Prototype pollution via console properties in Node.js

IT20021870 – Dilshan K.N
Department of Computer Systems Engineering
Sri Lanka Institute of Information Technology,
New Kandy Rd, Malabe, Sri Lanka
it20021870@my.sliit.lk

***Abstract* — Node.js is popular, free, Open-source, and Lightweight JavaScript web framework which can be deployed in different platforms (Unix, Linux, Windows, and Mac OS). Without Node.js, there was not much of choices for beginners to develop backend servers and databases for data-intensive applications such as real time and streaming applications. More or the same Node.js can be integrated to develop all the component like front-end, middleware and the back end by using web development stacks such as MEAN stack, MERN stack and MEVN stacks.**

The "Console'' is a module provided by Node.js that can be used to debugging purposes with the help of different methods. "console.table" is such kind of method, but due to the formatting logic of the "console.table()" function it was not safe to allow user controlled input to be passed to the "properties" parameter while simultaneously passing a plain object with at least one property as the first parameter, which could be lead to access or update the "_proto_", constructor and protocol. Which means injecting properties into existing JavaScript language construct prototypes, such as objects or simply defined as "Prototype Pollution attack" could lead to alter a root level access or root level prototypes of any objects.

The tools like, Retire.js, WhiteSource Renovate, OWASP Dependency-Check and OSS INDEX can be used to identify the known vulnerabilities available in the Node.js. Since prototype pollution has very limited control, "object.freeze" can be used for freeze the objects, schema validation make sure that no _proto_ contain in JSON data. Some other versions of Node.js assign object.create(null), a null prototype for the object these properties

Keywords— Node.js, proto, vulnerabilities, Console, console.table, properties, Prototype Pollution attack

I. INTRODUCTION

JavaScript is a famous programming language that is extensively used in multiple systems, including Node.js. That has many dynamic and adaptive features like, prototype-based, which implies that for a definition, every property lookup does not end with the current object but instead travels farther up to explore a chain of archetypal objects, known as a prototype chain. Addition to that another dynamic feature of JavaScript is that it allows for flexible redefinitions of almost all objects, including built-in, functions. Because of those dynamic additional options, malicious actors can make application-wide changes to all objects by modifying object, The combination of two dynamic features mentioned above leads to prototype pollution, a new type of object-related vulnerability [1].

An adversary uses vulnerable property lookups to traverse the prototype chain of the underlying object and then redefines a built-in function. Consider the following illustration: Consider a susceptible statement that has two property lookups and an assignment, such as obj[a][b] =c. If an enemy has authority over a, b, and c, the attacker can utilize obj [" proto__"]. To alter the built-in function Object.prototype.toString, use ["toString"]="hack." As according to previous research, the consequences of prototype pollution are severe, including Denial-of-Service (DoS), arbitrary code execution, and session fixation.

II. RESEARCH STATEMENT

This study covers the literature on Prototype pollution via console.table properties in Node.js. Also, this paper covers how the Prototype pollution works, Node.js properties and addresses how the specified technology has been advanced for provide the resilience for such attacks and future advances in the field. Moreover, the most typical flaws with threats and how they can affect the broader domain will be reviewed. Most recent and emphasized attack scenarios, attack landscapes, and indicate realistic remedies will also be addressed focusing on future resolutions that will make the Node.js environment more secure will be presented.

III. REVIEW OF LITERATURE

1. Node.js

In early 2009, Ryan Dahl came up with the concept for Node. He was the creator of Web Machine before that. That was Google Chrome's very first online application framework. The idea started off as a simple experiment. Despite this, it gained popularity among engineers looking for new ways to create high-performance programs. This is without accounting for the complexities of traditional languages [4].

Node.js is a free and open-source server environment that operates on a variety of platforms

(Windows, Linux, Unix, Mac OS X, etc.) and JavaScript is used on the server [2]. The V8 engine runs the JavaScript runtime environment, which enables JavaScript code to be executed outside of a web browser. Node.js allows developers to use JavaScript to construct command-line tools and server-side scripting, which involves running scripts on the server to generate dynamic web page content before sending the page to the user's browser. As a result, Node.js represents a "JavaScript everywhere" paradigm, uniting web-application development under a single programming language rather than two separate languages for server-side and client-side scripts. The event-driven architecture of Node.js allows for asynchronous I/O as well [3].

