Secure speedup of the future

Aitor Ruiz Garcia

September 2023

1 Abstract

JavaScript has emerged as a fundamental language in global development. Much of the success of Node.js, both in server-side applications and full-stack web applications (client-server), can be attributed to it.

Node.js was primarily created to enable JavaScript to run on the server. The creator of Node never anticipated that JavaScript would be used for literally everything.

JavaScript is employed in desktop applications using Electron, and in mobile applications using React Native or Ionic. As an interpreted language, it has a low entry barrier. As the default language of the web, it also boasts a wealth of interesting libraries.

Node JS took the engine that runs JavaScript on Chrome, V8, and wrapped it in more C++ code to make it possible for it to interact with system calls.

All these factors have contributed to the rise of JavaScript as the language of the future. Nonetheless, as the creator himself contends, it has experienced some 'growing pains'.

2 The problem

Ryan Dahl, the creator of Node.js, has expressed his regrets about the language. He has stated that he would not use JavaScript for a new project. He has also stated that he would not use Node.js for a new project. Node.js has grown past what it was intended to be, and it has become a 'monster'.

Whyle experimenting with my TFM project, I have found some problems with the current state of the JavaScript ecosystem. I will explain them in the following sections. This TFM project was a web application with blockchain integrations for decentralized identity management. Even though this project is not in the

scope of this TFM, it is important to understand the context of the problems that I have found.

2.1 No types

JavaScript, does not have a type checker. This means that the compiler does not check the types of the variables, and therefore, it is possible to assign a value of one type to a variable of another type. This can lead to unexpected errors in the code.

Take a look at the following example:

```
const user = {
    name = "",
    age = 0
}

console.log(user.email)
```

This small code will not rise any error to warn the developer that the user object does not have an email property. This lead to a runtime error, which is not good for the developer experience.

It is even worse when the user recives the error in production, because the error will be shown to the user, and the user will not understand what is happening.

This happens because JavaScript is a dynamic language, and it does not have a type checker.

The original TFM used Next.js. A popular React-in-the-server framework. At the start of the project, I was not aware of the problems that I will explain in the following sections. I was using Next.js because it was the framework that I was more familiar with. Obviuslly, as the project grew, I started to have problems with the lack of types.

2.2 node_modules

Node.js has a package manager called npm. This package manager is used to install libraries in the project. The problem is that the libraries are installed in the node_modules folder, and this folder can grow a lot.

By default npm does not softlinks the libraries between them. This means that if you have two libraries that use the same library, the library will be installed twice. More packages installed means more space used, and more time to install the packages. A CI build could take a lot of time to just install all the packages.

2.2.1 node_modules security implications

As JavaScript is used in Web applications mainlly and both in the server and in the client, it is a good target to attack. Currentlly npm holds more that 1.6 millions packages.

A sussesfull attack on a web framework would mean access a big

chunk of browsers that a small porcentaje of them may be vulnerable.

However the JavaScript in a browser is sandboxed, it is *safer*. In Node.js this is not the case. All the code can interact with the file system and the whole internet.

In the past there has been multiple cases of compromised npm, the most famous being `colors` and `faker.js`.

The developer went rogue and introduced some infinite loops...

Another notable example was `UAParser.js`, and it downloaded and installed a password stealer and a cryptominer. It is important to note that this package was and still is used by millions of users dayly.

2.3 Node GYP

Node GYP is the state of the art when creating native libraries in Node JS. This tool is originally from google. When google discontinuated it the comunity made a fork and the project continued.

It is a tool writen in python, witch is not JavaScript, and it is inecessarilly complicated to write a native library.

3 The solution

Ryan Dahl, created Deno, a solution to his own issues created by working on Node JS without a clear plan.

3.1 TypeScript

TypeScript appeared in 2012 to add types to JavaScript. It works as a superset of JavaScript. Meaning all keywords would be valid in TypeScript but vicebersa.

TypeScript does not work by changing V8, but making JavaScript the compile target of TypeScript. This allowed also targeting multiple versions of JavaScript and the use of polyfils if the desired API was not present.

However TypeScript in this form, while very helpfull, it add more time to the development process because it now has to compile TypeScript to JavaScript and then run the intended program.

Deno, translates TypeScript on the fly and feeds it to V8, meaning the whole process is just plug and play and developers do not have to configure an external tool, as `tsc`, the compiler, can be pretty hard to set up and get it to behave.

In the code, this mens that every variable would have an associated type and they could not change anywhere in the code.

```
var name: string = "";
```

The variable name can not have another value asignated without being a string.

```
name = 0;
```

This code will result in error as `0` is not a string.

This will catch multiple runtime errors.

3.2 URL based imports

A major change with Deno comes with the URL based imports the modules are cached and reused when needed.

However with this as the script comes from an URL it may come with malicius code in it. How could a developer protect itself?

3.2.1 Security checks

Deno implements multiple checks to prevent uncontrolled access to the machine.

-

3.3 Rust based FFI lib

4 The data

4.1 Why not WASM?