Fall 2018

Existence of Dependency-Based Attacks in NodeJS Environment

Prachi R. Patel
Iowa State University

Existence of Dependency-Based Attacks in NodeJS Environment

by

Prachi Patel

A creative component submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Lotfi Ben Othmane, Co-major Professor
Doug Jacobson, Co-major Professor

Iowa State University

Ames, Iowa

2018

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to thank Dr. Lotfi Ben-Othmane for his guidance throughout the course of this research and Prof. Doug Jacobson for his support. I would also like to thank Ameerah-Muhsinah Jamil for helping out with surveys and observations and Brian Pfretzschner for his guidance.

I would like to thank the department faculty and staff for making my time at Iowa State University a wonderful experience and to friends and family for all their help and support.

ABSTRACT

Node.js is an open source server-side run-time platform for JavaScript applications. Node.js applications are dependent on several, even hundreds, packages, which in turn have many dependencies. Node.js supports monkey-patching, global variables, shared cache of loaded modules and functions as function parameters, which can be exploited through dependent packages. There is always a risk of malicious code hidden in one of these dependencies.

This work analyzes vulnerabilities found in Node.js based applications, discusses five basic types of attacks: data manipulation on global and local level, leakage of sensitive data to malicious server, interruption of service provided by the host application and dependency cache manipulation, and reports about the assessment of five frequently-used Node.js packages with respect to dependency-based vulnerabilities. The assessment revealed a vulnerability in file bson.js in an NPM package name 'bson,' which is the default Binary JSON (BSON) parser. This package is used by 375 dependent applications.

CHAPTER 1. INTRODUCTION

JavaScript plays a significant role in modern web development. It allows for rapid development and high performance since it is dynamic and asynchronous. It was intended as a front-end language to be used in browser environments. Due to advent of Node.js, JavaScript can also be used on back-end.

Node.js applications use same programming concepts as Javascript. However, Node.js applications are based on a large number of third-party dependencies. These dependencies can have a huge impact on the security of the application.

One of the security risks could be the presence of malicious code in third-party dependencies. This problem becomes even more severe in server-less cloud computing environments where applications are automatically deployed and invoked. Cloud service providers resolve and provision the required dependencies for a given application. The result of this is that the application owner has to give up control over the exact folder structure or dependency source code since it is opaquely downloaded by the cloud service.

1.1 Problem

Dependency-based attacks can be launched by third-party dependencies on a host Node.js application. This report studies the existence of dependency-based attacks in Node.js environment.

The malicious dependency is assumed to contain code that performs actions that the host application developer did not expect. Such actions could modify JavaScript core features, interrupt the Node.js execution environment or leak sensitive data. The malicious code might spread and duplicate itself such that it hides in unexpected and unpredictable locations.

Performing attacks discussed in this report requires that the attacker successfully manages to get malicious code installed in the victim application through distribution via Node Package

Manager (NPM) or modifying an existing dependency in NPM. Threat model is focused around the attack vectors such as - overtaking the host application, manipulating or leaking data processed in the application and interrupting services provided by the application. Whether the dependency is loaded directly by the host application or indirectly as part of another dependency does not matter.

1.2 Approach

Reported Node.js vulnerabilities were analyzed with the goal to identify the relation to dependency between packages.

Static code analysis is performed using the open source analysis framework - T.J. Watson Libraries for Analysis (WALA). Attacks are classified into five different categories. Using WALA, analysis is designed and implemented for each of the identified attack categories. Control Flow Graphs are extracted from the input source code and dataflow analysis is performed using dataflow functions provided by WALA. These concepts are explained in detail in chapter 5 of this report.

A set of NPM packages were analyzed using the detection framework. Manual analysis was performed to verify the results of automation. In this thesis, we only focus on Node.js applications written in JavaScript.

1.3 Contribution

Main contributions of this thesis are -

- Analysis of existing vulnerabilities to understand their relationship to dependency concept.
- Application of static code analysis on a set of packages and manual analysis of source code for verification of automation results.
- Recognition of vulnerability in one of the dependencies of the analyzed packages during manual analysis. This vulnerability can be exploited to launch a dependency-based attack.

1.4 Organization

Basic details about Node.js, Node Package Manager (NPM), third-party dependencies in Node.js and T. J. Watson Libraries for Analysis (WALA) are discussed in chapter 2 of this report. Knowledge of these concepts is required for better understanding of the presented work.

Chapter 3 provides analysis of vulnerabilities or attacks that have been recorded in Node.js. It provides an overview of different categories of vulnerabilities.

In chapter 4, the five categories of dependency-based attacks are explained in detail. These categories are: Global Manipulation, Global Leakage, Local Manipulation, Service Interruption and Dependency Tree Manipulation. Examples are provided by including code snippets containing malicious code which can be detected by our framework.

