented and, most of the time, displayed side-by-side within the same DOM context. Without proper isolation one module might access or interfere with another's internal logic or event listeners. Without strict boundaries, the system is exposed to cross-module attacks, unauthorized data access or privilege escalation.

Dependency management. According to Black Duck's annual report [8], which analyzed 965 commercial codebases within 16 industries, 86% of them had open source software vulnerabilities, while 81% included at least one high or critical risk vulnerability. In a similar note, in 2022, the Snyk (a leader in developer security) and The Linux Foundation (a global nonprofit organization) reported in [9] that the average development project has 49 vulnerabilities spanning 80 direct dependencies, and the time it takes to fix something in an open source project has increased from 49 days in 2018 to 110 days in 2021.

In the MFE architecture, where different teams independently develop and deploy MFEs, managing dependencies can become a complex task. Each MFE may use separate versions of the same dependencies, leading to security vulnerabilities and performance issues if outdated libraries are used. Some older versions of libraries may contain known security flaws, while others may experience version conflicts that result in unexpected behavior. Unverified third-party code can also introduce unknown risks. By comparison, in a monolithic application these types of conflicts are not a concern, as all files use the same dependencies, but security risks can still occur. The development of software has improved thanks to the inclusion of open source packages and libraries that developers can use. In order to mitigate any of the risks that may appear, a dependency management solution must be applied to any codebase. Module Federation with shared dependencies with version alignment is a good start for a safe MFE implementation (or a centralized dependency management), followed by regular security audits using tools like npm audit or Snyk and enforcing whitelisting of approved libraries are key strategies to mitigate these risks, advised in [10].

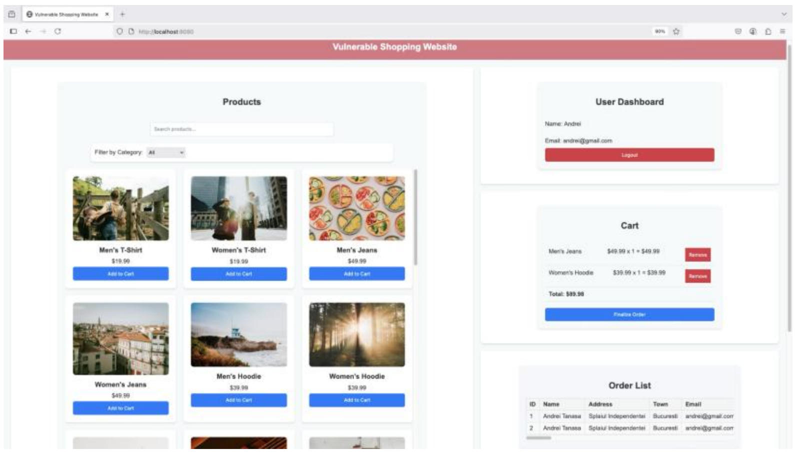
Cross-site request forgery (CSRF). This is a web vulnerability that causes authenticated users to perform certain actions on a website after being tricked by the attacker. This issue is more complex in MFE architecture, because multiple components interact with the same authentication system [11].

The attack has a simple structure; the user must first be logged in the web application. The attacker, using a malicious website or link makes the user send a state-changing HTTP request (this can be a POST, PUT or DELETE) to the target application. Because the domain is trusted, the request automatically gets credentials from the browser and ends up in the hand of the attacker.

This can be caused by different factors, for example if the authentication tokens are stored in the browser's cookies and are sent automatically in requests across components, or if one MFE exposes a critical functionality without validating the source of the request. As stated in [2], CSRF could be considered a newer vulnerability, appearing in the most recent top 10 risks of OWASP.

3. Design and implementation of the practical MFE architecture application

A generic shopping web application was designed and implemented to provide a testing environment for vulnerabilities. Fig. 1 presents the frontend of the vulnerable application in browser. This follows a horizontal split micro frontend architecture model, where the frontend of the system is divided into several smaller, independently deployable pieces on the same page. These web components are integrated into a container application that serves as the entry point for the user.



