Dynamic Taint Analysis in JavaScript for JavaScript

Submitted in partial fulfillment of the requirements for

the degree of

Master of Science

in

Information Security

Wai Tuck Wong

B.S., Information Systems, Singapore Management University

Carnegie Mellon University
Pittsburgh, PA

May, 2020

Acknowledgements

I would like to thank Professor Limin Jia for supervising and guiding me through this field of research - I have learnt countless things and was exposed to new ways of thinking which will prove to be invaluable in the future. I am grateful for having Professor Lujo Bauer as a reader of my thesis, providing insights and pointing out a corner case in my implementation of the static analysis. This work is done together with the team led by Darion Cassel, together with Spencer Yu. I am really grateful for the all feedback from the folks in Professor Limin Jia's group at Cylab for my defense. Finally, I would like to thank everyone at the Information Networking Institute for being supportive throughout my journey at Carnegie Mellon University. It was a tremendously fruitful experience - one that I would remember in the days ahead. The work in this thesis is self-funded.

Abstract

In recent years, we have seen a rise in the number of applications built on JavaScript, bolstered by the increase in popularity of application frameworks such as `NODE.JS`. In particular, `NODE.JS` provides convenience for developers through packages which they can import functionality from. This has attracted malicious actors to turn their focus to find ways of exploiting such packages. In fact, code injection attacks are prevalent in the ecosystem - they allow for arbitrary code or commands to be run - and are the most critical vulnerabilities in the language. Unfortunately, JavaScript is a complicated language, and significant research effort has been invested in developing dynamic taint analysis tools to identify such vulnerabilities at scale. Prior work attempted to build such tooling on top of JavaScript engines, but such attempts are highly unmaintainable due to the rate of change in both the language and the engines they run on. Other platform independent approaches lack the flexibility in keeping track of taint for individual characters in strings. In this thesis, we propose a flexible framework for doing dynamic taint analysis purely in JavaScript that is capable of byte level tainting.

We present NodeTaintProxy, a dynamic taint analysis engine built via augmenting the instrumentation engine, Jalangi2. Here, we developed a novel approach in dealing with primitives by specifying the behavior for wrapping, and we show that this behavior respects the semantics outlined in ECMA-262. We also show how taint propagation in dynamic code generation can be handled through a mix of code rewriting and static analysis on the dynamically generated code. Finally, we evaluate our tool on existing vulnerabilities in `NODE.JS` packages and show that our tool is successful in finding these vulnerabilities.

Table of Contents

List of Tables

List of Figures

Symbols

M_W	The Wrapper map that stores mappings of object references to their unique IDs
M_E	The shared context map that maps variables to be tainted to their unwrapped values
M_T	The Taint map that stores mappings of IDs to their taint entries
ID	A unique identifier assigned to a particular object reference

Abbreviations

JS	JavaScript
ES5.1	ECMAScript 5.1
ES6	ECMAScript 2015/ECMAScript 6
API	Application Programming Interface
ACI	Arbitrary Command Injection
ACE	Arbitrary Code Execution
AST	Abstract Syntax Tree
NPM	Node Package Manager
XSS	Cross-site Scripting
LHS	Left-hand side

1

Introduction

JavaScript, for the seventh year in a row, is the most popular language - according to a survey done by StackOverflow on nearly 90,000 developers. In fact, JavaScript is the most commonly used programming language among both amateur and professional developers [21]. JavaScript thrives because of its versatility - first introduced in September 1995 [26], it provided interactivity to users on the web in an era where static web pages were previously the norm. It has since been standardized under the ECMAScript standard [43]. The language evolves quickly, with revisions every year and a living standard. Table 1 describes the list of ECMAScript standards published over the years and the years they were published [42].

To no one's surprise, JavaScript became the de facto standard for programming on the web, and with the introduction of frameworks such as `NODE.JS` and Electron, they soon became accessible to server-side developers and desktop application developers. In fact, `NODE.JS` is most commonly used framework among developers, according to the same StackOverflow survey [21]. With such prolific use, our thesis will focus on `NODE.JS` - we want to find vulnerabilities in this platform before the attackers do. To do that, we need to understand the ecosystem and see how our approach fits in

ECMAScript Version	Date Standardized
ECMAScript 1	1997
ECMAScript 2	1998
ECMAScript 3	1999
ECMAScript 5	2009
ECMAScript 5.1	2011
ECMAScript 2015	2015
ECMAScript 2016	2016
ECMAScript 2017	2017
ECMAScript 2018	2018
ECMAScript 2019	2019
ECMAScript.Next	Living Standard

Table 1.1: ECMAScript Standards and Revision Dates

this space, and we will elaborate more on this point in the section below.

