Research Project: ESLint for enhanced detection and fixing of vulnerabilities in JavaScript code

Fabian Harlang, Rares-Ionut Mocanu

December 2022

1 Abstract

Sometimes, bad programming habits and a lack of knowledge in the domain of security could make the code vulnerable to web exploits. It is difficult for most developers to assess whether a code is safe against common attacks such as SQL Injection, Cross Site Scripting (XXS), Broken Authentication, and Session Management. While in the past decade, the JavaScript language has grown in popularity together with certain frameworks and libraries, the problem of detecting exploitable code remains a significant challenge for developers. Therefore our aim in this thesis is to create three rules that automatically fix certain vulnerabilities in the use of the JavaScript React library. The three vulnerabilities that we will fix and present the method in which we do so revolve around the iframe HTML attribute, the HTTP protocol and the usage of dangerous HTML. Static analysis could aid with this problem, but it requires substantial resources and effort to set it up for an entire code base. Therefore, a demand for a more efficient and elegant solution is needed. As proposed in the paper "Fixing Vulnerabilities Automatically with Linters" by Willard Rafnsson, Rosario Giustolisi, Mark Kragerup, and Mathias Høyrup, the linter tool ESLint would come in handy for serving as a strict guideline in terms of writing code with instantaneous feedback that will prevent the developer from writing vulnerable code and hence fixing and debugging the coding as it's written. [11]

2 Learning outcomes

In this project we will do research on the following matters:

- *Understand and research static analysis of JavaScript code in the context of React Development. (a prerequisite to understanding ESLint)*
- *How does the ESLint tool work and how can it prevent web exploitation?*
- *Can it mitigate and prevent vulnerabilities across multiple files/components in a code base?*
- *Add three additional security rules to ESLint that can detect and prevent flaws and vulnerabilities in React JS Code that make the application sensitive to web exploits.*

Goal of the Research Project: The purpose of the Research Project is to identify and understand vulnerabilities caused by insecure JavaScript code, gain knowledge in the field of static analysis and linting and create a good starting point for, later on, improve or adding security rules in ESLint that will fix these types of securities errors and/or prevent them from happening in the first place.

Methodology During this research project, we will scrutinize common JavaScript code for flaws and typical security vulnerabilities that may allow the most common web exploits to occur. This will help us identify more relevant rules and potential automatic fixes for them. We will also explore how ESLint may prevent this from happening, more specifically by linking the code through existing, improved, or added rules that we will develop ourselves. We will focus on JavaScript code in the context of website development using React JS and experiment with possible security issues that may be relevant specifically to React JS. This will include typical React security practices like sanitizing HTML, data binding and correct usage of iframes.

3 Introduction

3.1 Problem to address

A somewhat overlooked use for ESLint is enforcing security rules and identifying vulnerabilities in the code. We believe that there is great potential in using ESLint for this purpose and therefore through our research we were able to implement three different security rules. Indeed, in many cases the majority of developers write vulnerable software and such exploited vulnerabilities can cost a company billions in damages, and inflict irreparable harm to the privacy of individuals.

It is also the case that vulnerabilities arise under highly specific circumstances; most vulnerabilities arise from libraries and their use [11]. It is therefore certain that given the popularity of JavaScript React, a multitude of vulnerabilities can arise from its use.

3.2 Our solution

Our contribution is implementing linting rules that find - and automatically fix - certain vulnerabilities in the use of the JavaScript React library. This research project serves as a breeding ground for finding out which security rules could be implemented within ESLint such that vulnerabilities and subsequently exploitation of websites built using this specific linter tool are prevented. Later on, during our thesis project, we will further implement the discovered rules and possibly expand ESLint further with more security plugins targeted for JavaScript.

The security rules that we managed to implement within ESLint are the following:

1. Adding sandbox attribute to iframe tags together with warning about dangerous values.
2. Enforcing secure protocols for HTTP / FTP.
3. Sanitizing HTML in JSX attribute dangerouslySetInnerHTML.

In terms of novelty, there are a few things that characterise developers in the context of our research thesis. Developers want to write secure software and are in need of tools to guide them. Developers also use SAST for quality control and they do not want any false positives[11]. Rafnsson et al. also demonstrate that linters can be successfully used to find - and automatically fix - vulnerabilities in software. Subsequently our work that focuses on the implementation of certain security rules, is a proof-of-concept (PoC) which shows that specializing rules reduces or eliminates false positives.

Our rules bring a certain degree of novelty in this area. In the case of the sandbox rule, we were only able to identify a single existing rule that only warned about the missing attribute, whereas we also add the attribute but also warn the user about which values in the attribute could jeopardise the safety of a sandboxed attribute. For secure protocols, we were not able to find any rules that tackle the issue of the syntax - simply the missing *S* letter within HTTP. This matters as, HTTPS helps preventing man-in-the-middle attacks where adversaries may be able to intercept or "sniff" sensitive information provided by the users communicating with a web. As for dangerouslySetInnerHTML, we couldn't find any rules that actually fix the issue of rendering unsafe HTML attributes.