Next chapter provides details about the static code analysis built on top of WALA APIs. Topics of context-sensitivity, intermediate representation and basic blocks, control flow graphs and dataflow analysis are explained in detail in this chapter. Also, an overview of the analysis framework is provided.

The attack detection framework was applied to a set of packages in NPM. Results of this analyses are discussed in chapter 6 along with the discovered vulnerability. Finally, conclusion and future work are presented in chapter 7.

CHAPTER 2. BACKGROUND

This chapter provides background information on the following topics - Node.js, Node Package Manager (NPM), third-party dependencies in Node.js and T. J. Watson Libraries for Analysis (WALA). Knowledge of these concepts is required for better understanding of the presented work.

2.1 Node.js

Node.js [1] is a server-side language built on top of Google Chrome's v8 engine. It uses event-driven non-blocking I/O which makes it a perfect candidate for data-intensive applications. The core of Node is JavaScript. Node inherits weaknesses of JavaScript. JavaScript runs in browser environment whereas Node.js is executed using v8 engine on a server. This impacts the attack surface of applications in Node.js.

2.2 NPM

NPM stands for Node Package Manager. It is the online repository through which Node.js packages are uploaded and shared. It is currently the largest software repository consisting of total 742,509 packages with approximately 3 billion downloads per week [2].

2.3 Third-party dependencies in Node.js

Third-party dependencies are very common in Node.js applications. Primary dependencies can also have their own dependencies (indirect dependencies). Hence, the total number of dependencies for a Node.js application can easily exceed several hundred.

These dependencies are critical for application security since they have the same access level to the environment as the main application. There have been concerns about the same issues :

security and third-party dependencies. According to a recent survey by NPM, 77% of respondents were concerned with the security of Open Source Software (OSS) and third-party code [3].

2.4 Server-less Cloud Computing

In server-less cloud environments, dependencies are provided to the host application at run-time. The advantage of this approach is that only application-specific source code has to be uploaded to the cloud and the amount of data on the cloud server is minimized. However, the application owner does not have control over the dependency source code since its opaquely downloaded by the cloud service.

2.5 Injection of Malicious Dependency

Figure 2.1 depicts various attack vectors for injection of dependency. An attacker can upload a new dependency on NPM or modify an existing dependency. The aim of the attacker is to enter the dependency tree of an application. The dependencies can either be bundled by the cloud user and uploaded as a package, or automatically resolved by the cloud platform based on the package.json file. In either cases, the malicious dependency gets downloaded from a public repository, for instance NPM or GitHub. If the attacker decides to start a new dependency, he can promote it publicly using social engineering techniques. The application developer needs to review any newly installed dependency and its dependencies to make sure none of the code introduced to the application bundle performs undesired actions. However, this approach can be time-consuming if a large number of dependencies need to be reviewed [4].

2.6 T. J. Watson Libraries for Analysis (WALA)

WALA is an open-source library written in Java for static and dynamic program analysis of Java and JavaScript code. It is initially developed by IBMs T.J. Watson Research Center and was donated to open source community in 2006 under Eclipse Public License [5].

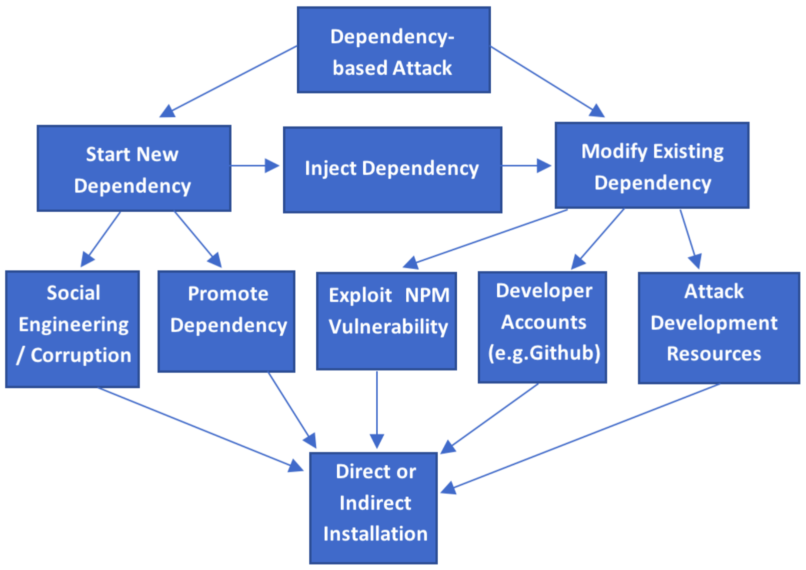
Dependency-based Attack
Start New Dependency
Inject Dependency
Modify Existing Dependency
Social Engineering / Corruption
Promote Dependency
Exploit NPM Vulnerability
Developer Accounts (e.g.Github)
Attack Development Resources
Direct or Indirect Installation

Figure 2.1 Attack vectors for dependency-based attacks.

