SECURE CODING REVIEW

By Ahmad Rahman

TABLE OF CONTENTS

INTRODUCTION

What is Secure Coding Review?

Secure code review involves meticulously examining the source code of an application to detect and rectify any potential vulnerabilities introduced during its development phase. This review can be conducted either manually by analysing each line of code, or through automated tools. I am conducting a manual review.

Why is it Important?

Secure code review is essential for ensuring the integrity of software systems by proactively identifying and mitigating security vulnerabilities. By checking the code, it helps safeguard against potential exploits and breaches, enhancing overall application security.

What is the aim?

Through manual examination of source code, we aim to identify and rectify potential vulnerabilities, ensuring the integrity and security of our applications. By continuously integrating secure coding practices throughout the software development lifecycle, we strengthen our defences against exploits and breaches, enhancing overall application security.

SCOPE & Purpose

We will be conducting the review on a couple .js files from the front end development of the Chameleon Website. The file I have selected are chatbot.js.

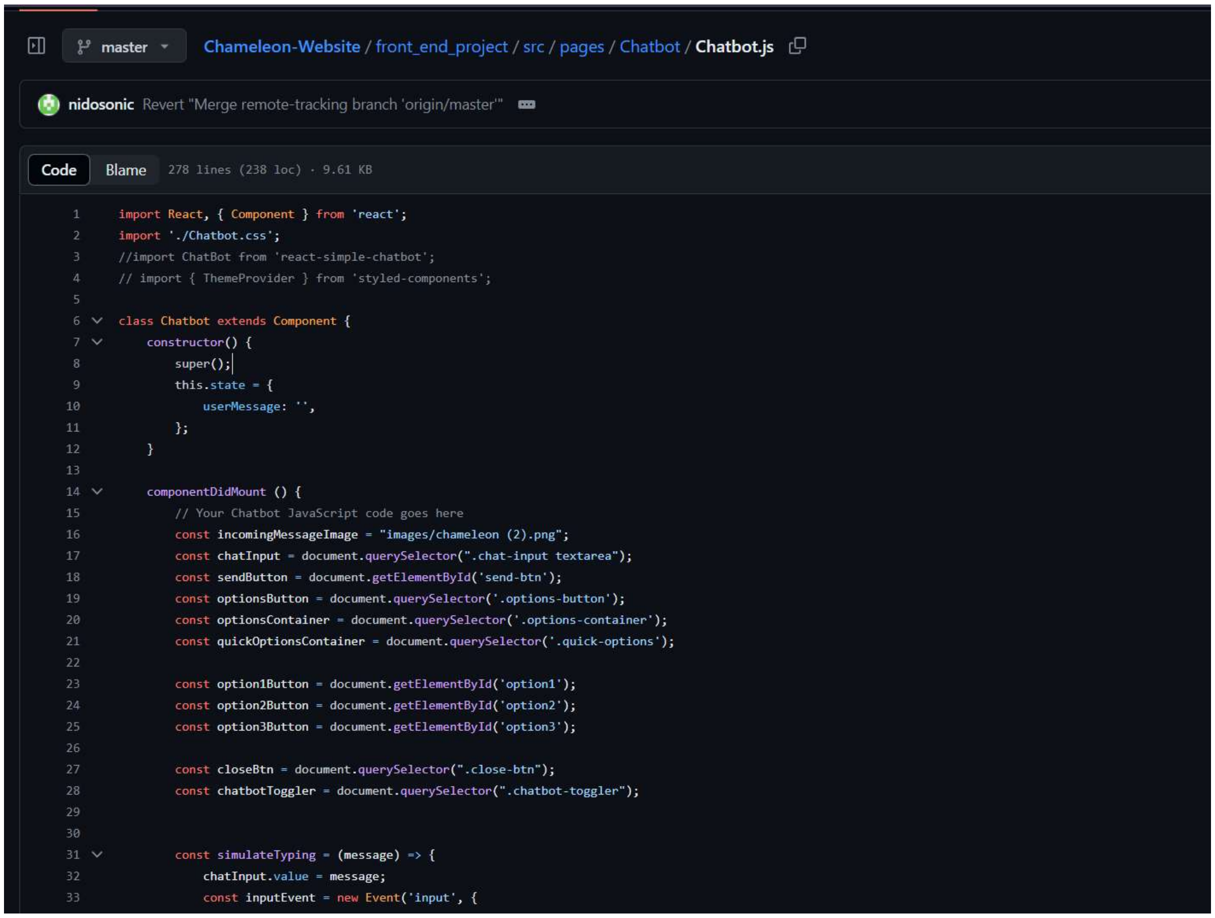
Purpose:

Chatbot.js file appears to implement the client-side functionality for a chatbot feature within the web application. Given its potential interaction with users and sensitive data, ensuring the security of this component is paramount to prevent potential vulnerabilities. It is an important page for this website. A secure code review on the chatbot page can set an idea for future secure coding practices.