Node is distinguished by several characteristic, what exactly would that be,

1. Google Chrome V8 JavaScript Engine

The Google Chrome V8 JavaScript runtime engine serves as the foundation for this runtime environment. The engine accepts JavaScript and converts it to readable code in the same way that a Java Virtual Machine does.

2. Modules/Packages

"Npm", a node package manager with over 350,000 packages, is included with Node.js to help you get your project or application up and running quickly and easily [4].

3. Event Driven, Single-Threaded I/O Model

JavaScript relies on user interactions or events to function. Most of the code is run synchronously. A system of promises or async/await functions is used to handle these inputs and outputs, server requests, and other asynchronous operations.

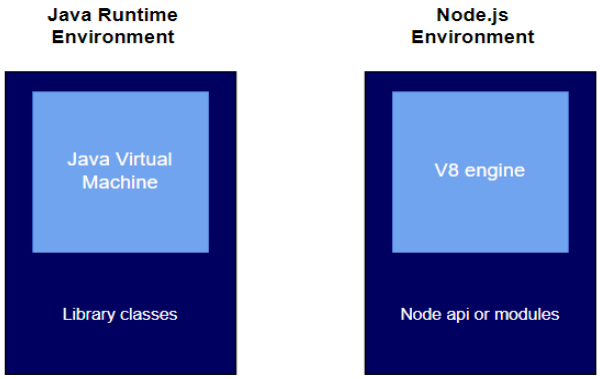

Figure 1.1

i. Features of Node.js

- Node.js is a good solution for developers that need to write basic code to create fast, scalable network applications. They can develop web servers and other backend technologies for use in mobile apps and websites [5].

- Node is designed to support real-time web applications. As a result, that has an advantage over competing languages and frameworks.

- Because of functional reactive programming, asynchronous callbacks can be employed in applications.

- Microsoft, Amazon Web Services, and Heroku, for example, have a strong community of Node contributors.

- Node.js JavaScript execution is made simple on the server side and that has not limited to PC and mobile apps.

- Merging with different tools with Node, facilitate the use of existing HTML, CSS and JS skillset for develop powerful web apps, games and other real time applications

2. Node.js Console & Buffer

The console is a Node.js module that works similarly to the JavaScript console that appears when analyzing a webpage in the browser. The console gives variety of debugging options that developers can use.

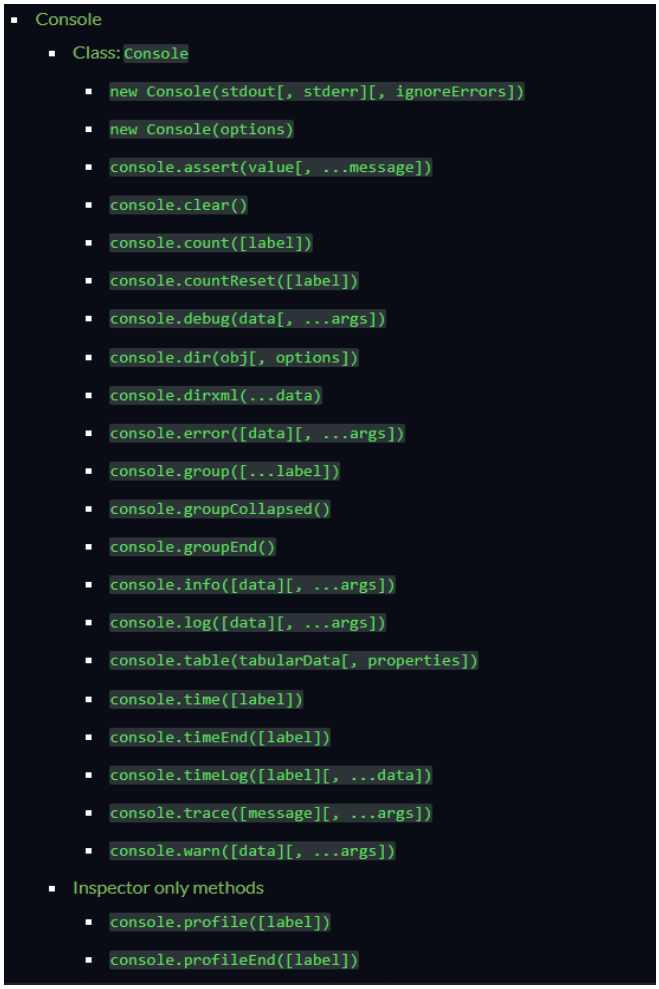

Figure 2.1

The console module exports two unique components:

- A Console class that can write to any Node.js stream using methods such as console.log(), console.error(), and console.warn() ().
- A global console instance uses process.stdout and process.stderr. You can use the global console without having to call require('node:console').