WALA is built around the following key design goals : robustness, efficiency and extensibility. Main features provided by WALA include pointer analysis, call-graph construction, interprocedural dataflow analysis framework, multiple language front-ends and generic analysis utilities/data structures [6]. WALA source code is available on GitHub [7].

CHAPTER 3. SURVEY OF VULNERABILITIES AND ATTACKS FOUND IN JAVASCRIPT APPLICATIONS

We conducted an extensive survey to analyze the different types of vulnerabilities that have been reported in Node.js applications.

Total 36 types of vulnerability/attack categories have been recognized with 585 total number of vulnerabilities/attacks found in Node Security Platform [8]. This data is collected as of 8th Aug 2018. Table 3.1 shows the list of categories in the descending order of number of vulnerabilities/attacks found for each category. (Note - This work is ongoing and more categories may be added in the future to this survey).

Top 10 of the 36 categories listed in Table 3.1 consist of around 90% of the total reported vulnerabilities. Figure 3.1 compares the number of vulnerabilities of these top 10 categories and table 3.2 contains the descriptions for these attack categories.

Amongst the top 10 categories of vulnerabilities/attacks shown in table 3.2, four categories : cross-site scripting, command injection, code execution and SQL injection occur due to weakly written code. But the following three categories of attacks: Denial of Service, hijacked environment variable and prototype pollution can be executed by malicious code in third-party dependencies. Our detection framework aims to detect these types of attacks.

The total 585 vulnerabilities were also classified into client-side and server-side depending on where they occurred. These are shown in Table 3.3.

JavaScript is used to write client-side scripts and Node.js is used for server-side programming. As seen from Table 3.3, most of the vulnerabilities occurred on the server-side. This points to the fact that server-side code written in Node.js is more vulnerable to attacks as compared to browser-side JavaScript code.

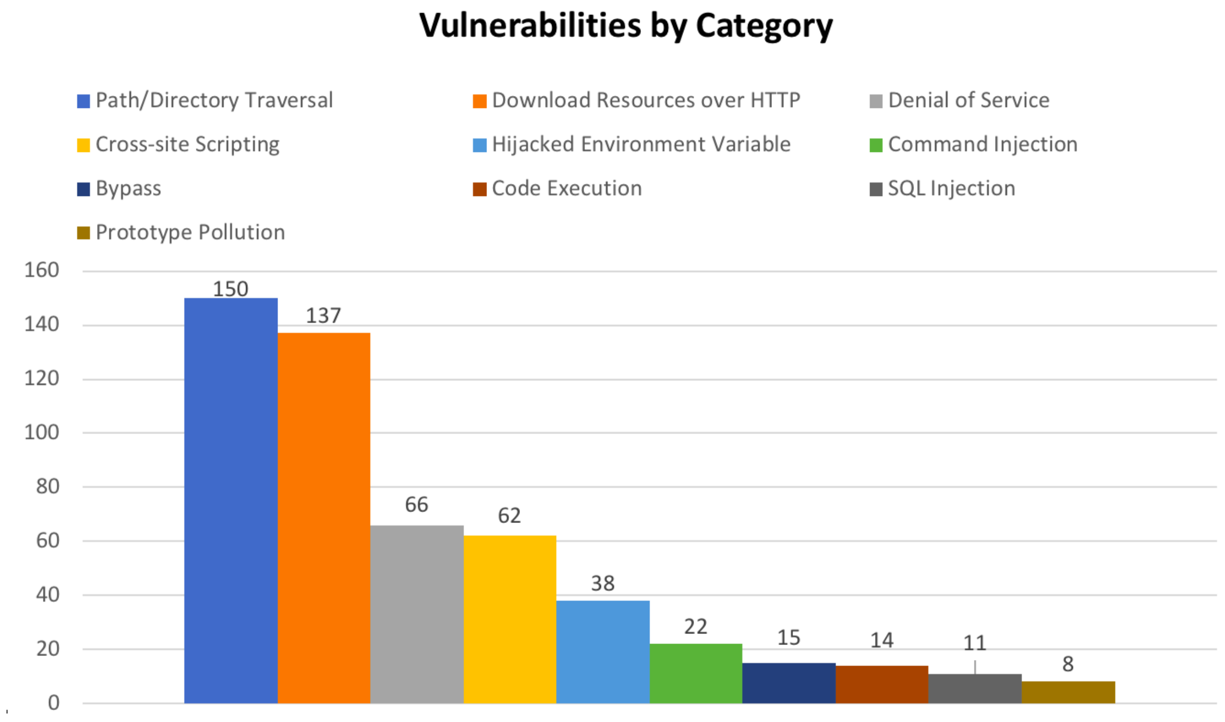

Figure 3.1 Number of vulnerabilities by category.

