A Study of Node.js Using Injection Vulnerabilities

Tushar Srivastava*
Scholar, Department of CSE, ABESIT, Ghaziabad, Uttar Pradesh, India
tusharsrivastava162@gmail.com

Ashutosh Pandey
Assistant Professor, Department of CSE, ABESIT, Ghaziabad, Uttar Pradesh, India
ashutosh.pandey@abesit.in

Rizwan Khan
Associate Professor, Department of CSE, ABESIT, Ghaziabad, Uttar Pradesh, India
rizwan.khan@abesit.in

Abstract—The Node.js community has prompted the making of numerous applications, for example, server-side web applications and work area applications. Not at all like client-side JavaScript code, Node.js applications can collaborate uninhibitedly with the working framework without the advantages of a security sandbox. The mind boggling exchange between Node.js modules prompts unobtrusive infusion vulnerabilities being presented crosswise over module limits. This paper displays a substantial scale consider crosswise over 235,850 Node.js modules to investigate such vulnerabilities. We demonstrate that infusion vulnerabilities are predominant practically speaking, both due to eval, which was already examined for program code, and because of the effective executive API presented in Node.js. Our investigation demonstrates that a great many modules might be helpless against charge infusion assaults and that notwithstanding for prominent undertakings it requires long investment to settle the issue.

Keywords—Node.js, open source, application, framework

I. INTRODUCTION

In the fast advancement of Web today, numerous sites are looked with new issues, for example, the issue multiuser solicitations and high simultaneousness. The dynamic scripting dialect JavaScript has turned out to be massively well known for customer and is broadly utilized as a part of Web advancement. Node.js remains for one new innovation in JavaScript. Node.js is a stage based on Chrome's JavaScript run time for effortlessly constructing quick, adaptable system applications [1]. JavaScript is the most broadly utilized programming dialect for the customer side of web applications, controlling more than 90% of the present sites. As of late, JavaScript has moved toward becoming increasingly prominent for stages past the program: server-side and work area applications (Node.js), versatile programming (Apache Cordova/Phone Gap); it is even utilized for composing working frameworks (Firefox OS). One of the powers behind utilizing JavaScript in different areas is to empower customer side software engineers to reuse their skills in different environments. Node.js utilizes an event driven, non blocking I/O display that makes it lightweight and effective, ideal for information serious ongoing applications that keep running crosswise over conveyed gadgets [1].

Node.js fame overviews performed by official site demonstrate that the normal downloads are more than 35,000 since the variant 0.10 discharged in March 2013. Enterprises are rapidly understanding the significance of Node.js and five noteworthy PAAS suppliers have bolstered Node.js . This paper is to completely research a security issue particular to JavaScript executed on the Node.js stage. In particular, we center around infusion vulnerabilities, i.e., expert grammar blunders that empower an aggressor to infuse and execute noxious code in a unintended way.

Infusion vulnerabilities on the Node.js stage vary from those on other JavaScript stages in three ways.

A. Injection APIs and impact of attacks:

Node.js provides two families of APIs that may accidentally enable injections. The eval API and its variants take a string argument and interpret it as JavaScript code, allowing an attacker to execute arbitrary code in the context of the current application. The exec API and its variants take a string argument and interpret it as a shell command, allowing an attacker to execute arbitrary system-level commands, beyond the context of the current application. Moreover, attackers may combine both APIs by injecting JavaScript code via eval, which then uses exec to execute shell commands. Because of these two APIs and because Node.js lacks the security sandbox known from browsers, injection vulnerabilities can cause significantly more harm than in browsers, e.g., by modifying the local file system or even taking over the entire machine.

B. Developer stance:

While it is tempting for researchers to propose an analysis that identifies vulnerabilities as a solution, to have longer-range impact, it helps to understand Node.js security more holistically. By analyzing security issues reported in
the past and through developer interactions, we observed that, while injection vulnerabilities are indeed an important problem, developers who both use and maintain JavaScript libraries are reluctant to use analysis tools and are not always willing to fix their code.