Warning: The global console object's methods are not always synchronous, like the browser APIs they mimic, and they are not always asynchronous, like all other Node.js streams. See the note on process I/O for more information [6].

3.Classes, Objects and Prototype

i. Classes

In the modern JavaScript, there is a more advanced "class" construct, that introduces great new features that are useful for object-oriented programming [7]. In Node.js, the concept of a class begins with a function. The constructor of the class is the function itself.

The class syntax has two components:

1. Class expressions
2. Class declarations

```
// Class expression
let class_name  = class {
 constructor(method1, method2) {
    this.method1 = method1;
    this.method2= method2;
  }
};

// Class declaration
class class_name {
  constructor(method1, method2) {
    this.method1 = method1;
    this.method2= method2;
  }
}
```

Figure 3.1

In any case, in JavaScript, users do not need to define a class to construct an object. Users only need to define properties using curly bracket notation.

ii. Objects

A key-value pair can be thought of as an object, with the key being a string and the value being anything. Everything users type in JavaScript is an Object, with the exception of primitives.

. Example 01 :- const obj = {
property_1: 111,
property_2: 222,
}

iii. Prototype

Prototype is an Object attribute that allows JavaScript Objects to inherit features from one another. Because almost everything in JavaScript is an Object, the prototype is also an Object. An Objects Prototype may also have a Prototype, and from it, it can inherit his Prototype or other attributes, and so on. This is referred to as a prototype chain [8].

This "Example 01" object has two properties: property_1 and property_2. These aren't the only properties users have access to, though. For example, using obj.toString() would return "[object Object]". toString is provided by the prototype (along with a few other default members). There is a prototype for every JavaScript object (it can also be null). If the developer doesn't specify otherwise, an object's prototype is Object.prototype by default.

4. Magic Properties

i. _proto_

The special attribute _proto_ refers to all of an object's prototypes and all Objects have __proto__ as their attribute. As same as, _proto_ that also a prototype itself. Since the "__proto__" is a magical property that yields the object's class's "prototype." Even though this isn't a standard property in JavaScript, that fully supported in the NodeJS environment. That is worth mentioning that this property is implemented as a getter/setter property that, on read/write, calls getPrototypeOf and setPrototypeOf. As a conclusion, __proto__ was clearly intended to be a feature, supporting operations such as inheritance of all attributes from a JavaScript class. However, _proto_ logic 's gradually evolved a weakness, or to be more precise, a vulnerability.

```
function MyClass() {

}

MyClass.prototype.myFunc = function () {
      return 7;
}

var inst = new MyClass();
inst.__proto__ // returns the prototype of MyClass
inst.__proto__.myFunc() // returns 7

inst.__proto__ = { "a" : "123" }; // changing the prototype at runtime.
inst.hasOwnProperty("__proto__") // false. We haven't redefined the
property. It's still the original getter/setter magic property
```

Figure 4.1

ii. Constructor

The magical attribute "constructor" returns the object's creation function. It seems to be worth noting that each constructor contains a "prototype" attribute that refers to the class's prototype.

```
function MyClass() {

}

MyClass.prototype.myFunc = function () {
    return 7;
}

var inst = new MyClass();
inst.constructor //  returns the function MyClass
inst.constructor.prototype // returns the prototype of MyClass
inst.constructor.prototype.myFunc() // returns 7
```

Figure 4.2

5. Prototype Pollution Attack

Prototype pollution vulnerabilities occur when the code of the application allows the alteration of any prototype properties, usually those of the Object prototypes [9]. When an attacker manipulates __proto__, usually by placing a new Prototype upon __proto__. This addition is sent down the prototype chain to all JavaScript Objects because __proto__ exists for every Object and every Object inherits Prototypes from its Prototype. Malicious gamers could leverage the ability to insert properties into existing JavaScript code to perform Denial of Service attacks or Remote Code Execution attacks by inserting malicious code.

Example :- In the code snippet below, the toString() method of an object is hijacked and then all objects get polluted [10].

```
// you can play with the following code snippet in browser console.

let person = {name: 'John Doe'}
console.log(person.name)
// John Doe

person.__proto__.toString = () => {alert('evil')}
console.log(person.name)
// an alert box with "evil" pops up

let person2 = {}
console.log(person2)
// {}
// an alert box with "evil" pops up

// if you stay in the browser,
// clicking any place in the page, you will see a new alert box pop up...
```

Figure 5.1

6. Threat Modeling

i. Proto Scanner

The proto scanner is a tool that may be used to find prototype pollution flaws in Node.js packages. Python and JavaScript were used to create this utility. "Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao" are the initial developers of the tool and the author of this research paper contribute to develop the tool with additional features and then rename the new tool version as "proto scanner".