Table 3.1 The 36 categories of vulnerabilities.

Category	#Vul	%	Category	#Vul	%
Path/Directory Traversal	150	25.64	Prediction Attack	3	0.51
Download Resources over HTTP	137	23.42	Insecure Temp Files	3	0.51
Denial of Service	66	11.28	Malicious Script	2	0.34
Cross-site Scripting	62	10.60	Remote Memory Disclosure	2	0.34
Hijacked Environment Variable	38	6.50	LDAP Injection	2	0.34
Command Injection	22	3.76	Timing Attack	2	0.34
Bypass	15	2.56	Information Exposure	1	0.17
Code Execution	14	2.39	Rosetta-flash Attack	1	0.17
SQL Injection	11	1.88	Identity Spoofing	1	0.17
Prototype Pollution	8	1.37	Silently Run Cryptocoin Miner	1	0.17
Out-of-Bounds Read	7	1.20	Invalid Curve Attack	1	0.17
Remote Memory Exposure	5	0.85	Insecure Defaults	1	0.17
Exfiltrates Data	5	0.85	Cross-site Socket Forgery	1	0.17
Information Leakage	5	0.85	Preflight Request Headers	1	0.17
Malicious module	4	0.68	Insecure Comparison	1	0.17
Token Disclosure	4	0.68	File Overwrite	1	0.17
Open Redirect	3	0.51	Connection Overrides	1	0.17
Memory Exposure	3	0.51	Token Injection	1	0.17
Total #Vul = 585					

Table 3.2 Description of the top 10 vulnerability/attack categories.

No.	Attack Category	Description
1	Path/Directory Traversal	Allows attackers to access restricted directories and execute commands outside of the web server's root directory [9].
2	Download Resources over HTTP	Unauthorized download of resources by attacker by taking advantage of security loopholes of http.
3	Denial of Service	Makes a machine or network resource unavailable to its intended users by temporarily or indefinitely disrupting services of a host [10].
4	Cross-site Scripting	Enables attackers to inject client-side scripts into web pages viewed by other users [11].
5	Hijacked Environment Variable	Stealing environment variables and sending them to attacker-controlled locations.
6	Command Injection	Execution of arbitrary commands on the host operating system via a vulnerable application [12].
7	Bypass	Allows remote attackers to bypass authentication via a crafted token [13].
8	Code Execution	Attacker executes arbitrary code on the victim machine by taking advantage of weakly written program.
9	SQL Injection	Allows insertion of SQL statements into entry fields of data driven applications for execution by the query engine [14].
10	Prototype Pollution	Attackers can add or modify existing properties relating to an Object by using the utilities function to change the prototype of said Object [15].

Table 3.3 Total vulnerabilities divided into client-side and server-side depending on where they occurred.

Category	Number	Percentage
Client	250	42.74 %
Server	334	57.09 %
Total	584	100 %

CHAPTER 4. DEPENDENCY-BASED ATTACKS

This report focuses of five different categories of dependency-based attacks. This chapter explains them in detail. The five categories are - Global Manipulation, Global Leakage, Local Manipulation, Service Interruption and Dependency Tree Manipulation. Code snippets containing malicious code are included to provide examples of how these attacks can be launched.

4.1 Global Leakage

This attack is performed by the malicious dependency by accessing the global name-space and public data structures of the execution environment in order to leak them to an attacker-controlled server. As an example, a dependency can access and modify the security credentials provided by cloud systems through global variables. Although it is fairly simple to leak data to malicious server, it is difficult for automated systems to detect this type of attack. Listing 4.1 shows an example of global leakage attack. [4]

4.2 Global Manipulation

The goal of this type of attack is to manipulate a globally accessible value or function to alter the application behavior. This type of attack is also known as Global Namespace Pollution. JavaScript by design makes heavy use of global variables. This property makes the attack highly probable.

One example of manipulation of functions is monkey-patching, where original implementation of a function is overridden by the malicious version provided by attacker. The original function no longer behaves as expected and this can be used to execute unwanted code. An example of global manipulation attack is shown in listing 4.2 [4].

