# myNav_2.1_React_SAST

Jan 20, 2023

myWD Security

# Report Overview

## Report Summary

On Jan 18, 2023, a source code review was performed over the mynav_react_ui_v2.1 code base. 352 files, 31,646 LOC (Executable) were scanned. A total of 6 issues were uncovered during the analysis. This report provides a comprehensive description of all the types of issues found in this project. Specific examples and source code are provided for each issue type.

| Issues by Fortify Priority Order | |
|---|---|
| Critical | 6 |

# Issue Summary

## Overall number of results

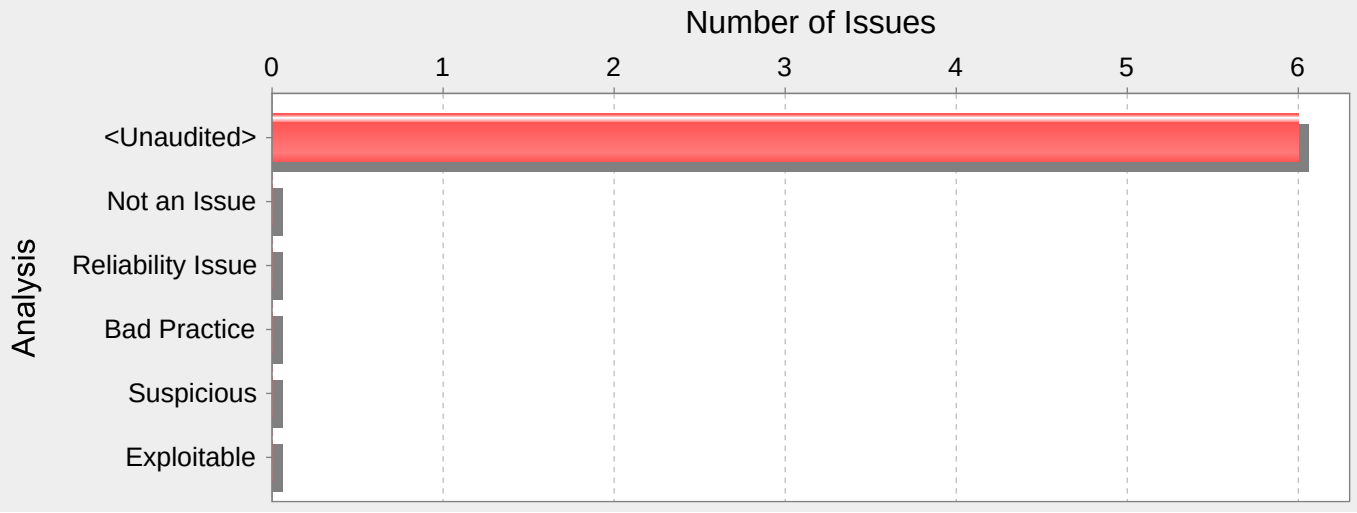
The scan found 6 issues.

| Issues by Category | |
|---|---|
| Cross-Site Scripting: DOM | 6 |

# Results Outline

## Vulnerability Examples by Category

## Category: Cross-Site Scripting: DOM (6 Issues)




### Abstract:

The method Button() in Button.js sends unvalidated data to a web browser on line 14, which can result in the browser executing malicious code.

### Explanation:

Cross-site scripting (XSS) vulnerabilities occur when:

1. Data enters a web application through an untrusted source. In the case of DOM-based XSS, data is read from a URL parameter or other value within the browser and written back into the page with client-side code. In the case of reflected XSS, the untrusted source is typically a web request, while in the case of persisted (also known as stored) XSS it is typically a database or other back-end data store.

2. The data is included in dynamic content that is sent to a web user without validation. In the case of DOM-based XSS, malicious content is executed as part of DOM (Document Object Model) creation, whenever the victim's browser parses the HTML page.

The malicious content sent to the web browser often takes the form of a JavaScript segment, but can also include HTML, Flash or any other type of code that the browser executes. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data such as cookies or other session information to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site.