Name: Andrei

Email: andrei@gmail.com

Men's Jeans	$49.99 x 1 = $49.99	Remove
Women's Hoodie	$39.99 x 1 = $39.99	Remove

Total: $89.98

ID	Name	Address	Town	Email
1	Andrei Tanasa	Splaiul Independentei	Bucuresti	andrei@gmail.com
2	Andrei Tanasa	Splaiul Independentei	Bucuresti	andrei@gmail.com

Fig. 1. Web application frontend with 4 MFEs in browser.

Each MFE has a counterpart in the backend side of the application, a microservice responsible for the logic behind (Fig. 2). The system consists of four main mini applications (MFE + microservice): *Auth*, *Products*, *Cart*, and *Order*. *Auth* handles user authentication, *Products* manages product listings and interactions including reviews, *Cart* handles user selections, and *Order* manages order submission and display.

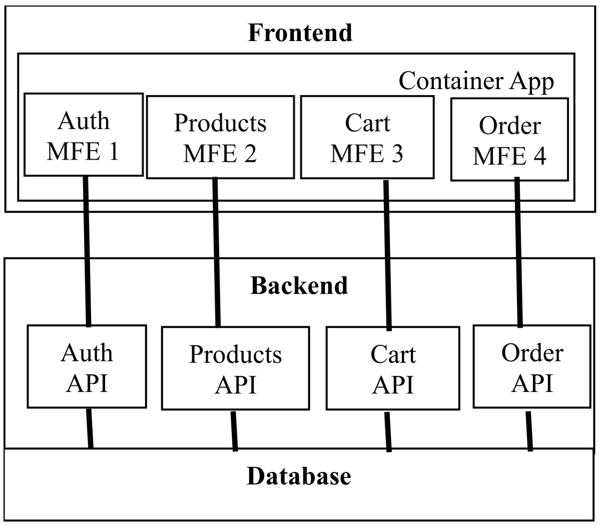


Fig. 2. Frontend with MFEs and backend sides of the web application

To implement MFEs we used the technique of Modular Federation, that is realized in a plugin of Webpack 5. We created each separate module representing each MFE of the web application. The frontend is written in React, a popular JavaScript library for building user interfaces, styled using Sassy CSS (SCSS) files (and later with SCSS modules). The MFEs are connected using Module Federation, each one exposing itself in order to be loaded in the container app (which acts as a shell) but in separate parts of the page. The backend is written in JavaScript using Node.js with Express framework, each one having its own server that handles HTTP requests. SQLite is used for database management, all apps sharing the same database. The authentication status is recognized using a token created using a JavaScript object notation (JSON) web token (JWT) package. The application works as a retail website that sells clothes.

4. Exploiting micro frontends vulnerabilities and mitigation solutions

We investigate in the following attack examples relative to the practical application introduced previously. We show how security vulnerabilities can be exploited by the attacker and we propose mitigation solutions.

4.1. Cross-site scripting attack in practice

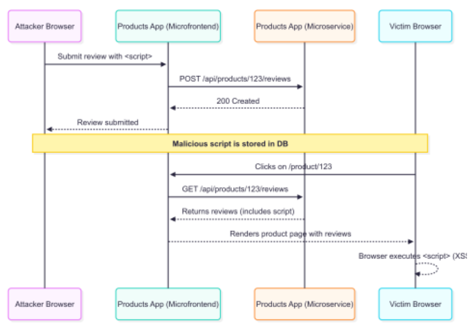
To perform this attack, the attacker will post a review for a product and use the input field to inject the script (Fig. 3a). The code presented in Listing 1 is made using an HTML tag used to display images (<img>), where the script is hidden in one of the tag's attributes (`onerror`), and is executed every time the user loads that specific product page. For a non-expert user, this seems like a broken element in the website. The MFEs application uses React, meaning that by default the code injection is mitigated thanks to how the framework is designed. On the frontend side, the code is rendered using a special HTML tag (dangerouslySetInnerHTML) in order for the script to work in this React application. The attribute is used to bypass the protection and is used frequently to integrate with legacy code (something very common when transitioning from monolith to micro frontend) and to render HTML snippets, as well as raw HTML from external sources. This flaw in the system represents a mistake a developer could have made in a real production environment, and it does not mean the attack can only happen this way. The XSS attack can also happen when JavaScript XML (or JSX) expressions are used, when direct DOM methods are used (like innerHTML or eval), or can even come from third-party libraries.