Listing 4.1 Demonstration of the global leakage attack where data is sent through http request.

```
 function leak(payload) {
        var URL = require('url');

        var url = URL.parse(process.env.POC_LEAK_URL || '');
        if (!url || !url.protocol) {
                console.log('Global Leakage PoC: Invalid POC_LEAK_URL,
                cannot leak:', payload);
                return;
        }

        var request = url.protocol === 'http:' ? require('http').request :
        require('https').request;
        var postData = JSON.stringify(payload);

        var options = {
                hostname: url.hostname,
                port: url.port,
                path: url.path,
                method: 'POST',
                headers: {
                        'Content-Type': 'application/json',
                        'Content-Length': Buffer.byteLength(postData)
                }
        };

        var req = request(options);
        req.write(postData);
        req.end();
}

leak(process.env);
```

Listing 4.2 Example of Global Manipulation attack where original function is overridden by monkey-patched implementation.

```
{
   function MyClass() {};
   MyClass.prototype.someFunction = function () {
    // Initial implementation
};
function performMonkeyPatch() {
var originalFunction
= MyClass.prototype.someFunction;
MyClass.prototype.someFunction = function () {
        // New monkey-patched implementation
        // originalFunction can be invoked here
}; }
var c = new MyClass();
c.someFunction(); // Initial implementation performMonkeyPatch();
c.someFunction(); // Monkey-patched implementation
}
```

4.3 Service Interruption

This attack aims at congesting the event loop of the JavaScript engine by asynchronous function invocations and can also be achieved using monkey-patching.

This is a type of Denial of Service (DoS) attack. Initially, the application will run at expected speed, but after a given point of time, the event loop is polluted enough to noticeably slow down the application execution. In a short amount of time, the event loop gets flooded and finally gets suspended. To recover, the application it has to be restarted completely. Listing 4.3 shows an example of global leakage attack [4].

4.4 Local Manipulation

This attack targets local properties that are only accessible from the context of the caller. Node.js scoping is based on standard JavaScript scoping and hence is not flawed. However, Node.js mechanism for loading additional modules has weaknesses and can be misused.

Listing 4.3 Example of Service Interruption attack where delay grows quadratically with time, finally resulting in suspension of event loop.

```
  (function interrupt() {
        var f = setTimeout;
        f(interrupt);
        f(interrupt);
})();
```

Modules are cached after being loaded for the first time. If any file from the package calls the module, same object is returned from cache. This is referred to as loaded module cache. This feature is a security weakness.

As an example of this type of an attack, consider a main file that requires dependencies A and B. When dependency B gets loaded, it can also require dependency A and gets the exact same object returned as the main file. In this situation, the main file and dependency B both have a object in their local scope which is shared between both files. Unfortunately, this object is writable. Any modification is reflected to any file that also requires that dependency.

To conduct a local manipulation attack, the order in which dependencies are loaded does not matter. In any case, all files that require a certain dependency get the same reference to an object returned. Whether a victim file or the malicious dependency requires the file first, is not relevant. [4].

4.5 Dependency Tree Manipulation

This type of attack takes advantage of the fact that in Node.js, all dependencies are loaded into cache. It focuses on manipulating the cache so that the malicious dependency prevents the benign dependency from being loaded. First, the required cache is artificially populated with a forged value. From that point on, any file that requires original dependency gets the forged value from spoiled cache. Hence, the order in which dependencies load into cache is very important for this attack to be accurately executed. See listing 4.4 for an example of dependency cache manipulation.

Listing 4.4 Example of Dependency Tree Manipulation attack.

```
{
    require(‘./malicious-lib’);
    require.cache[require.resolve(‘victim-lib’)] =
    require.cache[require.resolve(‘./malicious-lib’)];
}
```

In listing 4.4, reference of ‘victim-lib.js’ is overwritten by reference of ‘malicious-lib.js’. [4]

CHAPTER 5. STATIC CODE ANALYSIS

This chapter provides details about the static code analysis built on top of WALA APIs. Some of the concepts used for analysis such as context-sensitivity, intermediate representation and basic blocks, control flow graphs and dataflow analysis are explained in detail. In the end, overview of the analysis framework is provided.

5.1 Context-sensitivity

Context-sensitive interprocedural analysis can be used to study data propagation throughout the code. When information is propagated through procedure boundaries while respecting the actual control flow of the application, it is known as interprocedural analysis. Context-sensitive interprocedural analysis keeps information gathered on different paths separated. For each invocation site of a function, function internal details are kept separate from other invocation sites. As an example, if a function returning a value is invoked at two different call sites, and if the first invocation returns a tracked value and the second invocation returns an irrelevant value, context-sensitive analysis would separate these invocations and infer that the returned value from first invocation also needs to be tracked, while the return value from second invocation can be discarded.

5.2 Analysis Steps

5.2.1 Intermediate Representation

The program being analyzed is converted to Intermediate Representation (IR) form which reduces the complexity of the source code into a low level form. A type of IR is the Three-address code representation of source code where more complex statements can be represented by a combination of multiple three-address code instructions. Listing 5.1 shows an example of three-address representation of the statement “var r = a + f(b, c) + d”.