- Requirements
  - npm install
  - python 3
  - pip install

- Usage

The main function of the tool runs with the help of imported python scripts [OpGen.py, Graph.py and parse_args.py].

```
from src.core.opgen import OPGen
from src.core.graph import Graph
# from src.core.options import parse_args, setup_graph_env
from src.core.options import parse_args

if __name__ == '__main__':
    print('''

    ─────────────────────────── IT20021870 ───────────────────────────

    -
    -
    -                    Node.js  Prototype Pollution Scanner
    -                                                               -
    -                                                               -
    ──────────────────────────────────────────────────────────────────

            ''')
    opg = OPGen()
    opg.run()

print ('\n────────── Thanks For Using PROTO SCANNER ──────────')
```

Figure 6.1

- python3 ./protoscanner.py –help

```
Object graph generator for JavaScript.

positional arguments:
  input_file            Source code file (or directory) to generate object graph for. Use '-' to get source code from stdin. Ignore this argument to analyze ./nodes.csv and ./rels.csv.

optional arguments:
  -h, --help            show this help message and exit
  -p, --print           Print logs to console, instead of file.
  -m, --module          Module mode. Regard the input file as a module required by some other modules.
  -q, --exit            Exit the program when vulnerability is found.
  -s, --single-branch   Single branch. Do not create multiple possibilities when meet a branching point.
  -a, --run-all         Run all exported functions in module.exports. By default, only main functions will be run.
  -f FUNCTION_TIMEOUT, --function-timeout FUNCTION_TIMEOUT
                        Time limit when running all exported function, in seconds. (Defaults to no limit.)
  --timeout TIMEOUT     Time limit for testing an entrance. (Defaults to None)
  -c CALL_LIMIT, --call-limit CALL_LIMIT
                        Set the limit of a call statement. (Defaults to 3.)
  -e ENTRY_FUNC, --entry-func ENTRY_FUNC
  -l LIST, --list LIST
  --install             If set, we will install the packages to the run env
  --max-rep MAX_REP     If set, it will limit of the max time of calls of each function in the call stack to max-rep
  --run-all-files       If set, it will run all files of a package
  --no-prioritized-funcs
                        If set, it will not try to run prioritized functions before everything
  --entrance-func ENTRANCE_FUNC
                        If set, it will start from a specified function
  --pre-timeout PRE_TIMEOUT
                        timeout for pre-processing file (for entrance func set only)
  --exported-func-timeout EXPORTED_FUNC_TIMEOUT
                        timeout for single exported function
  --no-exports          if set, it will never run exported functions
  --max-file-stack MAX_FILE_STACK
                        If set, it will limit the max size of file stack to max-file-stack
  --skip-func SKIP_FUNC
                        If set, it will skip a list of functions, separated by ,
  --run-env RUN_ENV     set the running env location
  --no-file-based       No file based detection
  --parallel PARALLEL   run multiple package parallelly
  --auto-type           Auto change the type of wildcard obj based on the called method
```

Figure 6.2

Depending on the using options, users can analyze stored Node.js source codes.