Listing 1. Example XSS atack

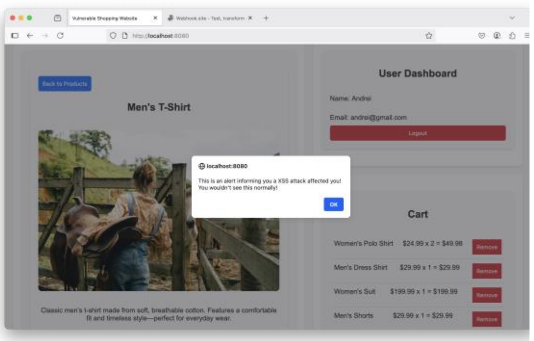
```
<img src=x onerror="
  alert('This is an alert informing you a XSS attack affected you!
You wouldn\'t see this normally!');
  var token = localStorage.getItem('token') || 'none';
  new  Image().src  =  'https://webhook.site/6a24b347-flyh-4324-
b3d4-5eed3bhecb1b?token=' + encodeURIComponent(token);
  setTimeout(() => location.href = 'https://6a216630-8dfc-42a4-
b095-16af81dee7a1-00-2qyugwsdj5cki.picard.replit.dev', 1500);
">
```

The script redirects the user to a fake website hosted using a free platform (Replit in this case) meant to trick the person into giving away login information for the website (the attacker creating an exact replica to the shopping website), counting on the user's lack of experience and attention. If the user is tricked, the credentials will get in the hands of the attacker. For demonstration purposes, the script also triggers an alert on the screen to highlight the successful execution of the script (Fig. 3b). The data entered by the user is collected using a logger application, available online for free, called webhook.site (Fig. 3c).

Mitigation solution. To mitigate this, the developer must be sure that everything that gets rendered in the web application and its source that can be malicious must be sanitized using available tools. Content security policies (CSP) should be enforced to prevent unauthorized script execution, and proper encoding must be implemented before displaying dynamic content (the term used is escape).


Attacker Browser
Products App (Microfrontend)
Products App (Microservice)
Victim Browser
Submit review with <script>
POST /api/products/123/reviews
200 Created
Review submitted
Malicious script is stored in DB
Clicks on /product/123
GET /api/products/123/reviews
Returns reviews (includes script)
Renders product page with reviews
Browser executes <script> (XS
Attacker Browser
Products App (Microfrontend)
Products App (Microservice)
Victim Browser

a) Sequence diagram of an XSS attack



Men's T-Shirt

Name: Andrei

Email: andrei@gmail.com

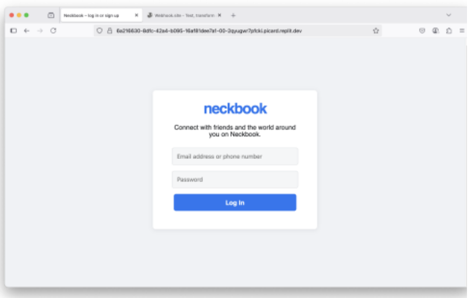
localhost:8080

This is an alert informing you a XSS attack affected you! You wouldn't see this normally!