Based on these reviews and findings, the development team can take note for all pages and fix vulnerabilities.

Below are the screenshots for a **part** of the code I am reviewing. The entire code is too large to be shown in screenshots in this document. The codes can be founded in the Github at:

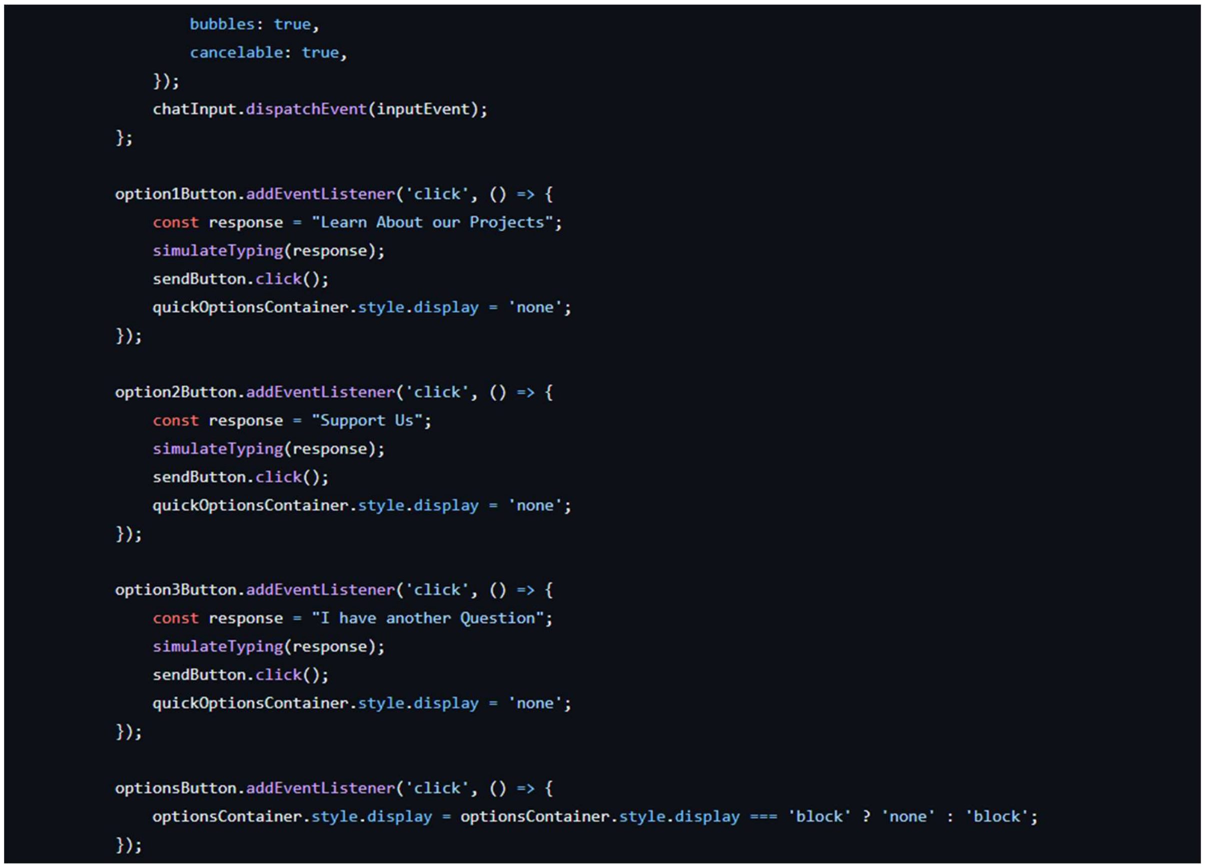
Chameleon-Website/front_end_project/src/pages/


master
Chameleon-Website / front_end_project / src / pages / Chatbot / Chatbot.js
Code
Blame

nidosonic Revert "Merge remote-tracking branch 'origin/master'"