3.3 Key take-aways

In this Research Project:

1. We elaborate on the vulnerabilities that our rules will prevent.
2. We elaborate on the possible solutions for the vulnerabilities.
3. We implement three security rules in ESLint and we document the process.

4 Background

4.1 JavaScript React

JavaScript has during recent years grown into the most popular programming language as the internet has evolved with more advanced and dynamic web pages. [1] Users have faster internet connections and many everyday tasks like communication, planning and shopping are being done directly from within a browser application.

This has made JavaScript language more relevant than ever since it still is the core technology of websites with its option to manipulate the Document Object Model (DOM), in a dynamic way where often event-driven "first-class function" changes content and states of an otherwise static HTML document. Meanwhile, JavaScript does not, unlike Java, have compilation steps prior to run time and modern browsers will therefore try to interpret the code and run it the moment after using a "Just-in-time Compilation"[5]. In other words, JavaScript is not a strictly typed language nor does it have a requirement for access modifiers, hence nothing is preventing the developer from writing and running code as long as the browser can run it. This has given JavaScript some obvious shortcomings in terms of enforcing the developer to write code safely against common security vulnerabilities. It seems that the problem of vulnerabilities in JavaScript code persists and it often leads to exploitation, where the adversary will attempt SQL injections, cross-site scripting, or serialization attacks where private data can be compromised or the website manipulated.

In our case, React which is an open-source robust JavaScript library that helps with building user interfaces based on UI components, also comes with additional possible vulnerabilities. React applications generally use JavaScript XML (JSX), which is a syntax extension to JavaScript and it allows developers to embed HTML-like code within JavaScript. While the use of JSX can make it easier to build user interfaces with React, it can also introduce potential vulnerabilities if it is not used properly.

For example, if a React component contains user-generated data that is not properly escaped or sanitized before it is rendered, an attacker could potentially use JSX to inject harmful code, such as a script that steals sensitive information from the user's browser.

4.2 ESLint - How it works

Static analysis security testing (SAST) is a well-known method advised by security experts and used by many large tech companies to prevent bugs and security issues in applications using static analysis and security testing. This involves scanning the source code for potential vulnerabilities before compilation and aids the developer with instant feedback on how to fix any detected issues found. Despite the potential of these kinds of tools, it seems that most programmers are somewhat reluctant to use them as they require effort to set up and often suggest false positives and give redundant feedback, which is time-consuming for the developer to go through and may not guarantee detection of the more subtle errors anyway. Another tool that has been widely used for checking code for errors and is considered a static analysis tool is the linter tool invented by Stephen C. Johnson in 1978.

A linter is a static analysis tool that warns software developers about possible code errors or violations of coding standards that the linter tool recognizes by comparing the code to certain rules, either custom-made or generic rules developed by the lint community. In the case of JavaScript, in order to flag programming errors, bugs, and vulnerable constructs, it has become more common among developers to use linters that reduce the complexity of the written code and enforce certain stylistic or syntactic patterns in the code.[2]. Up until recently, however, linting tools have not been user-friendly and highly customizable. That changed when the open-source project ESLint founded by Nicholas C. Zakas was released in 2013 which is now the preferred linting tool among JavaScript developers with more than 23 million downloads on Visual Studio Code extension marketplace. The main purpose of ESLint is to make code more consistent and help developers avoid bugs.

In this context, ESLint is fully configurable and can be customized with different rules within the code base, both in-house developed, or downloaded from maintained repositories [2]. As we have already established, our tool of focus will be ESLint.

ESLint is a useful tool for any kind of JavaScript project, as it allows the developer to set up consistent code formatting rules, which at the same time, greatly facilitates code reviews [2]. Indeed it seems that the norm for checking vulnerabilities in programs is through ESLint.

After the plugin is installed either globally to a project using "npm install eslint –global" or (as we recommend) to a specific file using "npx eslint yourfile.js". A configuration file containing all the rules (usually in JSON format called 'eslintrc.json') will then be added to the project. From here all the rules will be listed along with an error level setting:

```
  "rules": {
        "semi": ["error", "always"],
        "qoutes": ["warning", "double"]

    }
}
```

The "error" setting considers any failure to fulfill the rule an error, while the "warning" will just notify the developer about the issue. "off" setting will disable the rule. Another way of changing the error level setting is simply to assign "1" to the rule for warning and "2" for error at end of the rule name.

ESLint has plenty of built-in rules that can be configured or disabled in the config file, but most importantly custom rules can be added to meet the requirement of a certain type of coding pattern.

It is also possible to download 'plugins' to enforce Typescript or follow a specific JavaScript style guide. This makes it easier to streamline the coding when working in frameworks like React JS or Angular.

