**Patching & Remediation**

The web application we reviewed was relatively securely coded, apart from the eval injection vulnerability we identified. We will consider the authentication functions to be safe, as we could not identify clear vulnerabilities apart from the password-less authentication, which is intended for the purposes of this module, as we discussed earlier.

While our first recommendation may be to avoid using eval altogether, this should not be our only recommendation, as there are many occasions where the web application relies on this function, so we would need to provide a safer alternative. Similarly, there may be certain cases where modifying the vulnerable code may not be feasible, so we would need to provide extra security measures to patch the vulnerability without modifying the original code.

**Alternatives**

The vulnerability was found in the onError parameter of the validateString function used within eval. There are two possible alternative approaches to achieve the same result while maintaining the same function struggle (i.e. keeping onError as the second parameter).

**Function**

The first "recommended" alternative is to use onError as a function instead of a string passed to eval. While JavaScript is a dynamically typed language, meaning we cannot specify a strict type for onError, the function may still be coded with that basis by simply modifying eval(onError) to onError().

The main modification would be when the function is called, like the example in the generateQR function. When called, the onError parameter needs to be written as a function, as follows:

Code: js

!validateString(text, function () {

throw {

message:

role === "admin"

? `The input "${text}" contains the following invalid characters: [${text.match(

/['"`;]/g

)}]`

: "Invalid input",

statusCode: 403,

};

});

As we can see, now the user input is limited to the message parameter string instead of going into an eval call, so an injection would not be possible.

**Safe-Eval**

If the use of eval was necessary, which could be the case for multiple reasons, then we may recommend the use of a safe alternative of eval, such as the [safe-eval](https://www.npmjs.com/package/safe-eval) package found on npm. Such packages attempt to run the eval string within a sandbox, which should "theoretically" prevent the execution of any code on the backend server.

However, when we install safe-eval by adding it to the package.json file, we get notified that this function has a critical vulnerability:

Patching & Remediation

yovecio@htb[/htb]$ npm install safe-eval -S

added 1 package, and audited 193 packages in 2s

...SNIP...

1 critical severity vulnerability

Running npm audit shows multiple public PoCs of sandbox bypass and arbitrary code execution, even though we are running the latest available version:

Patching & Remediation

yovecio@htb[/htb]$ # npm audit report

safe-eval \*

Severity: critical

Sandbox Breakout / Arbitrary Code Execution in safe-eval - https://github.com/advisories/GHSA-9pcf-h8q9-63f6

safe-eval vulnerable to Prototype Pollution - https://github.com/advisories/GHSA-33vh-7x8q-mg35

safe-eval vulnerable to Prototype Pollution via the safeEval function - https://github.com/advisories/GHSA-hcg3-56jf-x4vh

safe-eval vulnerable to Sandbox Bypass due to improper input sanitization - https://github.com/advisories/GHSA-79xf-67r4-q2jj

Sandbox Breakout / Arbitrary Code Execution in safe-eval - https://github.com/advisories/GHSA-hrpq-r399-whgw

No fix available

node\_modules/safe-eval

1 critical severity vulnerability

It is clearly mentioned that no fix is available for this package, and reading the references shows payloads that we may use to exploit it. This is why it is always recommended to avoid using eval whenever possible. However, as mentioned earlier, there may be some cases where eval must be used, and for such cases, we recommend the use of safe-eval in addition to the extra security measures we will go through next.

**Challenge:** If you want to test your advanced code injection skills further, try to install safe-eval as shown above, and replace eval on line 9 with safeEval. Then, try to refer to the above PoC links to achieve code injection and, ultimately, command execution.

**Extra Security Measures**

Whether the above alternatives are used or not, we should always recommend the user input be both validated and sanitized of any unnecessary characters.

**Sanitization**

Whenever we need to sanitize any user input, we must limit it to the required characters and remove any other ones. The allowed character set varies from one function to another, so our recommendation would vary depending on the input type. For our use case, we should add a sanitization function to remove special characters from the user input before passing it to input validation in validateString.

We can do so through multiple third-party packages, such as [dompurify](https://www.npmjs.com/package/dompurify) (to remove HTML elements, usually front-end JavaScript), or [sanitize](https://www.npmjs.com/package/sanitize). To sanitize the text variable, we will first add the middleware to routes/service-routes.js before the generateQR endpoint:

Code: js

router.use(verifyToken);

router.use(require("sanitize").middleware);

router.post("/generate", generateQR);

After that, when obtaining text from req.body within generateQR, we can specify the middleware we need for sanitization, such as bodyString, as follows:

Code: js

const text = req.bodyString("text");

This should sanitize the text input to ensure it remains a string. While this may be useful for general cases, we can always add a line of code to simply remove the unwanted characters from text, as follows:

Code: js

const sanitizedText = text.replace(/['"`;]/g, "");

Either of these options would work, depending on the use case we are dealing with. We recommend using the latter for this web application as it gets the job done without relying on third-party packages.

**Validation**

Unlike sanitization, where we would simply remove certain characters, validation requires a thorough understanding of how user input should look and rejects it otherwise. This is why we would always recommend using third-party packages for user input validation, as validation is not usually easily coded effectively with a few lines of code. We may have various types of input, and each would need its function, which may not be feasible to be coded manually.

While a validation attempt was made through the validateString function, the vulnerability was found within that function, so the validation attempt was not applied for that specific vulnerability. A better way would be to use the [validator](https://www.npmjs.com/package/validator) or [Yup](https://www.npmjs.com/package/yup#yup) packages. Both of these packages provide various built-in types and patterns.

To validate the text input, we can create a basic schema with yup and specify the regular expression we want, as follows:

Code: js

let schema = yup

.string()

.required()

.matches(/^[^"\';`]\*$/);

schema.validateSync(text);

This will not only ensure that the text matches the specified pattern but also that it is not missing (i.e. required()). Furthermore, we can customize the error message depending on the user role, as needed by our application:

Code: js

let schema = yup

.string()

.required()

.matches(/^[^"\';`]\*$/, role === "admin" ? null : "Invalid input");

schema.validateSync(text);

Now, the schema would return the default verbose error message if the role is admin, automatically specifying what was wrong with the input (i.e. the specified regex pattern not met). Otherwise, it would simply return Invalid input, thus fully meeting the intended use of the previous validateString function.

We can similarly suggest replacing the validateEmail function with a yup schema, as follows:

Code: js

const validateEmail = (email) => {

const schema = yup.string().email().required();

return schema.isValidSync(email);

};

These packages are extensive and can cover much larger applications, but they can also apply to smaller ones, as we saw above, so this would be our recommendation.

**Exercise:** Try to apply all patches provided in this section, and then re-run the PoC exploit script to ensure the web application is no longer vulnerable. Finally, test all modified functionalities to ensure they still function as expected.