Women's Polo Shirt	$24.99 x 2 = $49.98
Men's Dress Shirt	$29.99 x 1 = $29.99
Women's Suit	$199.99 x 1 = $199.99
Men's Shorts	$29.99 x 1 = $29.99

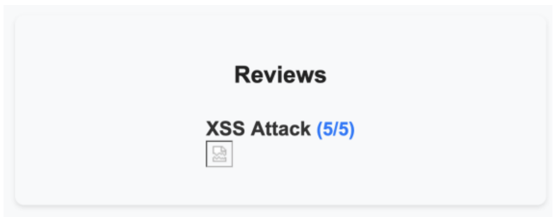
Classic men's t-shirt made from soft, breathable cotton. Features a comfortable fit and timeless style—perfect for everyday wear.

b) The alert of the simulated XSS attack


Neckbook – log in or sign up
Webhook.site - Test, transform
6a216630-8dfc-42a4-b095-16af81dee7a1-00-2qyugwr7pfckl.picard.replit.dev
neckbook
Email address or phone number
Password
Log In

Connect with friends and the world around you on Neckbook.

c) The fake website the user is being redirected to



d) Result of sanitizing the user input (the function removed the script's effect)

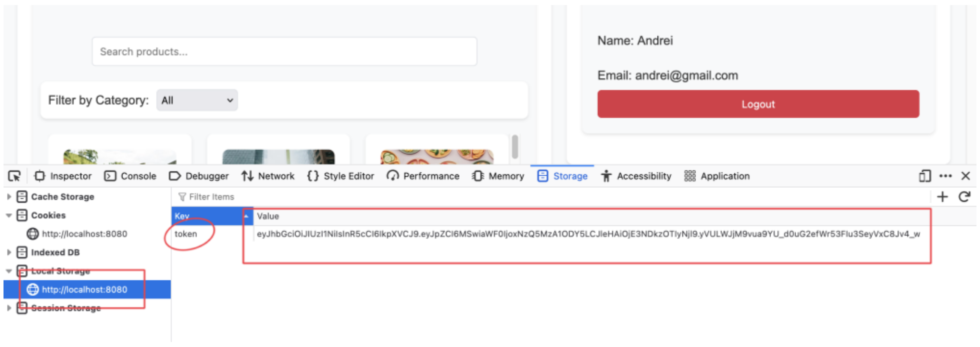
Fig. 3. Detailed example XSS attack in practice and results.

Also, it's important to stick to strict HTML templating, following only the best practices. For the practice application, the JavaScript library called DOMPurify was used to sanitize all the user input that gets rendered in the micro frontends, blocking the effects of the script. A content security policy was also set up to add another layer of protection using Helmet, which is a middleware that sets security headers in Express.js backends. The review ends up displayed as a harmless broken image as seen in Fig. 3d.

4.2. Insecure client-side storage in practice

The MFE application was designed to use a flawed authentication token handling process. In the service responsible for authentication, the user receives a token to acknowledge his successful logging in the application across all other services. The main problem arises when this token is stored in the browser's local storage, making it vulnerable to XSS attacks and malicious browser extensions (Fig. 4). On the browser side, the information can be accessed using the inspect element menu available for each browser. In the storage menu, the token will appear in local storage section. During the XSS attack from previous section, among the data stolen was also the user's authentication token.

Depending on how it was created, the token will expire after some time. For the MFE application, the token is valid for 24 hours, meaning that the attacker can hijack the account as long as it is valid. The method, in this case, involves creating a variable (or changing it if already exists) in the local storage of the browser with the value equal to the stolen token. After a page refresh, the account is hijacked, and for a limited time the attacker has access to every action as the owner.



Key	Value
token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MSwiaWF0IjoxNzQ5MzA1ODY5LCJleHAiOjE3NDkzOTIyNjl9.yVULWJjM9vua9YU_d0uG2efWr53Flu3SeyVxC8Jv4_w

Fig. 4. A view on the authentication token stored in browser's local storage

Mitigation solution. To mitigate the risks of sensitive data being leaked, secure storage solutions should be used, like HTTP-only cookies. It provides security if done correctly, which includes setting up a series of tags when creating the cookies: HttpOnly (blocks JS from accessing the data), Secure (to be sent only via HTTPS) and SameSite (to prevent CSRF attacks discussed later).