1.1 NODE.JS and the NPM Ecosystem

`NODE.JS` was first introduced as a way of running JavaScript on the backend and it soon became popular among developers as a framework to use for developing server-side applications. With a budding community support, NPM (Node Package Manager) was introduced as a way of allowing developers to leverage on modules created by others in their own projects by pulling from a common NPM Registry. As of February 2019, it houses over 800,000 packages in the registry [83].

However, such a registry is not without issues. In a typical `NODE.JS` application, developers simply import third party modules from NPM, without verifying the functionality of these modules. In the developer's mind, they have a mental model of how the underlying package ought to behave. For example, they infer that the underlying package is well-implemented and tested from the large number of downloads. These assumptions are frequently untested and may be violated by the underlying package [33].

Developers who rely on packages from the registry implicitly trust the packages

they use and all of the packages' dependencies. This has several issues. There is no guarantee of code quality of the underlying packages. It might even be the case that the contributor themselves cannot be trusted [83]. Cases of typosquatting occurred many times throughout the history of NPM, where misspelled package names led to malicious packages being installed on the unsuspecting developer's system [39]. While a powerful resource for developers, they may not understand the full repercussions of using a package from the repository, and may end up introducing vulnerabilities to their applications without them even realizing it.

1.1.1 Security Concerns in NPM

Here, we consider from the perspective of a developer the potential security problems that they might face, if they were to leverage on the library of existing packages in the NPM registry.

1. **Unvetted Publishing Model.** We first note that publishing to the NPM repository is completely unvetted. All it takes to publish is a simple command `npm publish` [3], and the package is available to the whole world. The publishers are responsible for the maintenance of the package. Code quality between packages in the ecosystem therefore varies widely when there's completely no review process. Even if a vulnerability was found in a package, the package remains on the NPM registry for download, and only a warning is displayed if it was downloaded. This decision was made to maximize availability, to allow packages that depend on the vulnerable package to continue to work. For example, the `node-serialize` module [9] continues to see over 1000 weekly downloads even after a critical vulnerability was discovered, and still sees use despite there being no patches available for it [13].

2. **Untrusted Authors.** Since the public NPM repository is unvetted, anyone,

including malicious actors, can simply upload a package containing a backdoor that runs malicious commands on the victim's machine. An unknowing developer may download such a package and find his application or even machine to be compromised. This is a real danger that developers worry about - in a survey done by NPM of 16000 `NODE.JS` developers, 77% of respondents were concerned about the security of open source code [28]. Their fears were not unfounded. In 2018, a popular `NODE.JS` module, Event-Stream, had their code backdoored with malicious code that stole private keys from the Bitcoin wallet CoPay [38]. It was undiscovered for over a month, affecting 3931 packages that depended on it [45]. We see that the highly interdependent nature of the NPM repository is fragile - a vulnerability or malicious code in a single package can have ripple effects and induce vulnerabilities in all packages that depend on it.

3. **Full Privilege of Package Code.** To exacerbate the problem, the packages not only have the same privileges as the user at the point of installation (through the install scripts), the packages themselves run without explicit authorization on what resources they are allowed to access [71]. For example, a package that provides a wrapper to delete files may do it via running the shell command `"rm -f <file>"`, thus leading to a command injection vulnerability through the `file` argument, and the unsuspecting developer using the package will not be aware of this underlying vulnerability unless they go in to audit the source code manually.

Without the proper procedures or tooling, vulnerabilities will creep into ecosystem and packages containing vulnerable code will be exploitable. In particular, JavaScript is prone to code injection attacks on many fronts - on the browser, it manifests as a cross-site scripting (XSS) attack, while on server-side applications, they could be arbitrary code execution or command injection attacks (or broadly

speaking, code injection attacks). In fact, these attacks are so prevalent that the OWASP Foundation lists these vulnerabilities as the top 10 web application security risks [17].