ESLint has some obvious benefits from other static analysis tools that also may be the reason for its major success: To begin with, it is user-friendly as it can be installed and used directly from the most popular IDE (VS Code, Atom, and

Sublime) which makes it easy to get started with and thereby make developer more likely to utilize it. Secondly, It not only detects errors but also suggests a fix that (if accepted) will alter the code automatically. Finally and most importantly it is highly customizable and rules can be activated and disabled as preferred, plugins can be downloaded from repositories, using the node package manager or made from scratch. On ESLint's website, all the generic rule names are listed and can be used by adding them to the rule configuration file e.g. if we need to enforce our code to only accept triple equal signs, the name for that rule is "eqeqeq". Their website also has a "playground section" to test the rule directly online prior to adding them to the code base.

To lint a specific file in a code base e.g. "index.js" the command "npx eslint /index.js" can be run and depending on our rule's error setting either a warning or error will be displayed in the console, if any code is violating a rule. It is possible to furthermore append our rules with options. The keywords like "always" and "never" can be used to specify if we for example always allow or never allow semicolons at the end of statements. This option is an easy we to enable and disable any rule.

If the rule has an option of automatically fixing the issue (usually indicated by a tool icon) the command "npx eslint /index.js –fix" can be executed and the issues will be automatically fixed. To check and fix an entire folder for issues the folder name followed by asterisks should simply be specified in the command instead of a single file: "npx eslint /src* –fix". This will check all js-files for issues and fix them. If the folder contains other file types (.css or .html), however, the parsing error "unexpected token" will occur. This is due to the fact that ESLint is strictly designed to analyze JavaScript code and can't parse other file types. To resolve this an "ignore file" can be added to the code base in a similar approach to using version control with GitHub.

The file can be named ".eslintignore" and the file types followed by an asterisk like .html*, .css*, or any file type we wish ESLint to ignore, can be added, and hence only lint all js-files next time we run the command. To avoid writing the same command over and over it's efficient to write a script in the package.json file i.e "lint: "./src/*". This will run ESlint on all files in the source folder simply by typing "npm run lint" in the terminal. Another useful script could be "fix: "./src/* –fix" that would fix all detected issues in the code base using the automatic fix option.

ESLint is highly compatible with VS Code and from the extension marketplace, the plugin can be installed with one click. The advantage of using the plugin is that it will display the error or warning message simply by hovering over the line of code. It also provides a button to view a detailed description of the problem and a quick fix button that will simulate the –fix command. In the console, a problem section will appear listing all the issues found in the code. Right-clicking them will display a drop-down menu with various options to either ignore or fix the error. It may at times be useful to extend the .eslintrc file with additional rules from another JSON file. This can be done by initializing the config file with the keyword "extends" followed by the file path of the additional rules or plugins:

```
"extends : [
"./MoreRules.json",
"plugin:"MyPluginName:recommended"
],
```

Moreover, ESLint community provides a set of rules that are recommended to use we can simply extend our configuration file with

```
"extends":[ eslint:recommended"],
```

and our code base will automatically be checked for the most common linting rules. The recommended rules can at any time be altered simply by overwriting the rule in our custom configuration using the exact same rule name. It's also worth considering the "env" (environment setting) and specifying if the code is being run from node or the browser. The latter will allow regular JavaScript DOM operations e.g. "document.getElementbyId('id').."

Finally, if we don't want to configure manually, ESLint can be configured conveniently and without a profound understanding of the above, simply by using the node command "npm init @eslint/config".

This will run a setup wizard in the console helping the developer set up the tool by asking some simple questions like "how would you like to use ESLint?", "What modules do you need?", Which framework do you use". "Do you use Typescript?" Answering these questions will customize ESLint to fulfill the most common needs when developing in JavaScript right out of the box.

Needless to say, ESLint is a very powerful and user-friendly linting tool that can provide many types of guidelines when coding. For that reason, it also has a huge potential to catch more sophisticated patterns in the code. Namely, those types of more "invisible" bugs that do not break the functionalities or logic in the code, but may cause vulnerabilities to common web exploits. We will in the following section addresses some of these issues and consider how ESLint can assist us in finding and fixing them [2].

4.3 Security Practices in React

React JS is a JavaScript framework created by Facebook in 2013. It allows developers to easily built encapsulated and reusable components that manage their own state One of the key benefits of using React JS is its ability to improve the performance of web applications. This is achieved through the use of a virtual DOM, which allows React to efficiently update only the components that have changed, rather than re-rendering the entire page. This can greatly improve the speed and responsiveness of the application, particularly when dealing with large or complex datasets. In terms of security, React has some best practices to follow that will prevent vulnerabilities in the code. Per default, React protects against XXS with data binding using JSX curly braces to dynamically display data and thereby escape the values:

```
Ex. <div>{data}</div>Ì
```

This protection, however, will not be useful when rendering HTML Attributes. To do so it's recommended to sanitize the code beforehand using the DOMpurify library:

```
import purify from "dompurify";
<div dangerouslySetInnerHTML=

{{__html:purify.sanitize(data) }} />
```

It is also recommended to be aware of potential URL-based script injection. URLs should always begin with http or https protocol and a validate function may be used to check if the URL is on an "allowed-list" and or follows a specific convention.