Moreover, for authentication, centralized modules that abstract token handling can be used, preventing components from accessing credentials directly. There are third-party options like OAuth2 service which simplify the authentication process and enhance security by leveraging well-established and secure mechanisms. Using a third-party solution with PKCE (proof of ownership) helps to protect against authorization code interception, while enforcing role-based access control to ensure that each MFE can only accesses the data when it needs it.

4.3. Cross-origin resource sharing misconfiguration in practice

By design, the CORS configuration in the backend for the order service was set up to be very simplistic, lacking a polished set of rules to limit unauthorized access. This means an attacker can send POST requests using a third-party scripting tool and affect the application and users in different ways (Fig. 5). The attacker, who has in possession authorization tokens from other users obtained during the XSS attack, could fill the accounts of respective users with fake orders, essentially making the order history feature unusable, for both the user and the store. The same script can be used to overload the system with many requests, crashing the website.

Mitigation solution. To avoid this risk explicit policies on the server-side need to be enforced. All backends involved should only whitelist specific origins (in this case the MFEs) so that any external or unnecessary access is blocked. It is also essential to restrict HTTP methods and headers in cross-origin requests and monitor API activity for any suspicious behavior.

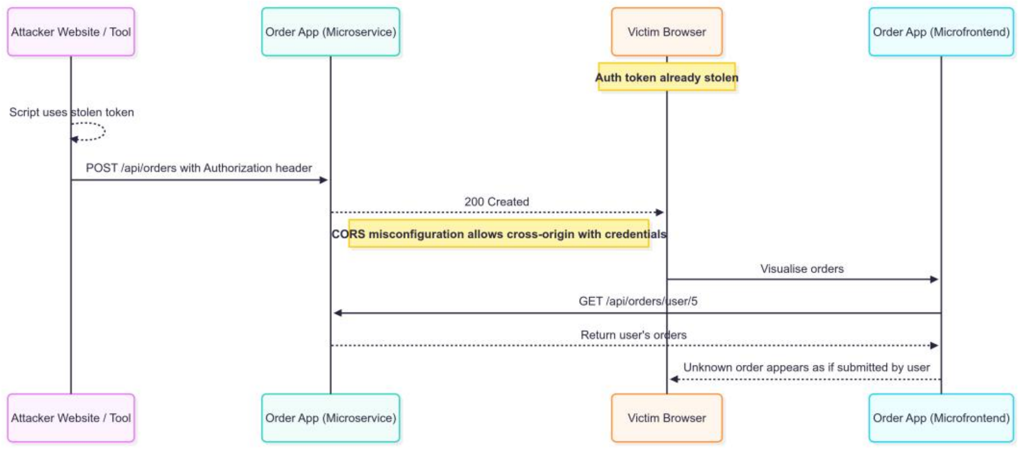

Attacker Website / Tool
Order App (Microservice)
Victim Browser
Order App (Microfrontend)
Auth token already stolen
Script uses stolen token
POST /api/orders with Authorization header
200 Created
CORS misconfiguration allows cross-origin with credentials
Visualise orders
GET /api/orders/user/5
Return user's orders
Unknown order appears as if submitted by user

Fig. 5. Sequence diagram for CORS misconfiguration exploit

4.4. Component isolation vulnerability in practice

When all MFEs are rendered in the same DOM (which acts like a container), one compromised area will affect to the whole application.

Listing 2. Script that affect all the MFEs

```
<img src=x onerror="
  document.body.innerHTML =
    '<h1        style=\'color:red;        text-align:center;\'>UI
disabled</h1>'
">
```

An attacker could use the same CORS misconfiguration or the XSS injection to break the user interface (UI) for the users. In this case, knowing the review section’s input field is not sanitized, the attacker will post another review (in the same way and format like in XSS case) for a different product which will disable the whole interface. This time, the script in Listing 2 will replace the entire body contents of the current HTML page which will affect all the MFEs. This will not allow the users to view that specific product page.