Client-side vulnerabilities have been well studied [67][62][54] and will not be a focus in our evaluation. We will focus on code injection attacks for server-side applications in the rest of the thesis, in particular for code injection vulnerabilities for packages in `NODE.JS`. We will present more details on code injection vulnerabilities for `NODE.JS` packages in Chapter 2.

On `NODE.JS`, we see a ripe set of tools that an attacker can leverage to interact with the system. Below, we show how these vulnerabilities can manifest in a package used by the developer.

1. **Input provided to shell commands.** In implementing certain functionality, some package developers may choose to implement an external script or leverage on an existing system binary. In their code, they would pass the arguments directly to the executable file using unsafe APIs such as `exec` or `execSync`. This can lead to command injection attacks.

2. **Evaluation of input.** Package developers are known to use `eval` as a tool to evaluate user input in a power user fashion to get *"cleaner"* code, despite the known dangers of doing so [82]. An example of this is the `mongui` package [76], which uses `eval` so that users of the package have a *"more natural programming interface"*. Such unrestricted evaluation of user input can lead to arbitrary code execution.

In fact, for packages that exhibit the above behavior, very few, if any of them, use any form of sanitization, and they rarely document that untrusted input should not be passed into the interface of the package. Here, *sanitization* refers to removing

or encoding some parts of the input such that it neutralizes malicious behavior that can result from the input [5]. In a study of injection attacks, `growl`, a highly popular package that passed input directly to `exec`, was discovered to have as many as 15 packages that depended on it that did not do any checks or sanitization [71]. The 4 packages that attempted to do sanitization were found to be bypassable. As we see here, these are real problems in the NPM ecosystem. In the next section, we detail potential ways of finding such vulnerabilities so we can solve the problem at hand.

1.2 Existing Approaches to Finding Code Injection Vulnerabilities

What we want to achieve is a way for developers to audit their dependencies for code injection vulnerabilities. More broadly, such a tool would also allow security analysts to probe at different packages and their interfaces and see if an input provided to the package interface would lead to code injection, such as calling `eval` on the input or the `exec` function of the `child_process` module on the input.

1.2.1 Static Analysis

One such way of determining whether an input would lead to undesired behavior is to statically analyze the source code of the application. For example, one can scan the underlying source code to see if there are any uses of `eval`, and then manually evaluate whether that usage is safe [71]. More advanced methods of static analysis (via static taint analysis) considers whether the user input could possibly flow to the function call. Given the source code of the application, a data flow graph is constructed, where an edge exists from one variable to another if and only if there is an assignment from the source variable to the target variable [36]. An advantage is that no concrete input needs to be provided to the package being analyzed. However, such an approach is would be highly inaccurate. Without runtime information, it would be very difficult to determine whether a branch would be taken, for example.

Static analysis requires us to reason about all execution paths, which inevitably leads to a state explosion, and potentially many false positives. Furthermore, we will not be able to determine actual runtime behavior where some parts of the code will not run, for example when an exception is thrown in the code or when dynamically generated code is supposed to run, which leads to us missing more cases than what we would like.

1.2.2 Dynamic Analysis

This naturally moves us to trying a different technique. A different approach uses dynamic analysis, where we have access to runtime information by providing a concrete input to the package's interface. We then check at the end whether some undesired function is triggered (e.g. `eval`) [44]. While this has the advantage of reducing false positives over the above approach, we cannot ascertain whether the input has any influence over the parameters provided to the undesired function. More concretely, we wish to figure out if the attacker/adversary has control over the arguments to the undesired function from the adversary's input alone. This however is not possible purely with Dynamic Analysis. In particular, just providing random inputs to the package's interface do not give us a principled way of analyzing the influence of the input on the end result [59].

1.2.3 Dynamic Taint Analysis

This moves us to the technique that we propose to use - dynamic taint analysis. In this case, we mark the input to the package as tainted, and we propagate the taint on the input to other variables if the variable was affected by the input [60]. At the end, we check to see if the undesired function was called with an argument that could potentially be controlled by the adversary (i.e. the argument was tainted). This gives us better results in terms of reducing false positives (since we know the

argument is indeed attacker controlled), but in exchange we need a more complicated infrastructure to keep track of taint and propagate taint as the JavaScript code runs. We will discuss more details on the background of this approach, related work, and how we implement this approach in the later chapters.

1.2.4 Challenges for JavaScript

In building a platform agnostic dynamic taint analysis framework for JavaScript, we will face challenges from the language itself. Below, we summarize the key challenges in dealing with the language.