Listing 5.1 Example of three-address form of Intermediate Representation (IR).

```
For eg : Three-address code representation of the statement -
var r = a + f(b, c) + d
param b
param c
t1 = invoke f, 2
r = a + t1
r = r + d
```

Listing 5.2 Example of SSA form of IR.

```
For eg : SSA transformed code of the statements
   var r = a + f(b, c) + d; return r == 0 ? 1 : 0;

v27 = invoke f b,c
v26 = binaryop(add) a,v27
r = binaryop(add) v26,d
v32 = binaryop(eq) r,0
conditional branch(eq, to iindex=8) v32, 0
v33 = 1
goto (to iindex=9)
v35 = 0
v11 = phi v35,v33
return v11
```

This thesis implements another form of IR known as Single Static Assignment (SSA) which does not have a restriction on the number of variables per instruction.

Listing 5.2 shows an example of SSA form for statements “var r = a + f(b, c) + d; return r == 0 ? 1 : 0;”. Advantage of using SSA form for analysis, is that each modification of a variable creates a new version of the variable and the old version remains unchanged. This ensures that value of a specific version of a variable is constant in a given context after definition. This makes it easier for static analysis and to track a given value which is held by some variable at a particular time.

5.2.2 Basic Block

Basic block (BB) is a sequence of consecutive instructions that are always executed successively and do not include any jumps, neither in nor out of the sequence.

Instructions inside a BB can be treated similar and can be grouped and handled as one single basic block instead of dealing with each instruction separately. This can lead to significant improvement in performance. However, for convenience and accuracy, this analysis treats every instruction as a BB. Hence, every BB only contains a single SSA instruction.

5.2.3 Control Flow Graph (CFG)

A Control Flow Graph (CFG) is a directed graph $G = (V, E)$ with vertexes V and edges E . Each basic block of a procedure is represented as exactly one vertex. Each vertex is represented in the form of Call Graph Node (CGNode). Any edge from vertex $V1$ to vertex $V2$ denotes a jump in the basic block represented by $V1$ to the basic block represented by $V2$.

For each vertex v, there are set of predecessors and successors depending on whether the edge is ending at, or originating form v. Hence, jumps between basic blocks are encoded in this CFG. Vertexes that don't have predecessors are classified as entry points and those that dont have successors are classified as exit points.

A CFG denotes one individual procedure. All CFGs together create the supergraph which represents an entire application and is the foundation of an interprocedural analysis.

5.2.4 Dataflow Analysis and facts

Dataflow analysis is used to propagate values of interest through the program and use the result to identify potential violations. Dataflow analysis uses the concept of facts - information attached to instructions of the program. Depending on already existing facts, new facts are computed and attached to other instructions on the path. Initial seeds are facts that are attached to certain instructions before starting the computation of further facts based on those initial seeds. These initial facts are an essential element of a data flow analysis, since a missed fact can result in an false negative.

5.2.5 Interprocedural Finite Distributive Subset Framework

Interprocedural Finite Distributive Subset Framework (IFDS) algorithm invokes dataflow functions on facts and uses resulting value for subsequent steps. It maintains a worklist of facts on which dataflow functions are yet to be invoked. First initial seeds are identified, which act as starting points for the actual analysis. Next, dataflow functions are used to propagate dataflow facts through control flow graph (CFG). New facts can be generated during the analysis which are inserted in worklist and acted upon later. Dataflow analysis can be intraprocedural or interprocedural, based on whether it is derived from CFG of individual function or the supergraph which consists of all CFGs.

5.2.6 Analysis Framework Overview

Analysis steps are summarized below -

- Source code is converted to Single Static Assignment (SSA) form.
- Every instruction in SSA form is stored as one Basic Block (BB).
- Basic Blocks are used to create Control Flow Graphs (CFGs) and supergraph.
- Pointer analysis is performed using Basic Blocks.
- Flowfacts are derived from Control Flow Graphs and supergraph.
- Dataflow analysis is performed by propagating facts through CFGs using dataflow functions from IFDS algorithm.
- Results of dataflow analysis are used by the five attack detection algorithms to identify attacks.

Figure 5.1 shows the analysis framework.