Another lack of isolation, in terms of styling, can be achieved in the MFE application by injecting a different script. The common methods of styling a web application are vulnerable to isolation issues, and our MFE application is no exception. The script which will alter the look of the website for anyone that comes across it is in Listing 3. Similar to other XSS attacks, the code will hide in a product review, and using the `document.style.innerHTML` tag the script will replace all the background colors of the website with light pink, white font color, and blur everything. The `!important` flag in the script signals the browser to apply the styling to all instances that it can. It does not cause permanent effects, if previously the user interface will return to normal after leaving the product page, this time the user must also reload the website to replace the injected styling with the original.

Listing 3. Script that alter the look of the website

```
<img src=x onerror="
  const style = document.createElement('style');
  style.innerHTML = `
    div {
      background: hotpink !important;
      color: white !important;
      filter: blur(4px) !important;
    }
  `;
  document.head.appendChild(style);
">
```

Mitigation solution. Mitigating all the vulnerabilities requires a complex strategy combining more than one concept. For runtime isolation there are explicit technologies designed to encapsulate the modules. Web components technology supports encapsulation using shadow DOM: iframe-based sandboxing offers isolation at browser level by default, and predefined popular frameworks like Single-SPA and Module Federation allow controlled loading of micro frontends. However, the later options do not enforce isolation by default, it requires extra setup (iframes). For styling conflicts there are several options described in [6]. A simple option is to use CSS modules. Sharing application states should be avoided, and instead backend-for-frontend (BFF) APIs should be implemented and used, according to [7]. And lastly, in order to secure micro frontends, trusted registries should be used, alongside integrity hashes when loading the modules.

4.5. Cross-site request forgery mitigation strategies

Mitigating strategies applied to our MFE application require a coordinated effort. Browsers support an attribute for cookies, called `SameSite`, that limits cross-site request transmission. Apps could also use a different type of token, called synchronized token, that gets validated server-side when a request is made, but it should be securely passed between components in the MFEs. Modern frameworks like React and Angular offer build-in support for CSRF protection. A proper CORS configuration can also provide security, as it ensures only trusted origins can interact with backend APIs.

5. Conclusions

This research sets out to explore and analyze MFE architecture specific security concerns, an area that is still under-examined despite the growing adoption of this architectural style. A consolidated background with a classification of security challenges and a demonstrator web application offered valuable insights.

The practical side of this research experiments with a limited number of methods an attacker can exploit a web application with. While the concepts presented have a generalized tone and are applicable to the majority of projects, the tech stack plays an important role in identifying the weak areas in an application.

While MFEs improve autonomy for teams working on different parts of the application and help scale the system better, they introduce complexity in managing security control. Vulnerabilities like XSS, insecure client-side storage, cross-origin resource sharing misconfiguration, component isolation issues, and others, become harder to detect and solve when too many teams are involved.

Many issues come from inconsistent security practices across the people involved in development. One mistake could trigger a domino effect and compromise the entire application, so it is important for all the teams onboard to have the best practices in mind when they handle security.

The decision to adopt micro frontends should be informed not only by scalability and team autonomy goals but also by the maturity of the company's security knowledge. In an environment without proper tooling, monitoring, and a developer security knowledge, the risks of misconfiguration and fragmented security implementation may outweigh the benefits.

Securing a MFE architecture requires more than technical measures, it demands a certain tacit knowledge. Just as DevOps introduced new cultural and operational norms, a secure MFEs development calls for something similar, a role that promotes security-aware design, shares responsibility across teams, and tooling to support modular yet consistent defenses, a "DevSecFrontendOps".

R E F E R E N C E S

[1] ***, OWASP cheat sheet series, Cross site scripting prevention cheat sheet, https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html (accessed July 2025)

[2] ***, OWASP Top 10: 2021, https://owasp.org/Top10/ (accessed July 2025)

[3] C. He, C. Ding, RASP: Next generation web application security protection methodology and framework, Computers & Security vol. 154, 104445, 2025

[4] ***, OWASP Top 10 client-side security risks. Available: https://owasp.org/www-project-top-10-client-side-security-risks/ , 21 February 2025.