Example 1: The following JavaScript code segment reads an employee ID, eid, from a URL and displays it to the user.

```
<SCRIPT>
var pos=document.URL.indexOf("eid=")+4;
document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>
```

Example 2: Consider the HTML form:

```
<div id="myDiv">
Employee ID: <input type="text" id="eid"><br>
...
<button>Show results</button>
</div>
<div id="resultsDiv">
...
</div>
```

The following jQuery code segment reads an employee ID from the form, and displays it to the user.

```
$(document).ready(function(){
$("#myDiv").on("click", "button", function(){
```

```
var eid = $("#eid").val();
$("resultsDiv").append(eid);
...
});
});
```

These code examples operate correctly if the employee ID from the text input with ID eid contains only standard alphanumeric text. If eid has a value that includes metacharacters or source code, then the code will be executed by the web browser as it displays the HTTP response.

Example 3: The following code shows an example of a DOM-based XSS within a React application:

```
let element = JSON.parse(getUntrustedInput());
ReactDOM.render(<App>
{element}
</App>);
```

In Example 3, if an attacker can control the entire JSON object retrieved from getUntrustedInput(), they may be able to make React render element as a component, and therefore can pass an object with dangerouslySetInnerHTML with their own controlled value, a typical cross-site scripting attack.

Initially these might not appear to be much of a vulnerability. After all, why would someone provide input containing malicious code to run on their own computer? The real danger is that an attacker will create the malicious URL, then use email or social engineering tricks to lure victims into visiting a link to the URL. When victims click the link, they unwittingly reflect the malicious content through the vulnerable web application back to their own computers. This mechanism of exploiting vulnerable web applications is known as Reflected XSS.

As the example demonstrates, XSS vulnerabilities are caused by code that includes unvalidated data in an HTTP response. There are three vectors by which an XSS attack can reach a victim:

- Data is read directly from the HTTP request and reflected back in the HTTP response. Reflected XSS exploits occur when an attacker causes a user to supply dangerous content to a vulnerable web application, which is then reflected back to the user and executed by the web browser. The most common mechanism for delivering malicious content is to include it as a parameter in a URL that is posted publicly or emailed directly to victims. URLs constructed in this manner constitute the core of many phishing schemes, whereby an attacker convinces victims to visit a URL that refers to a vulnerable site. After the site reflects the attacker's content back to the user, the content is executed and proceeds to transfer private information, such as cookies that might include session information, from the user's machine to the attacker or perform other nefarious activities.

- The application stores dangerous data in a database or other trusted data store. The dangerous data is subsequently read back into the application and included in dynamic content. Persistent XSS exploits occur when an attacker injects dangerous content into a data store that is later read and included in dynamic content. From an attacker's perspective, the optimal place to inject malicious content is in an area that is displayed to either many users or particularly interesting users. Interesting users typically have elevated privileges in the application or interact with sensitive data that is valuable to the attacker. If one of these users executes malicious content, the attacker may be able to perform privileged operations on behalf of the user or gain access to sensitive data belonging to the user.

- A source outside the application stores dangerous data in a database or other data store, and the dangerous data is subsequently read back into the application as trusted data and included in dynamic content.

## Recommendations:

The solution to prevent XSS is to ensure that validation occurs in the required places and that relevant properties are set to prevent vulnerabilities.

Because XSS vulnerabilities occur when an application includes malicious data in its output, one logical approach is to validate data immediately before it leaves the application. However, because web applications often have complex and intricate code for generating dynamic content, this method is prone to errors of omission (missing validation). An effective way to mitigate this risk is to also perform input validation for XSS.

Web applications must validate all input to prevent other vulnerabilities, such as SQL injection, so augmenting an application's existing input validation mechanism to include checks for XSS is generally relatively easy. Despite its value, input validation for XSS does not take the place of rigorous output validation. An application might accept input through a shared data store or other trusted source, and that data store might accept input from a source that does not perform adequate input validation. Therefore, the application cannot implicitly rely on the safety of this or any other data. This means that the best way to prevent XSS vulnerabilities is to validate everything that enters the application and leaves the application destined for the user.

The most secure approach to validation for XSS is to create an allow list of safe characters that can appear in HTTP content and accept input composed exclusively of characters in the approved set. For example, a valid username might only include alphanumeric characters or a phone number might only include digits 0-9. However, this solution is often infeasible in web applications because many characters that have special meaning to the browser must be considered valid input after they are encoded, such as a web design bulletin board that must accept HTML fragments from its users.