278 lines (238 loc) · 9.61 KB

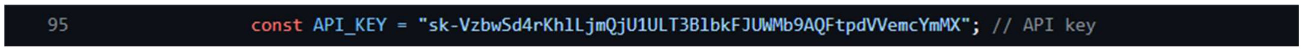
```javascript
import React, { Component } from 'react';
import './Chatbot.css';
//import ChatBot from 'react-simple-chatbot';
// import { ThemeProvider } from 'styled-components';

class Chatbot extends Component {
    constructor() {
        super();
        this.state = {
            userMessage: '',
        };
    }

    componentDidMount () {
        // Your Chatbot JavaScript code goes here
        const incomingMessageImage = "images/chameleon (2).png";
        const chatInput = document.querySelector(".chat-input textarea");
        const sendButton = document.getElementById('send-btn');
        const optionsButton = document.querySelector('.options-button');
        const optionsContainer = document.querySelector('.options-container');
        const quickOptionsContainer = document.querySelector('.quick-options');

        const option1Button = document.getElementById('option1');
        const option2Button = document.getElementById('option2');
        const option3Button = document.getElementById('option3');

        const closeBtn = document.querySelector(".close-btn");
        const chatbotToggler = document.querySelector(".chatbot-toggler");


        const simulateTyping = (message) => {
            chatInput.value = message;
            const inputEvent = new Event('input', {
```



```javascript
                bubbles: true,
                cancelable: true,
            });
            chatInput.dispatchEvent(inputEvent);
        };

        option1Button.addEventListener('click', () => {
            const response = "Learn About our Projects";
            simulateTyping(response);
            sendButton.click();
            quickOptionsContainer.style.display = 'none';
        });

        option2Button.addEventListener('click', () => {
            const response = "Support Us";
            simulateTyping(response);
            sendButton.click();
            quickOptionsContainer.style.display = 'none';
        });

        option3Button.addEventListener('click', () => {
            const response = "I have another Question";
            simulateTyping(response);
            sendButton.click();
            quickOptionsContainer.style.display = 'none';
        });

        optionsButton.addEventListener('click', () => {
            optionsContainer.style.display = optionsContainer.style.display === 'block' ? 'none' : 'block';
        });
```

Findings & Proposed Solutions

Starting off with the chatbot.js file. I have found some secure coding vulnerabilities which may need to be looked at.



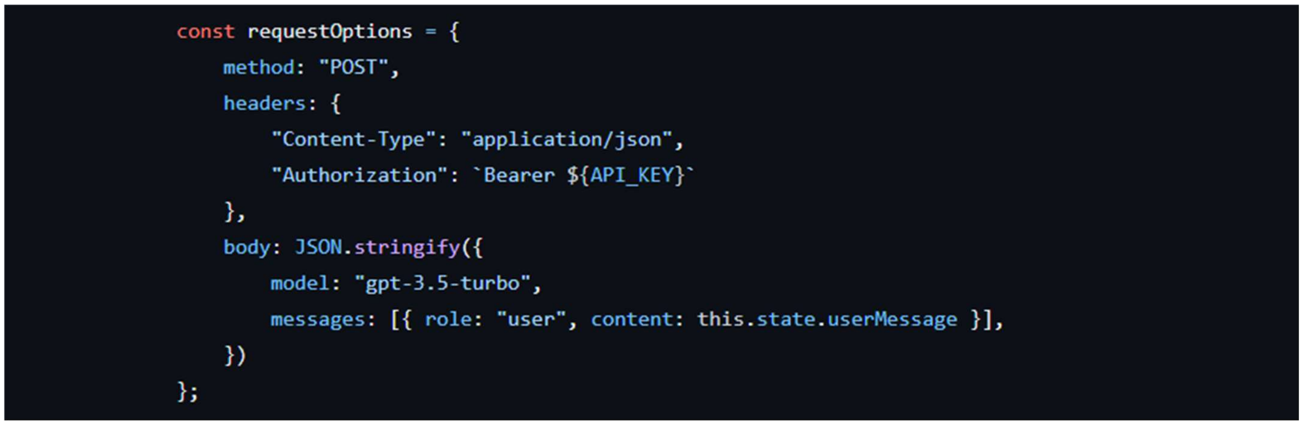
```
const API_KEY = "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"; // API key
```

Hard-coded API. Storing API keys directly in client-side code exposes them to anyone who views the source code. It's recommended to keep API keys secure on the server-side and use server-side logic to interact with external APIs.

Store sensitive information such as API keys securely on the server-side.

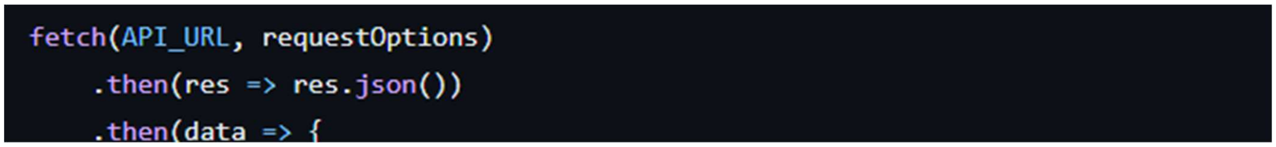
Use environment variables or configuration files to inject API keys into the client-side code securely during deployment.

Implement server-side authentication mechanisms to restrict access to sensitive APIs based on user permissions.