More security practices recommended by the React community include vulnerabilities in the dependencies, using the latest versions of React, Avoiding JSON injection attacks by escaping strings etc. [8] Although these recommendations are useful and great to know about, there is meanwhile nothing that enforces the developer to follow them or notifies the developer if they are being violated somewhere in the code. In other words, these are best practices, but not actual linting rules. For that reason, we have tried to identify potential ESLint security rules for React. In the research process, it would seem that especially the JSX attribute DangerouslySetInnerHTML which allows the developer to insert customized HTML directly in the DOM is an ongoing security concern in the context of React and although there are many rules out there warning about the usage of the attribute apparently none of them provide an automatic fix option.

5 Customized rules

5.1 Vulnerabilities using DangerouslySetInnerHTML

This attribute is often used to render user-generated content, such as comments or forum posts, on a web page. However, this can be a security concern because it allows attackers to inject malicious code into the page, which can then be executed by the browser. This can lead to a variety of attacks, such as cross-site scripting (XSS) or SQL injection.

To mitigate this risk, developers should use the "dangerouslySetInnerHTML" attribute with caution, and always sanitize the user-generated content to remove any potentially dangerous characters. Additionally, they should consider using other methods, such as server-side rendering, to display user-generated content on the page. This can help to prevent attackers from injecting malicious code into the page and protect the website from common security vulnerabilities. Meanwhile, as mentioned earlier no rules currently automatically sanitize the HTML content being inserted in the attribute i.e:

```
<div
    dangerouslySetInnerHTML={{
      __html:
      "<h2 onmouseover='alert(something malicious)'>hello world</h2>
      <script>something even worse</script>" ,
   }}
  ></div>
```

In the above example, a mouse-over event will execute JavaScript code. After the h1 element, another script is executed using the script attributes. If this is done intentionally it's fine, but it leaves the application vulnerable to script injections of the unwanted kind. Therefore we wish to only allow plain HTML in this attribute i.e sanitizing the HTML.

To do so we can either use a complex regex pattern to identify and replace all js code within HTML or use a library. A popular one is the widely used sanitize-html library. To use the "sanitize-html" package, developers simply need to install it from the npm registry and require it in their code. (we included this in our dependencies) This will allow them to use the sanitize() function to sanitize a string of HTML content, passing in options to specify which elements and attributes should be allowed.

```
const sanitize = require('sanitize-html');

let html =
'<p>Hello, <b>world!</b></p><script>alert("XSS")</script>';

let sanitized = sanitize(html, {
 allowedTags: ['p', 'b'],
 allowedAttributes: {
  'p': ['style'],
 'b': ['style']
}
});
```

This seems however like a bit of work to do for a single HTML insertion. Therefore we thought about utilizing this library automatically as a fix in a customized ESLint rule.

5.2 Implementing HTML sanitize rule

Initially, we tried to make a rule that would call the sanitize function "inline" of the current attribute. i.e

.

```
dangerouslySetInnerHTML={{
__html: sanitize(<h1>some html</html>) (...)
```

This approach did not work however as the library in that case still had to be imported manually to work.

Another way would then be to require the library directly from the ESLint rule and then traverse through the abstract syntax tree and call the sanitize function on the value of the node found with the name "dangerouslySetInnerHTML".

```
JSXAttribute: function(node) {
if (node.name.name === 'dangerouslySetInnerHTML') { ...
```

Once the rule detects the usage of such an attribute we find the content of the attribute using the context.sourcecode() function. Within this source code, we can fetch the value of the attribute and declare it to a variable called insecureHTML. Then another variable can be declared with the name sanitized. This variable will call the sanitize function and take insecureHTML as an argument i.e:

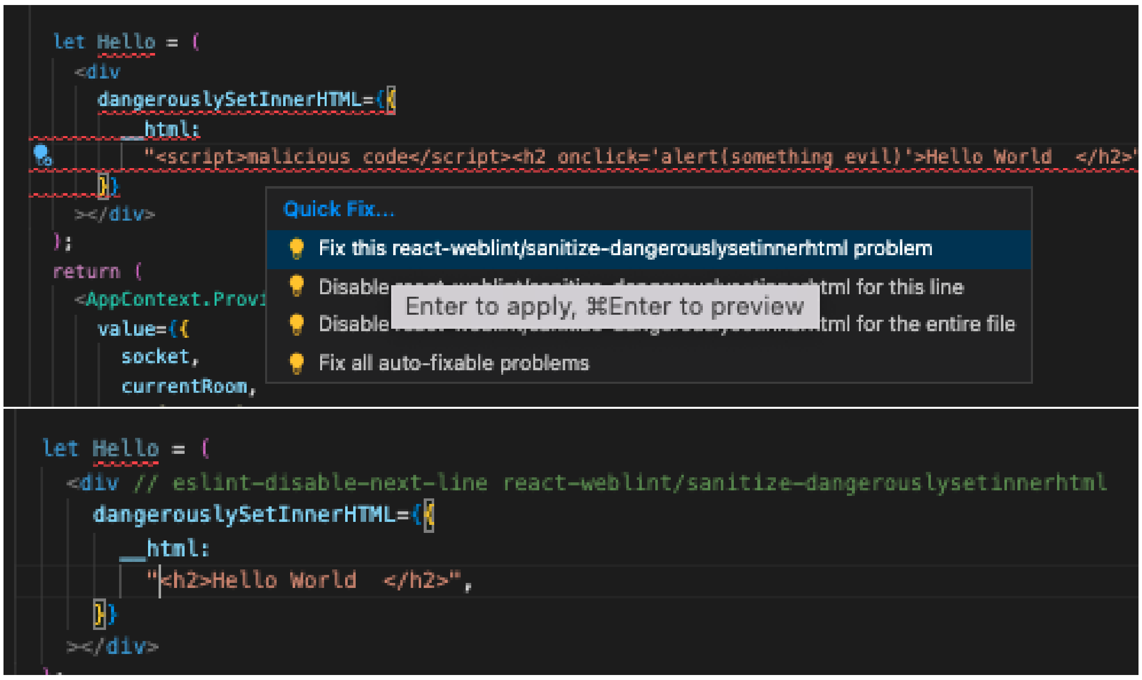
```
// Get the value of the dangerouslySetInnerHTML attribute
let insecureHTML = sourceCode.getText(node.value)

// Use Sanitize library to sanitize the HTML
let sanitizedHTML = sanitizeHtml(insecureHTML);
```

Finally, we utilize the fixer object and call the function replaceText to substitute the insecure HTML with the sanitized HTML. In addition to that, we use another fixer function InserTextAfter to insert a comment above the code that will disable any further warnings for this attribute:

```
return[
 fixer.replaceText(node.value, sanitizedHTML),
 // When the automatic fix has run.
 Disable the error/warning for this rule.
 fixer.insertTextAfter(node.parent.name, "
 // eslint-disable-next-line
 react-weblint/sanitize-dangerouslysetinnerhtml.js")
```

[3]


Quick Fix...
Fix this react-weblint/sanitize-dangerouslysetinnerhtml problem
Disable ...tml for this line
Enter to apply, ⌘Enter to preview
Disable ...tml for the entire file
Fix all auto-fixable problems

```
let Hello = (
  <div
    dangerouslySetInnerHTML={{
      __html:
        "<script>malicious code</script><h2 onclick='alert(something evil)'>Hello World  </h2>'
    }}
  ></div>
);
return (
  <AppContext.Provi
    value={{
      socket,
      currentRoom,
```

```
let Hello = (
  <div // eslint-disable-next-line react-weblint/sanitize-dangerouslysetinnerhtml
    dangerouslySetInnerHTML={{
      __html:
        "<h2>Hello World  </h2>",
    }}
  ></div>
```

Figure 1: Automatic fix for sanitizing HTML insertion

5.3 Vulnerabilities of insecure protocols

HTTP (Hypertext Transfer Protocol) and HTTPS (HTTP Secure) are both protocols for transmitting data on the web. The main difference between the two is that HTTPS uses a secure SSL/TLS connection to encrypt data sent between a web server and a client. This makes HTTPS more secure than HTTP, as it helps to prevent man-in-the-middle attacks where adversaries may be able to intercept or "sniff" sensitive information provided by the users communicating with a web server. FTP (File Transfer Protocol) and FTPS (FTP Secure) are similar protocols for transferring files between computers on a network. The main difference between HTTP/HTTPS and FTP/FTPS is that the latter is primarily used for transferring files, while the former is used for transferring web pages and other data on the web. Similar to HTTPS, FTPS-protocols use a secure SSL/TLS connection to encrypt data sent between a client and a server, making them more secure than HTTP and FTP, which do not use encryption. This makes HTTPS and FTPS more suitable for transferring sensitive information and can help to prevent man-in-the-middle attacks and other forms of data interception that can be done using packet-analyzer tools like Wireshark that allow attackers to see traffic being transmitted over a network. Although it seemingly would be simple to ensure a website has an SSL certificate, surprisingly many websites don't and transmit data back and forth from the server without encryption. We have therefore attempted to enhance an ESLint rule that would detect usage of HTTP/FTP-protocols in the code base and encouraged the developer to use our automatic fix that simply replaces the protocol with the secure alternative. [4]

5.4 Implementing secure protocols rule

To begin with, we created two different messages in the meta object of the rule. One is to be displayed for each type of protocol (HTTP/FTP). Then we tried to detect the usage of insecure protocols and localize it/them in the code by traversing through the abstract syntax tree and returning any values of nodes that would be of type string and have the prefix "http".

```
        if (typeof node.value === 'string') {
            if (node.value.startsWith('http:')) {....
```

If any matches are found we simply utilize the fixer object with the function replaceText to substitute the insecure protocol with its counterpart.