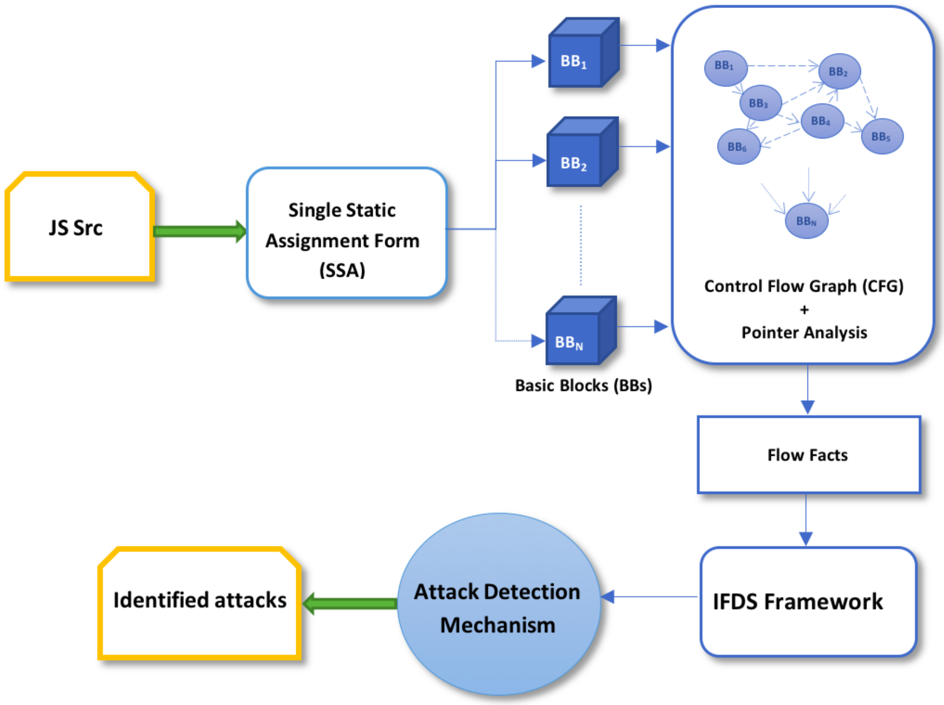

Figure 5.1 Overview of Analysis Framework.

CHAPTER 6. ANALYSIS OF NPM PACKAGES

We applied out static code analysis on a set of packages in NPM to check the existence of dependency-based attack in Node.js environment. This chapter discusses the results. These are followed by the description of the vulnerability that was discovered in one of the packages.

6.1 Analyzed Packages

We selected five packages for analysis. Some of the packages (for example mongoose) have large number of dependent applications and some packages like cross-env have similar functionality as that of the attack categories targeted in this report. Both of these are security critical modules. For example, cross-env package runs scripts that set and use environment variables across platforms. This points to the fact that this package uses global environment variables and is highly useful for our analysis.

Table 6.1 contains version and a brief description of the five modules that were analyzed.

Table 6.1 Description of analyzed modules.

Package	Version	Description
cross-env	5.2.0	Runs scripts that set and use environment variables across platforms. [16]
mongoose	5.1.5	MongoDB object modeling tool designed to work in an asynchronous environment. [17]
reduce-css-calc	1.2.4	Aims to reduce CSS calculations to the maximum. [18]
pouchdb	6.0.3	Pocket-sized database. [19]
mobile-icon-resizer	0.4.2	Used to resize iOS and Android application icons in batch. [20]

Listing 6.1 Code snippet containing the vulnerability.

```
    var isolateEval = function (functionString) {
        // Contains the value we are going to set
        var value = null;
        // Eval the function
        eval('value = ' + functionString);
        return value;    };
```

6.2 Analysis Results

We analyzed 5 packages containing total 87 files. Our analysis detection framework signalled global manipulation attacks for one file in cross-env package and 8 files in mongoose package. As an example, following lines of code in the file map.js (Location : mongoose/bson/lib/bson) contribute to one of the detections - module.exports = global.Map; module.exports.Map = global.Map; This is detected due to a read/write on a global entity.

Local Manipulation attacks were flagged for 6 files in mongoose package. As an example, inside file index.js (Location : mongoose/bson), we have : 'var BSON = require('./lib/bson/bson');' Here, BSON variable references required file bson under the location lib/bson/bson. This is a local entity which is tracked for manipulations. The line of code : 'BSON.BSON-INT32-MAX = 0x7fffffff;' is considered as manipulation of local entity BSON. There are total 21 such assignments in the file resulting in 21 detections.

Table 6.2 provides the overview of these results. Manual analysis confirmed that the alerts were false positives.

6.3 Analysis of Vulnerability

Manual Analysis was conducted for packages analyzed by WALA to verify absence of false negatives. Package bson - a dependency of primary package 'mongoose' was also manually analyzed and a vulnerability was observed in files bson.js and deserializer.js. Code snippet containing the vulnerability is shown in listing 6.1.

Table 6.2 Description of Analysis Results.

