OCTOBER 28, 2021 / #REACT

How to Secure Your React.js Application

Shachee Swadia

React.js is a scalable open-source JavaScript library and is one of the most commonly used front-end frameworks out there today.

It's dynamic and is easy to get started with if you want to create interactive web applications with reusable components.

There are lots of reasons to use React.js in your application:

- its flexibility – you can create complex applications without reloading the webpage
- its simplicity – you can get a project up and running quickly and easily
- its ease of use with other JS libraries
- its customizability – there are many open-source components that can be integrated with your project.

As you can see, React is great. But there are some thing you need to be aware of when using it for your projects.

React Security Vulnerabilities

React is convenient and fast, which can make it risk-prone and it's easy to forget about security concerns.

Though React has a smaller number of attack points than other frameworks, it is still not entirely secure. Since React is compatible with other open-source components and does not have strong default security settings, it becomes vulnerable to security slips.

Massive amounts of personal data are constantly being shared by various apps. This increases the danger (and the probability) of exposing private and financial data. And if your company uses React, they could face violations of privacy regulations in case of a data breach.

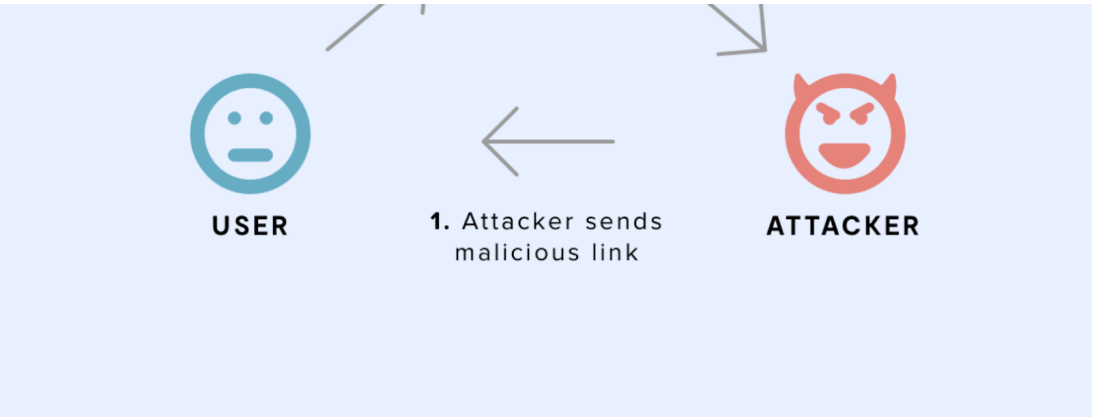
Your React application will be useless without proper security features, so it's better to err on side of caution and tackle these security threats head-on.

Most Common Security Threats to a React Application

Since React is always being updated and improved, I can't create an exhaustive list of vulnerabilities here. But I'll discuss some of the more well-known and common threats here.

1. Cross-Site Scripting (XSS)

XSS is a serious client-side vulnerability. A perpetrator is able to add some malicious code to your program that is interpreted as valid and is executed as a part of the application. This compromises the functionality of the app and the user data.



[Source]

There are two types of cross-site scripting attacks:

1. **Reflected XSS** – Here, an attacker uses a malicious link with some JS code that the browser processes to access and manipulate the page content, cookies, and other important user data.
2. **Stored XSS** – In this attack, the malicious content is stored on a server and executed when a user requests the stored data. This leads to unwanted content on your webpage.