C. Blame game:

A dynamic we have seen develop is a blame game between Node.js module maintainers and developers. who use these modules, where each party tries to claim that the other is responsible for checking untrusted input. Furthermore, while a developer can find it tempting to deploy a local fix to a vulnerable module, this patch is likely to be made obsolete or will simply be overwritten by the next module update. These observations motivated us to develop an approach that provides a high level of security with a very small amount of developer involvement.

II. BACKGROUND

Node.js was created by Ryan Dahl in 2009 [2], sponsored by Joyent, the firm where Dahl worked. Node.js is an open source, cross-platform runtime environment for server-side and networking applications. Node.js applications are written in JavaScript, and can be run within the Node.js runtime on OS X, Microsoft Windows, Linux and FreeBSD. Node.js provides an event-driven architecture and a non-blocking I/O API that optimizes an application's throughput and scalability. These technologies are commonly used for real-time applications[2].

According to its own website, Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.[3]. Node.js allows the creation of web servers and networking tools, using JavaScript and a collection of "modules" that handle various core functionality. Modules handle File system I/O, Networking (HTTP, TCP, UDP, DNS, or TLS/SSL), Binary data (buffers), Cryptography functions, Data streams, and other core functions. Node.js modules have a simple and elegant API, reducing the complexity of writing server applications[6].

Node.js has grown to be one of the most used and talked about web technologies in recent years[4]. A simple search on StackOverflow shows over 13 000 questions related to Node.js[8] and the question "How to get started with Node.js" has over 1200 upvotes and an answer with over 2700 upvotes[9] (An upvote is the action of giving an article, post or video on an internet forum or website a positive rating, such as a "thumbs up"). A large number of corporations and organizations use Node.js as part of their core IT infrastructure, these corporations include Yahoo, Uber, New York Times, Dow Jones, PayPal, Microsoft, eBay and Walmart.

Originally founded by Tom Preston-Werner, Chris Wanstrath, and PJ Hyett to simplify sharing code, GitHub has grown into the world's largest code host and is now the most used open source community on the web[12]. To date GitHub has over 6.7 million users and over 1.1 million projects[13]. Open source software has a massive impact on innovation in information technology[14] and is therefore a reflection on the entire tech community[13].

Node.js and GitHub are unified and the development of Node.js is dependent on the GitHub community as all development takes place on the Node.js GitHub repository.

Projects on GitHub can be accessed and manipulated using the standard git command-line interface and all of the standard git commands work with it. GitHub also allows registered and unregistered users to browse public repositories on the site. Multiple desktop clients and git plugins have also been created by GitHub and other third parties which integrate with the platform.

The site provides social networking-like functions such as feeds, followers, wikis (using wiki software called gollum) and a social network graph to display how developers work on their versions ("forks") of a repository and which fork (and branch within that fork) is newest.

A user must create an account in order to contribute content to the site, but public repositories can be browsed and downloaded by anyone. With a registered user account, users are able to discuss, manage, create repositories, submit contributions to others' repositories, and review changes to code[7].

III. A STUDY OF INJECTION VULNERABILITIES

To better understand how developers of JavaScript for Node.js handle the risk of injections, we conduct a comprehensive empirical study involving 235,850 npm modules. We investigate four research questions (RQs):

RQ1: Prevalence. At first, we study whether APIs that are prone to injection vulnerabilities are widely used in practice. We find that injection APIs are used frequently and that many modules depend on them either directly or indirectly.

RQ2: Usage. To understand why developers use injection APIs, we identify recurring usage patterns and check whether the usages could be replaced with less vulnerable alternatives. We find that many uses of injection APIs are unlikely to be replaced by less vulnerable alternatives.
RQ3: Existing mitigation. To understand how developers deal with the risk of injections, we study to what extent data gets checked before being passed into injection APIs. We find that most call sites do not at all check the data passed into injection APIs.

RQ4: Maintenance. To understand whether module developers are willing to prevent vulnerabilities, we report a sample of vulnerabilities to developers and analyze how the developers react. We find that, even for widely used modules, most vulnerabilities remain in the code several months after making the developers aware of the issue.