[5] *A. Mannouse, S. Yahiouche, M. C. Nait-Hamoud,* Twenty two years since revealing cross-site scripting attacks: A systematic and a comprehensive survey, Computer Science Review, 52, 100634, 2024.

[6] *F. Rappl*, The art of micro frontends: build highly scalable, distributed web applications with multiple teams, Packt Publishing, 2024.

[7] *C. Potter, N. Saxena, S. Maity,* Clarity: Analysing Security in Web Applications, 15th Int. Conf on COMmunication Systems & NETworkS (COMSNETS), pg. 522-528, 2023.

[8] *J Alger*, Open source software vulnerabilities found in 86% of codebases, Security Magazine, 2025. Available: https://www.securitymagazine.com/articles/101420-open-source-software-vulnerabilities-found-in-86-of-codebases (accessed July 2025)

[9] ***, The Linux Foundation, The state of open source security, The Linux Foundation Press Release, 2022.

[10] *N.P.S. da Silva, E. Rodigues, T. Conte*, Catalog of micro frontends anti-patterns, 2025 IEEE/ACM 47th Int. Conf. on Soft. Eng.(ICSE), 2025, DOI 10.1109/ICSE55347.2025.00079.

[11] Cloudflare, Cross-site request forgery (CSRF), Cloudflare Learning Center. Available: https://www.cloudflare.com/en-gb/learning/security/threats/cross-site-request-forgery/

[12] *L. Dobrica, R. Pietraru*, Security Analysis at Architectural Level in Embedded Software Development, CEAI Journal, vol. 11, no. 2, pp. 51-58, 2009

[13] *A. Petcu, M. Frunzete, DA Stoichescu*, Benefits, Challenges, And Performance Analysis of a scalable web architecture based on micro-frontends, U.P.B. Sci. Bull., Series C, 85(3), 2023

[14] *V. Kunstnar, P. Podh*orsky, Micro Frontend Architecture, Zooming Innovation in Customer Technologies Conference, pg. 124-129, IEEE 2024.

[15] *I. Poloskei, Udo Bub*, Enterprise-Level Migration to Micro Frontends in a Multi-Vendor Environment, Acta Polytehnica Hungarica, vol. 18, no. 8, pg. 7-25, 2021

[16] *Dobrica L.*, Knowledge in action towards a better knowledge management in organizations, Management Research and Practice, vol. 13, no. 2, pg. 41-50, 2021

[17] *Dobrica L., Ionita AD, Pietraru R,, Olteanu A.*, Automatic transformation of software architecture models, U.P.B. Sci. Bull., Series C, vol. 73, Iss.3, pg. 3-16, 2011

[18] *D. Klein, T. Barber, S. Bensalim, B. Stock, M. Johns*, Hand Sanitizers in the Wild: A large-scale study of custom Javascript sanitizer functions, 7th European Symposium on Security and Privacy (EuroS&P), IEEE, 2022

[19] *S. Peltonen, L. Mezzalira, D. Taibi*, Motivations, benefits and issues for adopting micro-frontends: a multivocal literature review, Inf. and Software Tehnology, 136, 106571, 2021

[20] *J. Manisto, AP Tuovinen, M. Raatikainen*, Experiences on a frameworkless micro-frontend architecture in a small organization, 20th Int. Conf. on Soft. Arch. Companion, IEEE, 2023

[21] *E. Gashi, D. Hyseni, I. Shabani, B. Cico*, The advantages of micro-frontend architecture for developing web application, 13th Mediteranean Conf on Embedded Computing, 2024

[22] *S. Alazmi, D.C. de Leon*, A systematic literature review on the characteristics and effectiveness of web application vulnerability scanners, IEEE Access, vol. 10, 33200-33219, 2022.