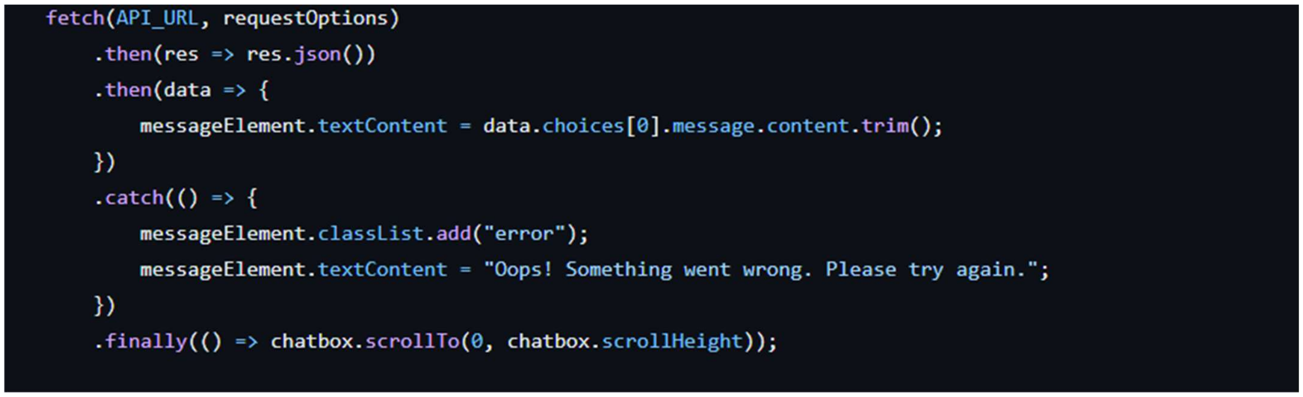
```
const requestOptions = {
    method: "POST",
    headers: {
        "Content-Type": "application/json",
        "Authorization": `Bearer ${API_KEY}`
    },
    body: JSON.stringify({
        model: "gpt-3.5-turbo",
        messages: [{ role: "user", content: this.state.userMessage }],
    })
};
```

The userMessage stored in this.state is directly used in the request body without any validation or sanitization. This could lead to security vulnerabilities such as injection attacks if the user message is not properly sanitized.



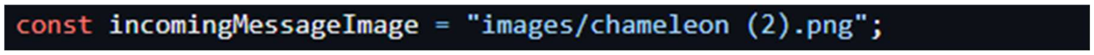
```
fetch(API_URL, requestOptions)
    .then(res => res.json())
    .then(data => {
```

Making requests to external APIs from client-side JavaScript code may lead to CORS (Cross-Origin Resource Sharing) issues if the server hosting the API does not have appropriate CORS headers configured. This can result in the browser blocking the request due to CORS policy violations.



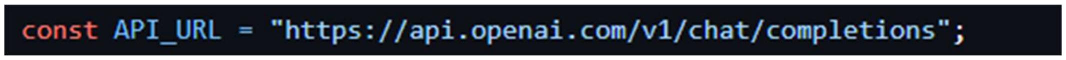
```
fetch(API_URL, requestOptions)
    .then(res => res.json())
    .then(data => {
        messageElement.textContent = data.choices[0].message.content.trim();
    })
    .catch(() => {
        messageElement.classList.add("error");
        messageElement.textContent = "Oops! Something went wrong. Please try again.";
    })
    .finally(() => chatbox.scrollTo(0, chatbox.scrollHeight));
```

Generic error messages like "Oops! Something went wrong." may leak information about the application's internals to potential attackers. It's better to handle errors more gracefully without exposing sensitive details. Well-implemented error handling aids developers in debugging and troubleshooting the application. Detailed error messages, including stack traces and contextual information, allow developers to identify the root causes of issues more efficiently. This information assists in reproducing and isolating bugs, leading to faster resolution and improved software quality.



```
const incomingMessageImage = "images/chameleon (2).png";
```

Although this is a png file. File paths and URLs should not be hard-coded in client-side code, especially if they reference sensitive or private resources. This information could be exposed to users or attackers inspecting the source code



```
const API_URL = "https://api.openai.com/v1/chat/completions";
```

Directly exposing sensitive information such as API endpoints in client-side code can lead to potential security risks, especially if this endpoint requires authentication or handles sensitive data. It's advisable to handle such information securely on the server-side and expose only necessary interfaces to the client.

Avoid hardcoding sensitive information, such as API keys, directly into the client-side code.

Store sensitive data securely on the server-side or in environment variables.

Utilize server-side endpoints to fetch sensitive information securely.

If using API keys, store them securely on the server-side and retrieve them when necessary via server-side requests.

Regularly review the codebase to identify and remove any instances of sensitive information hardcoded in the code.