A. RQ1: Prevalence

To assess how wide-spread the use of injection APIs is in real-world JavaScript code written for Node.js, we analyze call sites of these APIs across all 235,850 npm modules. For each module, we perform a regular expression-based search that detects call sites of injection APIs based on the name of the called function. We call modules that rely on these injection APIs injection modules. Furthermore, we analyze dependencies between modules, as specified in their package.json file, to assess whether a module uses another module that calls an injection API. Given a module minj that calls an injection API, we say that another module m1 has a “level-1” (“level-2”) dependence if it depends on minj (via another module). For example, if many modules with level-1 dependencies on injection APIs would depend on only a single injection module, then protecting this module against injections would protect all its dependent modules. To estimate how many modules would be directly and indirectly protected by protecting a subset of all modules, we analyze the module dependence graph. Specifically, we initially mark all injection modules and all modules with level-1 dependencies as unprotected and then, assuming that some set of injection modules is protected, propagate this information along dependence edges. Based on this analysis, we compute a minimal set of injection modules to protect, so that a particular percentage of modules is protected. The strategy we consider is to fix vulnerabilities in those modules with the highest number of level-1 dependences.

Figure 1 summarizes the results of this experiment. We find that fixing calls to the injection APIs in the most popular 5% of all injection modules will protect almost 90% of the level-1 dependences. While this result is encouraging, it is important to note that 5% of all modules still corresponds to around 780 modules, i.e., many more than could be reasonably fixed manually.

B. RQ2: Usages

The results from RQ1 show that the risk of injections is prevalent. In the following, we try to understand why developers take this risk and whether there are any alternatives. We manually inspect a random sample of 50 uses of exec and 100 of eval to identify recurring usage patterns. Our classification yields four and nine recurring patterns for exec and eval, respectively, illustrated in detail in the appendix.
Patterns of exec usage: The majority of calls (57%) trigger a single operating system command and pass a sequence of arguments to it. For these calls, the developers could easily switch to spawn, which is a safer API to use, equivalent to the well-known execv functions in C. The second-most common usage pattern (20%) involves multiple operating system commands combined using Unix-style pipes. For this pattern, we are not aware of a simple way to avoid the vulnerable exec call. The above-mentioned spawn accepts only one command, i.e., a developer would have to call it multiple times and emulate the shell’s piping mechanism. Another interesting pattern is the execution of scripts using relative paths, which accounts for 10% of the analyzed cases. This pattern is frequently used as an ad-hoc parallelization mechanisms, by starting another instance of Node.js, and to interoperate with code written in another programming language.

Patterns of eval usage: Our results of the usage of eval mostly match those reported in a study of client-side JavaScript code, showing that their findings extend to Node.js JavaScript code. One usage pattern that was not previously reported is to dynamically create complex functions. This pattern, which we call “higher-order functions”, is widely used in server-side JavaScript for creating functions from both static strings and user-provided data. We are not aware of an existing technique to easily refactor this pattern into code that does not use eval.

Overall, we find that over 20% of all uses of injection APIs cannot be easily removed. Furthermore, we believe that many of the remaining uses are unlikely to be refactored by the developers. The reason is that even though several techniques for removing some usages of eval have been proposed years ago [12], [11] and the risks of this function are widely documented [4], [9], using eval unnecessarily is still widespread.

C. RQ3: Mitigation

The high prevalence of using APIs prone to injections raises the question how developers protected their code against such vulnerabilities. To address this question, we manually analyze the protection mechanisms used by the sample of API usages from RQ2. Specifically, we have analyzed (i) whether a call site of an injection API may be reached by attacker-controlled data, i.e., whether any mitigation is required, and (ii) if the call site requires mitigation, which technique the developers use.