A more flexible, but less secure approach is to implement a deny list, which selectively rejects or escapes potentially dangerous characters before using the input. To form such a list, you first need to understand the set of characters that hold special meaning for web browsers. Although the HTML standard defines which characters have special meaning, many web browsers try to correct common mistakes in HTML and might treat other characters as special in certain contexts. This is why we do not recommend the use of deny lists as a means to prevent XSS. The CERT(R) Coordination Center at the Software Engineering Institute at Carnegie Mellon University provides the following details about special characters in various contexts [1]:

In the content of a block-level element (in the middle of a paragraph of text):

- "<" is special because it introduces a tag.

- "&" is special because it introduces a character entity.

- ">" is special because some browsers treat it as special, on the assumption that the author of the page intended to include an opening "<", but omitted it in error.

The following principles apply to attribute values:

- In attribute values enclosed in double quotes, the double quotes are special because they mark the end of the attribute value.

- In attribute values enclosed in single quote, the single quotes are special because they mark the end of the attribute value.

- In attribute values without any quotes, white-space characters, such as space and tab, are special.

- "&" is special when used with certain attributes, because it introduces a character entity.

In URLs, for example, a search engine might provide a link within the results page that the user can click to re-run the search. This can be implemented by encoding the search query inside the URL, which introduces additional special characters:

- Space, tab, and new line are special because they mark the end of the URL.

- "&" is special because it either introduces a character entity or separates CGI parameters.

- Non-ASCII characters (that is, everything greater than 127 in the ISO-8859-1 encoding) are not allowed in URLs, so they are considered to be special in this context.

- The "%" symbol must be filtered from input anywhere parameters encoded with HTTP escape sequences are decoded by server-side code. For example, "%" must be filtered if input such as "%68%65%6C%6C%6F" becomes "hello" when it appears on the web page.

Within the body of a <SCRIPT> </SCRIPT>:

- Semicolons, parentheses, curly braces, and new line characters must be filtered out in situations where text could be inserted directly into a pre-existing script tag.

Server-side scripts:

- Server-side scripts that convert any exclamation characters (!) in input to double-quote characters (") on output might require additional filtering.

Other possibilities:

- If an attacker submits a request in UTF-7, the special character '<' appears as '+ADw-' and might bypass filtering. If the output is included in a page that does not explicitly specify an encoding format, then some browsers try to intelligently identify the encoding based on the content (in this case, UTF-7).

After you identify the correct points in an application to perform validation for XSS attacks and what special characters the validation should consider, the next challenge is to identify how your validation handles special characters. If special characters are not considered valid input to the application, then you can reject any input that contains special characters as invalid. A second option is to remove special characters with filtering. However, filtering has the side effect of changing any visual representation of the filtered content and might be unacceptable in circumstances where the integrity of the input must be preserved for display.

If input containing special characters must be accepted and displayed accurately, validation must encode any special characters to remove their significance. A complete list of ISO 8859-1 encoded values for special characters is provided as part of the official HTML specification [2].

Many application servers attempt to limit an application's exposure to cross-site scripting vulnerabilities by providing implementations for the functions responsible for setting certain specific HTTP response content that perform validation for the characters essential to a cross-site scripting attack. Do not rely on the server running your application to make it secure. For any developed application, there are no guarantees about which application servers it will run on during its lifetime. As standards and known exploits evolve, there are no guarantees that application servers will continue to stay in sync.

## Tips:

1. The Fortify Secure Coding Rulepacks warn about SQL Injection and Access Control: Database issues when untrusted data is written to a database and also treat the database as a source of untrusted data, which can lead to XSS vulnerabilities. If the database is a trusted resource in your environment, use custom filters to filter out dataflow issues that include the DATABASE taint flag or originate from database sources. Nonetheless, it is often still a good idea to validate everything read from the database.

2. Even though URL encoding untrusted data protects against many XSS attacks, some browsers (specifically, Internet Explorer 6 and 7 and possibly others) automatically decode content at certain locations within the Document Object Model (DOM) prior to passing it to the JavaScript interpreter. To reflect this danger, the Rulepacks no longer treat URL encoding routines as sufficient to protect against cross-site scripting. Data values that are URL encoded and subsequently output will cause Fortify to report Cross-Site Scripting: Poor Validation vulnerabilities.

3. Older versions of React are more susceptible to cross-site scripting attacks by controlling an entire component. Newer versions use Symbols to identify a React component, which prevents the exploit, however older browsers that do not have Symbol support (natively, or through polyfills), such as all versions of Internet Explorer, are still vulnerable. Other types of cross-site scripting attacks are valid for all browsers and versions of React.