- python3 ./protoscanner.py -m -a --timeout 300 -q ./tests/packages/pp.js

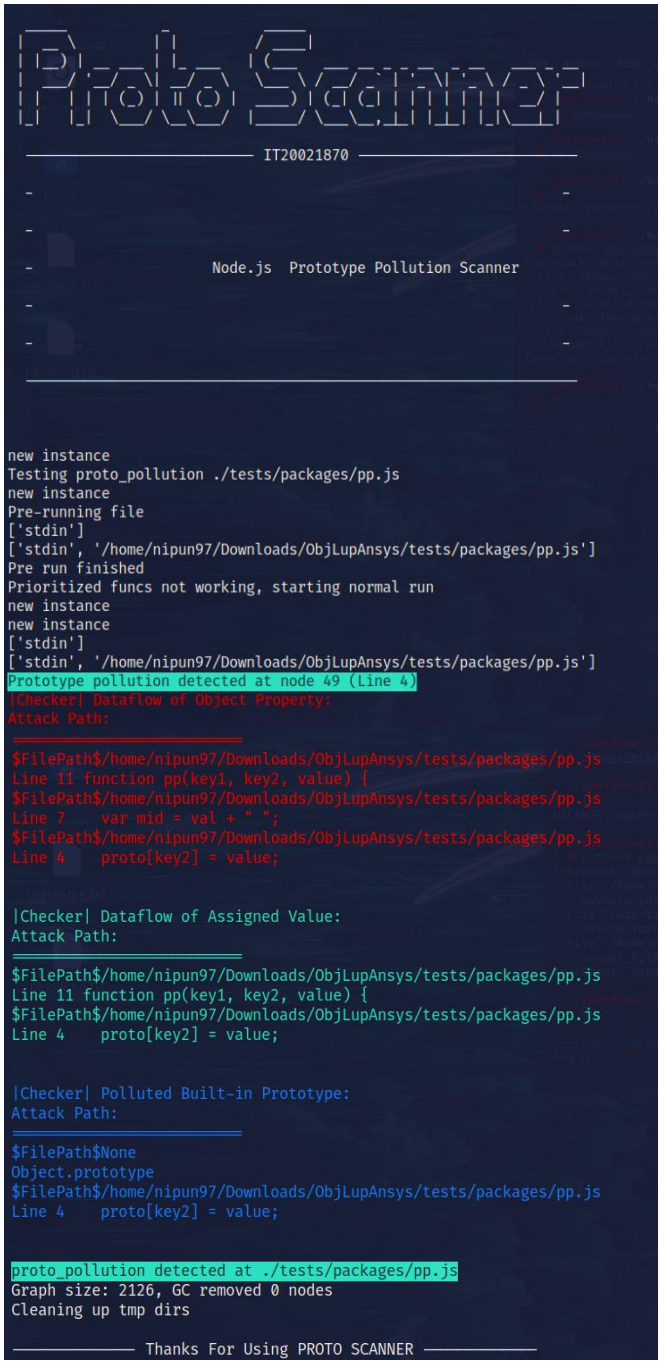

Figure 6.3

pp.js is prototype pollution vulnerable Node.js source code which was stored in /home/nipun97/Downloads/ protoscanner/tests/packages/pp.js. The tool analyzes the source code and detects that pp.js is vulnerable to prototype pollution.

7. Mitigation

i. Freezing the prototype

The ECMAScript standard version 5 introduced a very interesting set of functionalities to the JavaScript language [11]. This allowed for the creation of non-enumerable properties, getters, and setters, among other things. When applying the “Object.freeze”function on an object, any subsequent modifications to that object will fail silently. That's feasible to freeze the prototype of "Object" because it's an object. As a result, practically all exploitable cases will be mitigated.

mitigation.js

1. Object.freeze(Object.prototype);
2. Object.freeze(Object);
3. ({}).__proto__.test = 123
4. ({}).test // this will be undefined

Figure 7.1

ii. Schema validation of JSON input

Several NPM libraries, such as avj, support schema validation for JSON data. Schema validation guarantees that JSON data contains all the expected characteristics and is of the correct type. When employing this method to reduce "prototype pollution" attacks, it is vital to reject extraneous attributes. This is accomplished by setting "additionalProperties" to "false" on the schema in avj.

iii. Using Map instead of Object

The Map primitive was first introduced in EcmaScript 6 [12]. That works in the same way as a HashMap, but without the security restrictions that an Object does. This is now fully supported in newer NodeJS environments, and browsers are rapidly adopting functionality. Map should be used instead of Object when a key/value structure is required.

```
1  const map1 = new Map();
2
3  map1.set('a', 1);
4  map1.set('b', 2);
5  map1.set('c', 3);
6
7  console.log(map1.get('a'));
8  // expected output: 1
9
10 map1.set('a', 97);
11
12 console.log(map1.get('a'));
13 // expected output: 97
14
15 console.log(map1.size);
16 // expected output: 3
17
18 map1.delete('b');
19
20 console.log(map1.size);
21 // expected output: 2
```

Figure 7.2

iv. Object.create(null)

In JavaScript, it's possible to construct objects that don't have a prototype. That is necessary to use the "Object.create" function. The "__proto__" and "constructor" attributes will not be present in objects created using this API. This method of object creation can aid in the reduction of prototype pollution.

```
1. var obj = Object.create(null);
2. obj.__proto__ // undefined
3. obj.constructor // undefined
```

Figure 7.3

IV. FUTURE RESEARCH

DAPP: automatic detection and analysis of prototype pollution vulnerability in Node.js modules [13]. DAPP, which is intended at all Node Package Manager-registered real-world modules. DAPP executes and incorporates a static analysis based on an abstract syntax tree and a control flow graph, and it can detect the recommended patterns in each Node.js module in a matter of seconds. This new study offers a more effective and efficient method of analysis. Researchers conducted multiple empirical tests to evaluate and compare their cutting-edge methodology to previous analytic tools and found that the tool is comprehensive and works well with modern JavaScript syntax.