2. Broken Authentication

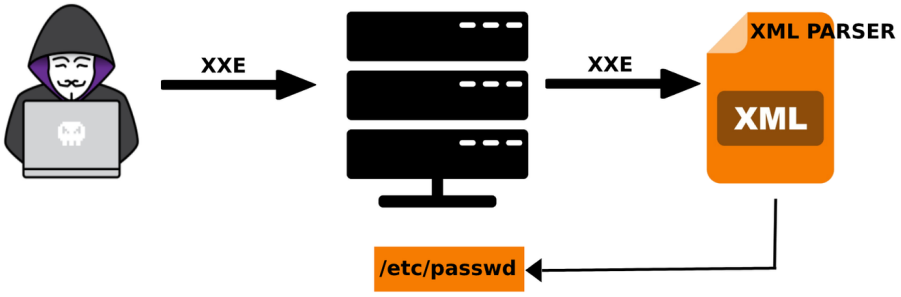
Another common issue in React.js applications is inadequate or poor authorization. This can result in attackers hacking user credentials and carrying out brute force attacks.

There are various risks associated with broken authorization, like session IDs being exposed in URLs, easy and predictable login details being discovered by attackers, the unencrypted transmission of credentials, persisting valid sessions after logout, and other session-related factors.

3. SQL Injection

This vulnerability exposes the database of your application. An attacker injects harmful SQL code that allows them to edit data without permission.

4. XML External Entity Attack (XXE)



[Source]

An XXE attack is when the offender targets the XML parser which is required to transform XML into readable code.

Malicious code is injected into the parsers to collect sensitive data or even attempt a CSRF (Cross-site request forgery) and DDoS (distributed denial-of-service) attack.

5. Zip Slip

There's a very specific vulnerability in React applications known as "zip slip" which involves the exploitation of the feature that enables uploading zip files.

The attacker could unzip the uploaded files outside the assigned directory if the archive used to unpack the zip file is not secure and then they can gain access to the file.

6. Arbitrary Code Execution

This threat is a general risk that enables an attacker to execute arbitrary

your configuration files or any part of the code for that matter.

Alright, now that we know what can go wrong, let's see how to guard against it.

Best Practices for React.js Security

As they say, an ounce of prevention is worth a pound of cure – so it's always a good idea to follow proper protocols and ensure that your application is secure.

You might not think about every possible vulnerability, but you can definitely make your app more secure by mitigating the most common risks.

[Source]

Following are some of the best practices you should follow to secure your React applications:

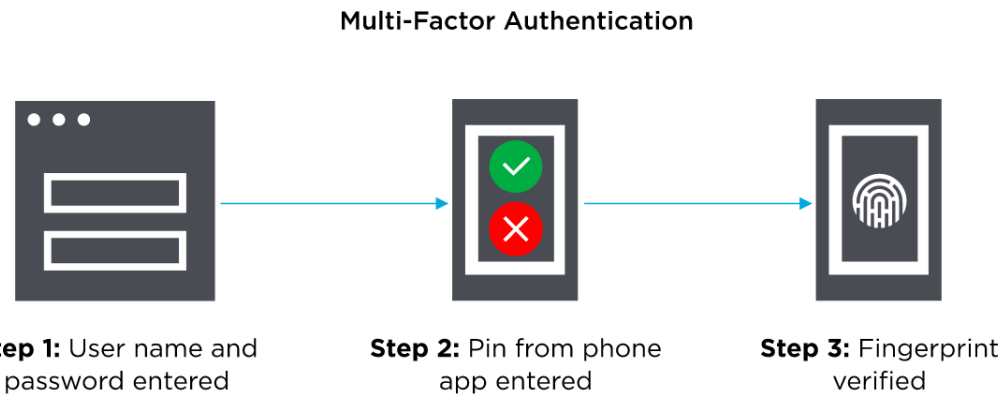
make sure that the connection between the server and the client is secure.

A simple way to do this when building your application is to ensure that the domain header has a realm attribute. A realm contains the list of valid users and prompts for a username and password when accessing any restricted data.

Following is an example of how you can set up a security realm:

```
<security-realm name="ApplicationRealm">
  <authentication>
    <local default-user="$local" allowed-users="comma-separated-list"/>
    <properties path="application-users.properties"/>
  </authentication>
  <authorization>
    <properties path="application-roles.properties"/>
  </authorization>
</security-realm>
```

When possible, another easy and effective technique is to use multi-factor authentication. This authentication method ensures that a user is granted access to important parts of your application only after providing two or more authentication credentials for verifying their identity.