## Button.js, line 14 (Cross-Site Scripting: DOM)

| | | | |
|---|---|---|---|
| **Fortify Priority:** | Critical | **Folder** | Critical |
| **Kingdom:** | Input Validation and Representation | | |
| **Abstract:** | The method Button() in Button.js sends unvalidated data to a web browser on line 14, which can result in the browser executing malicious code. | | |

**Source:** CloudAdminSpecificCatalog.js:31 ~JS_Generic.useParams()

```
29
30  const CloudAdminSpecificCatalogContext = () => {
31      const { type, id } = useParams();
32      const history = useHistory();
```

**Sink:** Button.js:14 Assignment to onClick()

```
12                      : 'btn btn-secondary')
13                  }
14                  onClick={onClickHandler}
15                  {...(rest && rest)}
16              >
```

## Button.js, line 14 (Cross-Site Scripting: DOM)

| | | | |
|---|---|---|---|
| **Fortify Priority:** | Critical | **Folder** | Critical |
| **Kingdom:** | Input Validation and Representation | | |
| **Abstract:** | The method Button() in Button.js sends unvalidated data to a web browser on line 14, which can result in the browser executing malicious code. | | |

**Source:** EditAppStack.js:38 ~JS_Generic.useParams()

```
36          Inspector: 'DeployInspector'
37      };
38      const { type, id, appid } = useParams();
39      const history = useHistory();
40      const { data: catalogDetails } = getQuery(GET_SPECIFIC_CATALOG_DETAILS, {
```

**Sink:** Button.js:14 Assignment to onClick()

```
12                      : 'btn btn-secondary')
13                  }
14                  onClick={onClickHandler}
15                  {...(rest && rest)}
16              >
```

## Button.js, line 14 (Cross-Site Scripting: DOM)

| | | | |
|---|---|---|---|
| **Fortify Priority:** | Critical | **Folder** | Critical |
| **Kingdom:** | Input Validation and Representation | | |
| **Abstract:** | The method Button() in Button.js sends unvalidated data to a web browser on line 14, which can result in the browser executing malicious code. | | |

**Source:** CreateAppStack.js:41 ~JS_Generic.useParams()

```
39          Inspector: 'DeployInspector'
40      };
41      const { type, id } = useParams();
42      const history = useHistory();
43      const { data: catalogDetails } = getQuery(GET_SPECIFIC_CATALOG_DETAILS, {
```

**Sink:** Button.js:14 Assignment to onClick()

```
12                      : 'btn btn-secondary')
13                  }
```

```
14                          onClick={onClickHandler}
15                          {...(rest && rest)}
16                      >
```

## Button.js, line 14 (Cross-Site Scripting: DOM)

| | | | |
|---|---|---|---|
| **Fortify Priority:** | Critical | **Folder** | Critical |
| **Kingdom:** | Input Validation and Representation | | |
| **Abstract:** | The method Button() in Button.js sends unvalidated data to a web browser on line 14, which can result in the browser executing malicious code. | | |

**Source:** CloudAdminViewAppStack.js:28 ~JS_Generic.useParams()

```
26      access
27  }) {
28      const { type, id } = useParams();
29      const history = useHistory();
30      const [searchAppStackData, setSearchAppStackData] = useState([]);
```

**Sink:** Button.js:14 Assignment to onClick()

```
12                          : 'btn btn-secondary')
13                  }
14                  onClick={onClickHandler}
15                  {...(rest && rest)}
16              >
```

## Button.js, line 14 (Cross-Site Scripting: DOM)

| | | | |
|---|---|---|---|
| **Fortify Priority:** | Critical | **Folder** | Critical |
| **Kingdom:** | Input Validation and Representation | | |
| **Abstract:** | The method Button() in Button.js sends unvalidated data to a web browser on line 14, which can result in the browser executing malicious code. | | |

**Source:** EditCatalog.js:21 ~JS_Generic.useParams()

```
19  function EditCatalog() {
20      const history = useHistory();
21      const { type, id } = useParams();
22      const [selectedCatalog, setSelectedCatalog] = useState({});
23      const [showEditErrDialog, setShowEditErrDialog] = useState(false);
```

**Sink:** Button.js:14 Assignment to onClick()

```
12                          : 'btn btn-secondary')
13                  }
14                  onClick={onClickHandler}
15                  {...(rest && rest)}
16              >
```