.

V. CONCLUSION

Dynamic and adaptable Node.js features not only make life easier for web developers, but they also create new vulnerabilities, such as prototype pollution. Many years ago, the Node.js community coined the phrase "prototype pollution" to describe libraries that added new methods to the prototype of base objects like "Object," "String," or "Function." This was rapidly regarded a poor strategy because it resulted in unanticipated application behavior. "Prototype"1 was the most recent big library to use this type of mechanic. Although the library is still open, it is widely assumed to be closed. Since the prototype pollution is a dangerous vulnerability that has to be researched more, both in terms of detecting new vectors and discovering new exploitation. Because the vector also isn't generated on the client until the payload is saved on the server, there is room for further investigation as well.

.

VI. ACKNOWLEDGMENT

The author is grateful to Dr. Lakmal Rupasinghe and Miss. Chethana Liyanapathirana, our lecturers in charge of the Secure Software System module, those who have aided and motivation since the beginning of this project. I'd also like to thank all my colleagues who contributed to the success of this research. Finally, I'd like to express my heartfelt gratitude to everyone who contributed resources and ideas to help me complete this project on time.

VII. REFERENCES

[1] B. Dickson, "Prototype pollution: The dangerous and underrated vulnerability impacting JavaScript applications," https://portswigger.net, 2021. [Online]. Available: https://portswigger.net/daily-swig/prototype-pollution-the-dangerous-and-underrated-vulnerability-impacting-javascript-applications. [Accessed 12 May 2022].

[2] "Node.js Introduction," w3schools.com, [Online]. Available: https://www.w3schools.com/nodejs/nodejs_intro.asp. [Accessed 10 May 2022].

[3] "Node.js," en.wikipedia.org, [Online]. Available: https://en.wikipedia.org/wiki/Node.js. [Accessed 12 May 2022].

[4] C. Kopecky, "What is Node.js? A beginner's introduction to JavaScript runtime," www.educative.io, [Online]. Available: https://www.educative.io/blog/what-is-nodejs. [Accessed 12 May 2022].

[5] "What Is Node.js?," developer.oracle.com, [Online]. Available: https://developer.oracle.com/nodejs/what-is-node-js/. [Accessed 12th May 2022].

[6] "Node.js v18.2.0 documentation," nodejs.org, [Online]. Available: https://nodejs.org/api/console.html#consoletabletabulardata-properties. [Accessed 12 May 2022].

[7] "How to use Class in Node.js ?," geeksforgeeks.org, [Online]. Available: https://www.geeksforgeeks.org/how-to-use-class-in-node-js/. [Accessed 12 MAy 2022].

[8] D. ELKABES, "The Complete Guide to Prototype Pollution Vulnerabilities," mend.io, 22 JULY 2021. [Online]. Available: https://www.mend.io/resources/blog/prototype-pollution-vulnerabilities/#Mitigating_Prototype_Pollution_Vulnerabilities. [Accessed 13 May 2022].

[9] R. Marot, "Identifying Prototype Pollution Vulnerabilities: How Tenable.io Web Application Scanning Can Help," tenable.com, 2021. [Online]. Available: https://www.tenable.com/blog/identifying-prototype-pollution-vulnerabilities-using-tenable-io-web-application-scanning. [Accessed 13 May 2022].

[10] C. Xu, "What is Prototype Pollution?," codeburst.io, 2019. [Online]. Available: https://codeburst.io/what-is-prototype-pollution-49482fc4b638. [Accessed 13 May 2022].

[11] O. Arteau, Prototype pollution attack in NodeJS Applications, http://prototypejs.org/.

[12] "Map," developer.mozilla.org, [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map. [Accessed 13 May 2022].

[13] "DAPP: automatic detection and analysis of prototype pollution vulnerability in Node.js modules," link.springer.com, 2022. [Online]. Available: https://link.springer.com/article/10.1007/s10207-020-00537-0. [Accessed 13 May 2022].