To also check for FTP-protocols we prolonged the rule to detect usage of the prefix "ftp" i.e

```
   ...   } else if (node.value.startsWith('ftp:')) {
```

Now the developer will be encouraged to use secure protocols. Although it may be a subtle fix, it will be efficient to have an automatic fix for an entire code base with potentially many URLs. The rule was the initial one we did and provided us a great opportunity to become familiar with the customization of ESLint rules and the usage of the abstract syntax tree (AST) to map the syntactical structure of the code.

5.5 What is an iframe?

Another vulnerability, that we have decided to investigate revolves around iframe. Basically, the role of this HTML element is to load another HTML page within a given document. Each iframe element represents a nested browsing context that has its own session history and document. The browsing context that embeds the subsequent ones, is called the parent browsing context. The topmost browsing context is usually the browser windows, which is represented by the *Window* object [6]. These elements are commonly used for advertisements, embedded videos, web analytics, tweets embedded into news articles and interactive content such as maps [10].

5.6 Vulnerabilities for iframe

The same-origin policy is an important security feature of any modern browser. Its purpose is to restrict cross-origin interactions between documents, scripts, or media files from one origin to a web page with a different origin.

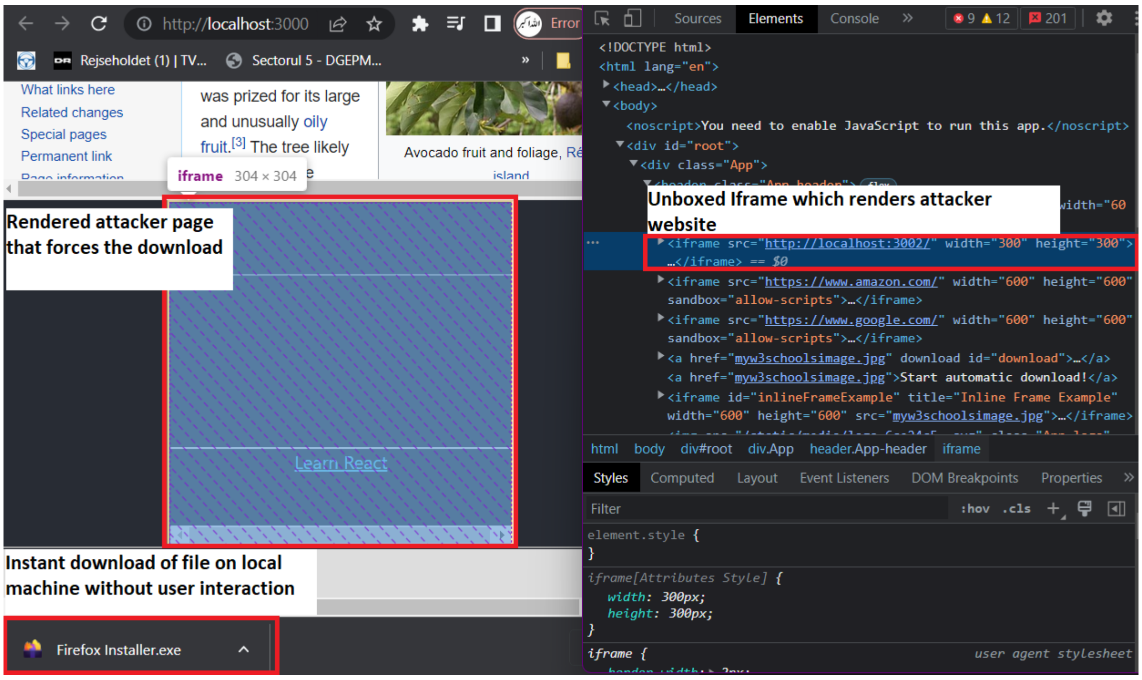
It is possible to use an iframe maliciously. A hacker who compromises a website can inject an invisible iframe to hijack page clicks, install malware or steal information. Computer malware with browser extensions can also inject an iframe into legitimate websites. This can cause legitimate websites to show unwanted ads or use the computer as part of a robot network (botnet). Other issues could also be click jacking, malicious popups/forms, execution of plugins, scripts, or downloads without user activation, etc. This is of course the case if the rendered page within the frame belongs to a potential attacker or if the respective page has been exploited.

It can also be the case that another site loads a legitimate site in an iframe, more specifically Cross-Frame Scripting (XFS), which represents an attack that combines malicious JavaScript with an iframe that loads a legitimate page in an effort to steal data from an unsuspecting user. For example, let's imagine that the attacker would trick someone into visiting *https://faceboook.com.* This fake website (with "book" spelled wrong) would contain an iframe that would load the content to the correct website *https://facebook.com.* As the victim logs in, the malicious website can now read different data on the victim. Fortunately, popular websites like Facebook, Google, or Amazon follow the same-origin policy (SOP), which means that the originating location of the requested resource has to be the same as the resource provider. If the two would match, then the browser allows the requested resource to be loaded.