We find that for 58% of the inspected call sites, it may be possible that attacker-controlled data reaches the injection API. Among these call sites, the following mitigation techniques are used:

None. A staggering 90% of the call sites do not use any mitigation technique at all. For example, the call to exec in the motivating example in Figure 3 falls into this category.

```
1 function backupFile (name, ext) {
2     var cmd = [];
3     cmd.push("cp");
4     cmd.push("-" + name + " + ext");
5     cmd.push(`/localBackup ");
6     exec(cmd.join(" "));
7     var kind = (ext === "pg") ? "pics" : "other";
8     console.log(eval("messages.backup_ + kind");
9 }
```

Fig-3 Motivating example

Regular expressions. For 9% of the call sites, the developers harden their module against injections using regular expression-based checks of input data. An example in our data set is shown in Figure 4. Unfortunately, most regular expressions we inspected are not correctly implemented and cannot protect against all possible injection attacks. For example, the escape method in the figure does not remove the back ticks characters allowing an attacker to deliver a malicious payload using the command substitution syntax, as illustrated in the last line of Figure 4. In general, regular expressions are fraught with danger when used for sanitization [10].

```
1 function escape(s) {
2     return s.replace (/\"/g, \" ");
3 }
4 exports.open = function open (target, callback) {
5     exec (opener + '+' + escape (target) + '');
6 }
7 // possible attack:
8 open("rm -rf ");
```

Fig-4 Regex-based sanitization and input that bypasses it.
Sanitization modules. To our surprise, none of the modules uses a third-party sanitization module to prevent injections. To validate whether any such modules exists, we searched the npm repository and found six modules intended to protect calls to exec against command injections: shell-escape, escape-shellarg, command-join, shell-quote, bash, and any-shell-escape. In total, 198 other modules depend on one of these sanitization modules, i.e., only a small fraction compared to the 19,669 modules that directly or indirectly use exec. For eval, there is no standard solution for sanitization and the unanimous experts advice is to either not use it at all in combination with untrustworthy input, or to rely on well tested filters that allow only a restricted class of inputs, such as string literals or JSON data.

For the remaining 42% of call sites, it is impossible or very unlikely for an attacker to control the data passed to the injection APIs. For example, this data includes constant strings, IDs of operating system processes, and the path of the current working directory.

We conclude from these results that most modules are vulnerable to injections and that standard sanitization techniques are rarely used, despite some of the specialized modules available for this purpose. Developers are either unaware of the problem in the first place, unwilling to address it, or unable to properly apply existing solutions.

D. RQ4: Interactions with Module Maintainer

The moderation strategies talked about for RQ3 rely upon developers putting time and exertion into staying away from vulnerabilities. To better comprehend to what degree engineers will contribute that sort of exertion, we revealed 20 beforehand obscure charge infusion vulnerabilities to the designers of the individual modules that call the infusion APIs. We have physically identified these vulnerabilities amid the investigation portrayed in this segment. For every defenselessness, we depict the issue and give a case assault to the task developers.

One may guess that these vulnerabilities are trademark to disagreeable modules that are not anticipated that would be all around kept up. We checked this speculation by estimating the quantity of downloads between January 1 and February 17, 2016 for three arrangements of modules: (i) modules with vulnerabilities revealed either by us or by others by means of the Node Security Platform, (ii) all modules that call an infusion API, (iii) all modules in the npm vault.

Figure 5 summarizes our outcomes on a logarithmic scale. The cases are drawn between the lower quartile (25%) and the upper one (75%) and the even line denotes the middle. The outcomes refute the theory that helpless modules are disagreeable. In actuality, we watch that different helpless modules and infusion modules are profoundly prevalent, uncovering a large number of clients to the danger of injections.

Fig-5 Comparison of the popularity of all the modules, modules with calls to injection APIs, and modules with reported vulnerabilities.

IV. CONCLUSION

In this paper, we endeavor to consider Node.js utilizing the injection vulnerabilities, which might be the principle unsettling influence to its activity. The Node.js people group has incite the making of various applications, for instance, server-side web applications and work region applications. Not in the slightest degree like customer side JavaScript code, Node.js applications can team up uninhibitedly with the working system without the upsides of a security sandbox. Therefore, the security needs in the manet are significantly higher than those in the customary wired systems.
REFERENCES