1. **Complexity of Semantics.** JavaScript is a complicated language which prevents introspection into its internals. For example, we won't be able to view the heap address of a JavaScript object, or view certain properties which are *internal* to the JavaScript object. Some properties are read-only and prevent modification from within JavaScript itself. Furthermore, JavaScript has rather complicated semantics, for instance the equality operator `==` is neither reflexive nor transitive [52]. In dealing with JavaScript, we must be careful not to make any assumptions which do not adhere to the underlying semantics of JavaScript, lest we end up with a buggy implementation of our tool.

2. **Native Functions.** Much like how C syscalls are a problem [68], native functions are the equivalent construction in JavaScript, where only the input and result of the computation are visible from JavaScript itself. This lack of transparency forces us to adopt workarounds to ensure our tool still works as expected, as we will see later.

3. **Dynamically Generated Code.** Finally, JavaScript allows for additional code to be generated and evaluated at runtime through the `eval` function and `Function` constructor. Usage of this feature is more common in reality than

once believed [65], so we will have to ensure that our tool works even in the case where code is dynamically generated. In cases where `eval` is not the sink, we want to make sure that taint is propagated correctly as well, even when the argument to `eval` is tainted.

We will outline the background required to tackle these problems in Chapter 2 and detail how we deal with the above problems in detail in Chapter 4.

JavaScript is a challenging language to analyze, but it has a large impact in the real world, where an entire ecosystem of packages could be analyzed with the right tooling. As of now, the developer has to implicitly trust the modules he utilizes without a way of determining whether the package does what he expects. We want to provide tooling to allow analysts to find code injection vulnerabilities faster, and even allow developers to find such vulnerabilities in their own code base.

We are motivated to create a state of the art dynamic taint analysis framework that allows us to analyze JavaScript code for code injection vulnerabilities; we make it platform agnostic by implementing it on top of existing instrumentation frameworks that employ source-to-source rewriting. Through this, we can do taint tracking and propagation, and we further augment it with the ability to propagate taint even when the argument to `eval` is tainted, using a novel way of tainting via program rewriting and static analysis. We show that our approach works on the vulnerabilities described earlier through a series of case studies that display code injection vulnerabilities. Our current implementation focuses on `NODE.JS`, but the implementation design should minimize effort required to port to a different JavaScript platform.

1.3 Outline

The thesis is broken down into the following parts. First, in Chapter 1, we outline the motivation in carrying out the development of a dynamic taint analysis tool for

JavaScript, specifically to analyze NPM packages. Next, we outline the necessary background needed to understand our approach in Chapter 2. In Chapter 3, we look at prior work that has been done in developing dynamic taint analysis for JavaScript in different areas, as well as look at other large scale studies done in the NPM ecosystem, and we compare our approach to theirs. In Chapter 4, we explain our tool, NodeTaintProxy, and we describe how we approach maintaining and propagating taint in JavaScript, while at the same time maximizing our adherence to the underlying semantics. We also show our novel method that propagates taint in dynamically generated code through program rewriting and static analysis. In Chapter 5, we show that our tool is able to detect code injection vulnerabilities that were made public recently, as well as case studies that were selected from prior work. We also show that our approach is promising in finding new vulnerabilities in a feasibility study in that chapter. We then round up our discussion with limitations and future work for the framework in Chapter 6. Finally, we conclude by summarizing our key findings in Chapter 7.

2

Overview and Background

In this chapter, we first briefly outline the construction that our approach will take. We will then delve deep into the background required to understand each component of our approach in the subsequent sections of this chapter.

2.1 Overview

In Figure 2.1, we provide an overview of the internals of our approach. On the left side of the diagram in Figure 2.1, we outline the source code of a simple JavaScript program running on our framework, which is a `NODE.JS` package under inspection. On the right, we show the data structures that are involved in the framework. The wrapper map keeps track of a mapping of object references to unique identities of all the values we have seen thus far, and the taint map keeps track of a mapping of the unique identities to their taint information. We will elaborate more concretely about the data structures in Chapter 4.