In terms of Cross-Site Scripting (XSS), iframe could also be exploited if a related vulnerability exists. However, this attack is effectively the same as a conventional XSS attack. The only difference is that with iframe, the attacker can hide the element such that the user can't realize that they just visited a malicious website [7].

In the case of CVE-2021-38503, the iframe sandbox rules were not correctly applied to XSLT stylesheets, allowing an iframe to bypass restrictions such as executing scripts or navigating the top-level frame [1]. We will discuss the security advantages of having a sandboxed iframe within the following section.

We have also simulated a situation in which the victim on $localhost : 3000$ renders an attacker page $localhost : 3002$ within an iframe. As mentioned before, it could be the case that the rendered page on $localhost : 3002$ could have initially been harmless but at some point in time it got hacked and now it downloads malicious files on the rendered page. This is bad news as if the rendered page downloads the malicious file, also the topmost browsing context ($localhost : 3000$) will download it on load. In Figure 2 it is shown how the victim could potentially download a harmful file on their computer without any interaction from the user's side. This highlights the matter that there is a risk for the main website in terms of every single rendered page within an iframe.


Rendered attacker page that forces the download
Unboxed Iframe which renders attacker website
Instant download of file on local machine without user interaction
Firefox Installer.exe
iframe 304 × 304
Learn React

```
<!DOCTYPE html>
<html lang="en">
 ▶<head>…</head>
 ▼<body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
   ▼<div id="root">
     ▼<div class="App">
        ▶<iframe src="http://localhost:3002/" width="300" height="300">
         …</iframe> == $0
        ▶<iframe src="https://www.amazon.com/" width="600" height="600"
         sandbox="allow-scripts">…</iframe>
        ▶<iframe src="https://www.google.com/" width="600" height="600"
         sandbox="allow-scripts">…</iframe>
        ▶<a href="myw3schoolsimage.jpg" download id="download">…</a>
         <a href="myw3schoolsimage.jpg">Start automatic download!</a>
        ▶<iframe id="inlineFrameExample" title="Inline Frame Example"
         width="600" height="600" src="myw3schoolsimage.jpg">…</iframe>
```

```
element.style {
}
iframe[Attributes Style] {
   width: 300px;
   height: 300px;
}
iframe {                                   user agent stylesheet
```

Figure 2: Iframe Renders Attacker Website

5.7 Our Solution for iframe

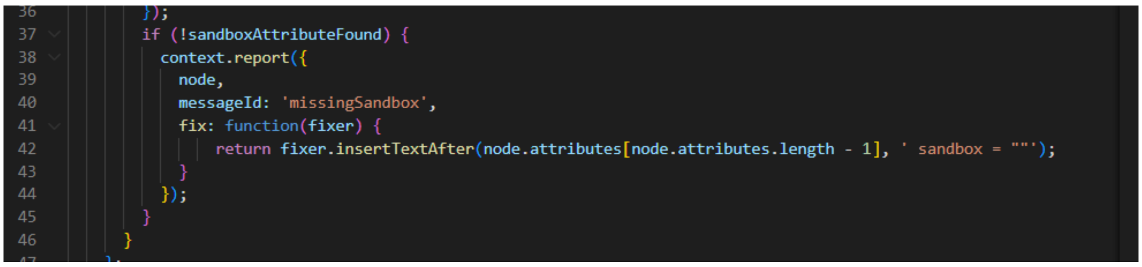
The **principle of least privilege (PoLP)** requires that a process, user, or program, must be able to access only the information and resources that are necessary for its legitimate purpose [9].

Therefore we are aiming to block each and every feature that isn't directly relevant to the functionalities that we would like to use. As a result, the piece of content embedded through the iframe won't be able to take advantage of the privileges given by the HTML tag while the "sandbox" attribute is in place. This attribute enables an extra set of restrictions for the content in the iframe. Sandboxing in iframe works on the basis of whitelisting certain permissions. The value of the sandbox attribute can either be empty (then all restrictions are applied), or a space-separated list of pre-defined values that will remove the particular restrictions.

When the sandbox attribute is present, it will: treat the content as being from a unique origin, block form submission, block script execution, disable APIs, prevent links from targeting other browsing contexts, prevent content from using plugins (through $< embed >, < object >, < applet >$, or other), prevent the content to navigate its top-level browsing context, block automatically triggered features (such as automatically playing a video or automatically focusing a form control). Since all, it takes for the sandbox attribute to be enabled is an empty string, and not knowing which permissions would a given user want to enable, we decided to create a rule that warns about the missing attribute and also adds it. On top of this, since the permissions "allow-scripts" and "allow-same-origin" both allow the embedded document to remove the sandbox attribute - therefore making it no more secure than not using the sandbox attribute at all - we also decided to warn the user about these two implementations, this way we would at least make sure that the user knows what they are doing.