[Source]

When your React.js app has the basic secure authentication all set, it helps mitigate XSS and broken authentication issues.

2. Make sure that the HTML code is resilient

Any React application will need HTML to render it, so it's imperative to make sure that your HTML code is not vulnerable. Three constructive ways to do this are:

A. Disable HTML markups

When the 'disabled' attribute is set for any HTML element, it becomes non-mutable. It is not possible to focus or submit that element with a form.

You can then put some validation and enable the element only when that validation is true. This prevents any malicious data from being submitted that can cause disastrous effects.

Here's an example code snippet to disable a button:

```
var Component = React.createClass({
    getInitialState() {
        return {
            submitting: true
        }
    },

    handleSubmit() {
    },

    render() {

        return (<div>
            <button type="button" disabled={this.state.submitting} onClick={ this.hand
    }
});

ReactDOM.render(
    <Component />,
    document.getElementById('container')
);
```

JavaScript XML (JSX) is a syntax that lets you write HTML in React. And it has an inbuilt auto-escaping functionality that you can use to secure your application.

If you bind data with curly braces {} by default, then React will automatically escape values that are not a part of the bound data.

Here's an example:

```
return (<p style={{color: myAppColor}}>{myAppRating}</p>);
```

If a hacker tries to inject extra code into the variable myAppColor such as *color: purple, background-color: pink* then the JSX parser will detect this invalid CSS input. So the additional data will be escaped, and the attack will be neutralized.

C. Utilize dangerouslySetInnerHTML and sanitize HTML

Your application may need to render dynamic HTML code like user-provided data. This is done using 'innerHTML' which makes the app vulnerable to malicious data.

React has a feature that can notify you of this potential vulnerability called the `dangerouslySetInnerHTML` prop. This acts as a warning so you can check and make sure that the data entered when this prop exists comes from a trusted source.

```
return (<p dangerouslySetInnerHTML={{__html: myAppReview}}></p>);
```

You can also use libraries like DOMPurify to scan user input and remove

```
// Import DOMPurify
const DOMPurify = require('dompurify')(window);

// Sanitize the review
return (<p dangerouslySetInnerHTML={{__html: myAppReview}}></p>);
```

Now, imagine that an attacker adds the 'onerror' code with the image as follows:

```
The app is <b>robust</b> and <i>interesting.</i>.
<img src="reviewPic.png" onerror="alert('This app is not good!');" />
```

The sanitized value would result in the following:

```
The app is <b>robust</b> and <i>interesting.</i>.
<img src="reviewPic.png">
```

All these measures protect your React application from attacks like XSS and arbitrary code execution.

3. Use allowlist/blocklist and validation while URL parsing

When using the anchor tag `<a>` and URLs for linking content, you need to be very careful about attackers adding payloads prefixed with JavaScript.

To avoid URL-based malicious script injection, always validate the URL using the HTTP or HTTPS protocols.

```
function validateURL(url) {
    const parsed = new URL(url)
    return ['https:', 'http:'].includes(parsed.protocol)
```

Another way of protecting your React application is using the allowlist/blocklist method. Allowlisting is when you have a list of all the links that are safe and allowed to be accessed, whereas blocklisting is having a list of all potential threats that will be blocked if access is requested.

It is difficult to keep track of all the possible harmful links, so a good practice is to allowlist known sites and block everything else.

URL validation helps prevent broken authentication, XSS, arbitrary code execution, and SQL injection.

4. Always use the principle of least privilege when allowing a connection to any database

In your React application, always use the principle of least privilege. This means that every user and process must be allowed to access only the information and resources which are absolutely necessary for their purpose.

It is dangerous to allow anyone to update, insert or delete when connecting to your application's database so it is important to assign the right database roles to various users.

Never give admin privileges for your application's database to anyone unless it is vital. This makes your application safer and less prone to SQL injection attacks.