The code referenced simply declares two variables y and s in lines 1 and 2, taints the variable s in line 4, declares a new variable x that is set to the value `y+s` in line 6 (which is the string `'1+1'`) evaluates the string dynamically in line 8. The

data structures depicted shows the internal state of our framework after line 6 has executed. For example, x has `ID(2)` since the ID associated with the string `'1+1'` is `ID(2)` in our wrapper map. Furthermore, we note that the indices of x that are tainted are the characters `'+1'` at the end of the string. We can tell that from the taint map, which tells as that indices 1 and 2 of the string are tainted for the value associated with `ID(2)`. This makes sense, since the tainted values were derived from a tainted variable s which has the value `'+1'`. We see that s was tainted since the value of s maps to `ID(1)` in the wrapper map, and in the taint map, `ID(1)` maps to a taint entry that tells us that the whole string was tainted (as we can see from the `true` in the first entry of the tuple). We note some interesting problems that this piece of code raises in our analysis.

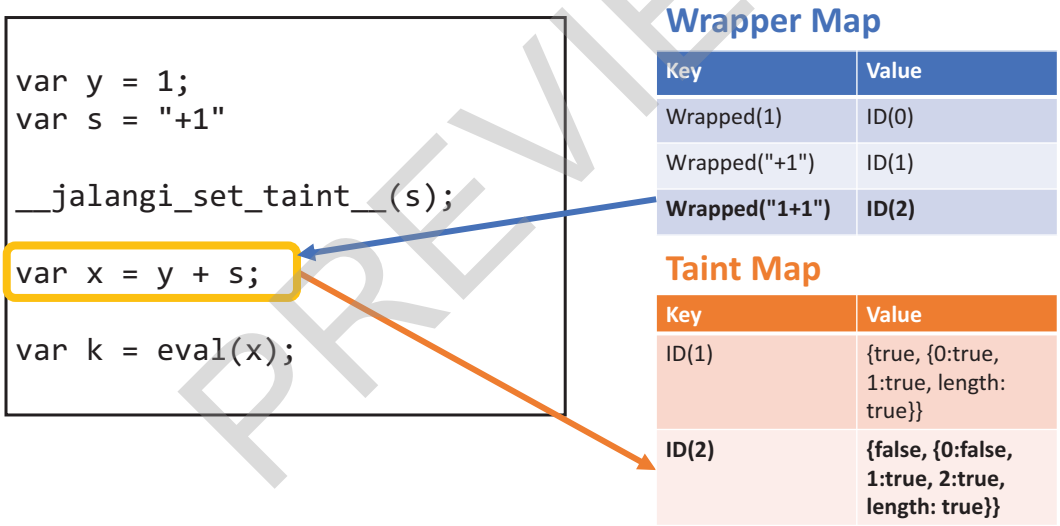

Figure 2.1: Overview of Internals of NodeTaintProxy

On line 1, we declared a variable y with value 1. The identity of this value 1 should differ from another variable which is also have a value of 1. We see that in this case, we have wrapped the value and mapped it to a unique ID, `ID(0)`, in our wrapper map. More generally, we need to wrap primitives which do not have

references in JavaScript so we can uniquely identify them. We do so with a object class called *Proxy*. We elaborate more on what primitives are in JavaScript, and what a *Proxy* is, and how it is useful for our analysis in Sections 2.1.1 and 2.2.1 respectively.

On line 4, we tainted the variable s, and we used it to compute the value of the variable x in line 6. This means that taint needs to propagate from the variable s to the variable x, and the taint map has to be updated (as indicated by the orange arrow), so that we can keep track of taint of every value in the system. We do so by hooking each operation in JavaScript using instrumentation, which allows us to define custom behavior before and after each operation. We will explain what instrumentation is and how it is achieved in Section 2.2.2 in this chapter.

Finally, on line 8, a variable k was declared by dynamically evaluating the string `"1+1"` stored in the variable x. Since x tainted, the new variable k should also be tainted. More generally, variables introduced in the dynamically generated code should also be tainted. We look at how dynamic code generation is achieved in JavaScript and note specific details we have to take care of in our framework in Section 2.1.3.

The simple example above highlights the need to understand the underlying language and all its nuances in order to perform dynamic taint analysis correctly. Furthermore, JavaScript provides some useful classes which we will leverage on to build our framework. In this section, we will describe important aspects of JavaScript that we have to be aware of when we develop our tooling. All of the description below are based on the latest edition of the ECMAScript standard, available here [43].

2.1.1 Primitives

The simplest types in JavaScript are the built-in primitive types, which are **Undefined, Null, Boolean, Number, String** and **Symbol**. These primitive types