5.8 Writting the Rules for Iframe

For implementing the required rule for the iframe element, we had to understand what the abstract syntactic structure of a possible iframe element looks like. For this, we used *https : //astexplorer.net/*. From there on, we were able to model specific code for what we wanted to achieve. Each iframe tag together with its content's, represents a JSXElement that has an openingElement and a closingElement. The openingElement itself, can contain different attributes including the *sandbox* attribute which we aim to tackle. The attributes are further split into the name of the attribute and the given value, which in our solution would be an empty string. We were therefore able to identify which JSXElements contain the sandbox attribute and subsequently report a warning message that the sandbox attribute is missing. Added to this, we have also decided to implement a fix for this issue, in which we are adding a sandbox attribute with an empty string. Having such a value, will basically enable all restrictions and will force the user to add their own restrictions accordingly. Figure 3 shows our fixed solution for the missing attribute. We are introducing the attribute at the end of the attribute list within the element, with an empty string. It is also important to mention that it will not overwrite existing sandbox attributes or their values.



```
        });
        if (!sandboxAttributeFound) {
          context.report({
            node,
            messageId: 'missingSandbox',
            fix: function(fixer) {
              return fixer.insertTextAfter(node.attributes[node.attributes.length - 1], ' sandbox = ""');
            }
          });
        }
      }
```

Figure 3: Adding the Sandbox Attribute With an Empty String Value

As mentioned in the previous section, since the permissions "allow-scripts" and "allow-same-origin", both allow the embedded document to remove the sandbox attribute, we decided to warn the user about this risk. A fix could be removing the permissions, however, at this point, it is just a matter of preference since the fix could interfere with what the user is actually intending to do.

6 Conclusion

When we initially started looking into this subject, we observed that many of the existing rules for security are not necessarily maintained or provide an automatic fix option. This obviously represents a serious risk in an ever-changing security environment as new vulnerabilities could arise that might even jeopardize the validity of some already existing rules.

The potential for developing security rules in linters, more specifically in ESLinter, is quite relevant and it is something that most programmers should be aware of. If not for developing the rules or expanding on them themselves, at least for adopting the habit of using them for security purposes.

In our endeavor to understand the value of creating and using such rules in ESLint, we looked at three possible vulnerabilities. The simple iframe HTML element holds great risk if not properly sandboxed. Such is the case for insecure protocol and usage of dangerous HTML rendered directly in the DOM without proper sanitation. Having to create the rules for these vulnerabilities encouraged us to scrutinize code for potential security flaws and understand how customized rules for ESLint work, but also forced us to familiarise ourselves with how to interact with the **Abstract Syntax Tree** (AST) of JavaScript during the implementation. As a result, we were able to implement the specified rules within ESLint and collect them into a custom node package, outlining that the potential for using this tool for security purposes is indeed significant.

References

[1] Armin Ebert. *Cross Frame Scripting.* URL: `https://bugzilla.mozilla.org/show_bug.cgi?id=1729517`. (accessed: 18.11.2022).

[2] ESLint. *ESLint Docs: Getting Started.* URL: `https://eslint.org/docs/latest/user-guide/getting-started`. (accessed: 10.11.2022).

[3] Rares Mocanu Fabian Harlang. *Github repository of react-weblint.* URL: `https://github.itu.dk/rarm/eslint-plugin-react-weblint`. (accessed: 10.12.2022).

[4] IBM. *Secure Sockets Layer (SSL) protocol.* URL: `https://www.ibm.com/docs/en/ibm-http-server/9.0.5?topic=communications-secure-sockets-layer-ssl-protocol`. (accessed: 10.12.2022).

[5] IBM. *The JIT compiler.* URL: `https://www.ibm.com/docs/en/ztpf/1.1.0.15?topic=reference-jit-compiler`. (accessed: 10.11.2022).

[6] Mozilla. *Iframe element.* URL: `https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe#scripting`. (accessed: 15.11.2022).

[7] Owasp. *Cross Frame Scripting.* URL: `https://owasp.org/www-community/attacks/Cross_Frame_Scripting`. (accessed: 15.11.2022).

[8] Liran Tal Ron Perris. *Best practices - React security.* URL: `https://snyk.io/blog/10-react-security-best-practices/`. (accessed: 10.12.2022).

[9] Schroeder Michael D. Saltzer Jerome H. “The protection of information in computer systems”. In: *IEEE Trans. Softw. Eng.* Proceedings of the IEEE. Institute of Electrical and Electronics Engineers (IEEE) (1975). DOI: `10.1109/proc.1975.9939`.

[10] Techtarget. *Iframe element.* URL: `https://www.techtarget.com/whatis/definition/IFrame-Inline-Frame`. (accessed: 15.11.2022).

[11] Mark Kragerup Willard Rafnsson Rosario Giustolisi and Mathias Høyrup. “The Adoption of JavaScript Linters in Practice: A Case Study on ESLint”. In: *Fixing Vulnerabilities Automatically with Linters* (2018).