Package	Files Analyzed	GM	LM	GL	SI	DTM	Attacks Flagged for files	Comments of Manual Analysis
Cross-env	5	1	0	0	0	0	variable.js	FP
mongoose	70	18	107	0	0	0	(1)	FP
reduce-css-calc	2	0	0	0	0	0	-	-
pouchdb	8	0	0	0	0	0	-	-
mobile-icon-resizer	2	0	0	0	0	0	-	-

(1) - index.web.js, objectId.js, binary.js, index.js, decimal128.js, connectionstate.js, cast.js, mongooseError.js

FP - False Positive
GM - Global Manipulation
LM - Local Manipulation
GL - Global Leakage
SI - Service Interruption
DTM - Dependency Tree Manipulation

6.3.1 Vulnerability Description

As seen in listing 6.1, function *isolateEval* takes a string as an input parameter and evaluates it using the `eval()` function. If unwanted input is passed to the function *isolateEval*, *eval*() will directly execute the input string *functionString* with host applications privileges (refer to line 6 in figure 3.1). This is highly vulnerable to arbitary code execution and use of `eval` needs to be avoided to enable greater protection. This can also result in global leakage attack if an https request is executed through the `eval` function.

6.3.2 Extending WALA to Detect the Found Vulnerability

Attack detection mechanism can be extended to observe if global variables or data structures are being overwritten or read by *eval*(). This might result in false positives for certain cases, but we can conduct a manual analysis to verify the findings.

6.3.3 Challenges Faced During Analysis

We faced two main challenges during the analysis. They are:

- If Source code contains a named function, WALA models function definition like global variable creation. Hence, even if a function is defined locally and is only accessible within current scope or children scopes, by WALAs approach, it will be listed under global variables. When these functions are invoked, WALA models it as a read of a global variable. This might create some false positives.

- Occurrence of large number of False Positives due to over-approximation during static code analysis. However, this is necessary to reduce false negatives.

CHAPTER 7. CONCLUSION AND FUTURE WORK

Node.js and JavaScript support global variables, monkey-patching and loaded modules cache by design. These features could be used to perform dependency-based attacks - global leakage, global manipulation, local manipulation, service interruption and dependency tree manipulation. Number of dependencies for a Node.js application can easily exceed several hundred as a dependency could use several other dependencies. This makes it extremely difficult for developers to review the dependencies they use for malicious behavior.

In this report, static code analysis implemented in WALA is used to detect the above mentioned attacks. Total 6 packages containing 87 files were analyzed. These are heavily relied-upon by other Node.js applications and together have more than 6000 dependent applications [2]. Automated analysis is performed followed by manual analysis of source code to make sure there are no false negatives. A vulnerability was found in files bson.js and deserializer.js with the help of manual analysis and extension of detection mechanism was proposed to detect such vulnerability.

Bibliography

[1] : Node.js v8.11.3 documentation. https://nodejs.org/dist/latest-v8.x/docs/api/

[2] : npm. https://www.npmjs.com/

[3] : Dangers of malicious modules in node.js. https://medium.com/intrinsic/common-node-js-attack-vectors-the-dangers-of-malicious-modules-863ae949e7e8

[4] Pfretzschner, B., ben Othmane, L.: Identification of dependency-based attacks on node.js. In: Proceedings of the 12th International Conference on Availability, Reliability and Security. ARES '17 (2017) 68:1–68:6

[5] : T.j. watson libraries for analysis. http://wala.sourceforge.net/wiki/index.php/Main_Page

[6] : Wala javascript tutorial. http://wala.sourceforge.net/files/PLDI_WALA_Tutorial.pdf

[7] : Wala github. https://github.com/wala/WALA

[8] : Node security platform. NodeSecurityPlatformhttps://nodesecurity.io/advisories

[9] : Directory traversal attacks. https://www.acunetix.com/websitesecurity/directory-traversal/

[10] : Denial-of-service attack. https://en.wikipedia.org/wiki/Denial-of-service_attack

[11] : Cross-site scripting. https://en.wikipedia.org/wiki/Cross-site_scripting

[12] : Command injection. https://www.owasp.org/index.php/Command_Injection

[13] : Authentication bypass. https://nodesecurity.io/advisories/151

[14] : Sql injection. https://en.wikipedia.org/wiki/SQL_injection

[15] : Prototype pollution vulnerability. https://www.sourceclear.com/vulnerability-database/security/prototype-pollution/javascript/sid-5834

[16] : cross-env. https://www.npmjs.com/package/cross-env

[17] : mongoose. https://www.npmjs.com/package/mongoose

[18] : reduce-css-calc package. https://www.npmjs.com/package/reduce-css-calc

[19] : pouchdb package. https://www.npmjs.com/package/pouchdb

[20] : mobile-icon-resizer package. https://www.npmjs.com/package/mobile-icon-resizer