5. Secure your React APIs

The good and bad part of React APIs is that they allow connections between your app and other services. These can store information and even execute commands. This exposes your app to XSS and SQL injection.

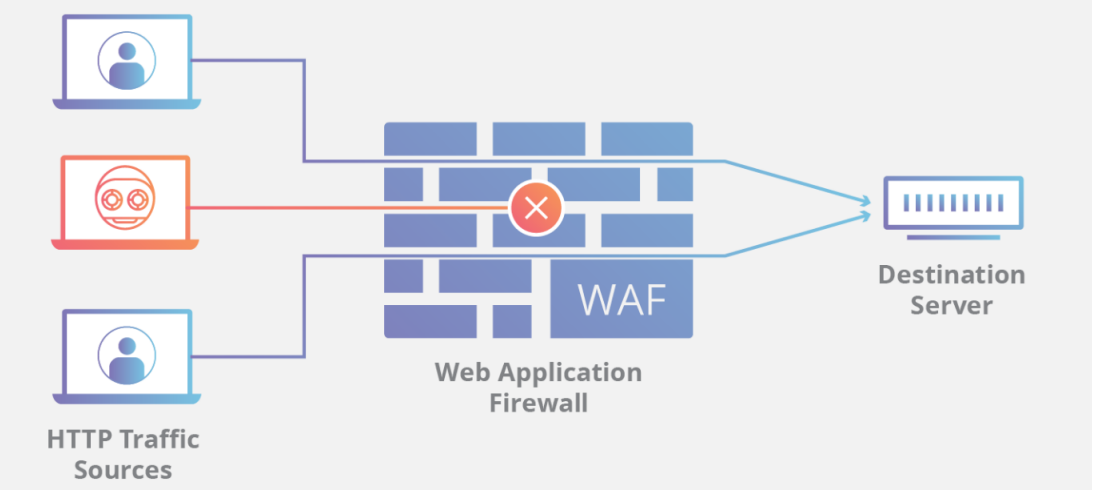
A powerful mitigation technique against this vulnerability is validating all the API functions with respect to their API schemas. Also, schedule timely

data through APIs.

```
window.__PRELOADED_STATE__ =  ${JSON.stringify(preloadedState).replace( /</g, '\\u003
```

6. Implement a Web Application Firewall (WAF)

A WAF is an application filter that detects and blocks malicious content by monitoring, analyzing and filtering bi-directional traffic.



[Source]

You can implement a web application firewall in three ways:

1. Network-based firewall which is on the hardware level.
2. Host-based firewall that is integrated into the software.
3. Cloud-based WAF

The signature-based filtering of WAF is quite effective in countering SQL injection, XSS, arbitrary code execution and zip slip.

- Confirm that the file names are standard and without any special characters.
- Whenever the files are uploaded as zip, always rename them before extracting and using the files.
- Store all files of a single component together in one folder so that any suspicious file can be quickly discovered.

8. Never serialize sensitive data

There is a good chance that your React app uses JSON to set the initial state of your application.

This can be potentially dangerous because JSON.stringify() is a function that converts any data into a string without detecting malicious values. An attacker can manipulate data like username and password by injecting a JS object that can modify valid data.

```
<script>window.__STATE__ = ${JSON.stringify({ data })}</script>
```

You can either use the serialize-javascript NPM module that will escape the rendered JSON or use complex JSON formats that will avoid serialization. But the best way to prevent any mishap is to omit confidential data from the serialized form.

Conclusion

There are a lot of potential threats you have to think about when creating a React application. Without proper security, your app may become the victim of a cyber-attack which can lead to financial loss, wasted time, breaches of trust, and legal issues.

You can either hire React developers specializing in security or outsource the development to a software development company who specialize in the development of React JS Applications. When it comes to security, make sure you have an expert on your side!

Shachee Swadia

I am a software freelancer who loves traveling and reading. Professionally, I create visual stories for individuals and organizations to help convey their messages in an effective way.
