In Dependencies We Trust: How vulnerable are dependencies in software modules?

Master's Thesis

Joseph Hejderup

In Dependencies We Trust: How vulnerable are dependencies in software modules?

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Joseph Hejderup
born in Tranemo, Sweden

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
http://ewi.tudelft.nl

Software AnaLysis and Testing (SALT)
Department of ECE
The University of British Columbia
Vancouver, BC, Canada
http://salt.ece.ubc.ca

In Dependencies We Trust:
How vulnerable are dependencies in software modules?

Author:	Joseph Hejderup
Student id:	4245210
Email:	`j.i.hejderup@student.tudelft.nl`

Abstract

Web-enabled services hold valuable information that attracts attackers to exploit services for unauthorized access. The transparency of Open-Source projects, shallow screening of hosted projects on public software repositories and access to vulnerability databases pave the way for attackers to gain strategic information to exploit software systems using vulnerable third-party source code.

In this thesis, we explore the character of JavaScript modules relying on vulnerable components from a dependency viewpoint. We studied the npm registry, a popular centralized repository for hosting JavaScript modules by using information from security advisories in order to determine: prevalence of modules depending on vulnerable dependencies, the propagation in the dependency chain and the time window to resolve a vulnerable dependency. This was followed by a qualitative study to understand dependency management practices in order to investigate why dependencies remain unchanged.

The outcome of this study shows that one-third of the modules using at least one advisory dependency resolve to a vulnerable version. The qualitative study suggested that a majority of the modules lacked awareness or discussion about known vulnerabilities. Furthermore, the key findings indicate that the *context use of the module* and *breaking changes* are potential reasons for not resolving the vulnerable dependency.

Thesis Committee:

Chair:	prof. dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	dr. A. Mesbah, Deptartment of ECE, UBC
Committee Member:	dr. S.E. Verwer, Faculty EEMCS, TU Delft

Preface

Over the past two years as a TU Delft student I have received incredible support and encouragement from a great number of individuals. First, I would like to express my sincere gratitude to Arie van Deursen for his excellent guidance throughout my thesis work: the great degree of autonomy and the relaxed work environment has giving me confidence to further develop myself and explore new avenues in my thesis topic. I admire his patience to listen and to give continuous encouragement in despite of countless weekly Skype meetings over the Atlantic and hundreds of half-baked proposals on my thesis work. *This master thesis has been one truly great experience!*

I would like to extend my sincere thanks to Ali Mesbah for inviting me to his research lab in UBC, Canada. His guidance and expertise in modern web-based applications have played a crucial role in forming the research direction in my thesis. His dedication towards conducting impact-based research has inspired me and I have developed a keen interest in research related to JavaScript. Besides, I enjoyed the beautiful Vancouver, BC and would love to come back soon!

Finally, I would like to take the opportunity to thank my friends Aditya, Debarshi, Prashanth, Soran and Wing who have made the study experience in Delft a great one! Special thanks goes to Wing for his insightful and lively discussions during the progress of my thesis. Last, but not least, I am grateful for the love and support my family have given me to pursue what I want in life.

Joseph Hejderup
Delft, the Netherlands
May 5, 2015

Contents

Preface **iii**

Contents **v**

List of Figures **vii**

1 Introduction **1**
1.1 Problem Statement 2
1.2 Dependency Analysis 3
1.3 Research Objective 3
1.4 Research Questions 4
1.5 Thesis organization 4

2 Background **7**
2.1 JavaScript 7
2.2 Centralized Software Repositories 9

3 Study design **13**
3.1 Methodology 13
3.2 Approach 13
3.3 Metrics used 14
3.4 Data Collection and Model 15

4 Implementation **19**
4.1 Recognizing vulnerable dependencies 19
4.2 Detecting vulnerable modules 22
4.3 Exploring cascading dependencies 24
4.4 Time latency from the release of an advisory to a patch fix 24
4.5 Dependency Analysis Tool: Rastogi.js 26
4.6 Infrastructure for processing the npm registry 28

5 Descriptive Statistics **29**
5.1 Dataset 29
5.2 Properties of the Dependency graph 29
5.3 Advisories 32

6 Known vulnerabilities captured in dependencies of npm modules **35**
6.1 Overview 35
6.2 Indirect dependency propagation of vulnerable modules 37
6.3 Time-to-update from vulnerable to vulnerable-free dependency 39
6.4 Summary 42

7 Management of advisory-related vulnerabilities in npm modules **45**
7.1 Overview 45
7.2 Comparison of the `package.json` on npm and GitHub 46
7.3 Code inspection of identified vulnerable modules 46
7.4 Security discussions within the vulnerable projects 51
7.5 Summary 54

8 Discussion **57**
8.1 Revisiting the research questions 57
8.2 Threats to validity 59
8.3 Implications 61

9 Related Work **63**
9.1 Software Security 63
9.2 Software Quality Assurance 65
9.3 Analyzing Software Ecosystems 67

10 Conclusions and Future Work **69**
10.1 Contributions 69
10.2 Conclusions 70
10.3 Future work 70

Bibliography **73**

A Glossary **77**

List of Figures

3.1 Data Model for neo4j database 17

4.1 Example of a package.json entry (in skimdb) 20
4.2 Example of an NSP advisory 21
4.3 Example of how the detection works 24
4.4 Process view of Rastogi.js 27
4.5 Overview of the distributed process for handling skimdb entries 28

5.1 In-degree and out-degree distribution for the npm graph 31
5.2 Percentile (9th to 91th) Box-plot of the out-degree distribution for *dep* dependencies 31
5.3 Distribution of the dependency chain 32
5.4 Overview of the usage for advisory modules in the npm registry 34
5.5 Download Statistics of advisory modules 34

6.1 Distribution in numbers 36
6.2 Distribution in percentage 37
6.3 Year distribution for top dependent modules 38
6.4 Overall distribution of the release year 38
6.5 A *dependency-of-dependency* graph with direct and indirect vulnerable modules 40
6.6 Findings of vulnerable dependencies for each depth level 41
6.7 Boxplots showing the *resolve-time* distribution over three time periods 43

7.1 Code snippet on how to invoke the methodOverride middleware 47
7.2 XSS filter functionality in validator.js 47
7.3 Test case showing how the vulnerability was patched in the `send` module . . . 50
7.4 Discussion in issue #438 52
7.5 Discussions in terraform module that harp depends on 52

Chapter 1

Introduction

The web has evolved at a rapid pace; from a document-driven system of static HyperText documents to a powerful runtime environment enabling sophisticated and dynamic web applications. Web content is served in a rich and dynamic fashion with highly interactive user-driven content. This has driven the success of highly popular sites such as Youtube and Twitter where roughly 3 billion Internet users[1] can instantly access and generate content.

The wide-open nature of the Internet is of grave concern to developers; the origin and character of user input data can be leveraged for abusive means and in worst case lead to information-stealing of confidential data. An intruder can detect and exploit security weaknesses such as poor sanitizing of user-supplied data and insert malicious content, known as *content injection* for unauthorized purposes.

A security breach can have devastating consequences for businesses and government services with the result of damaged reputation and loss of revenue. One of the most popular marketplace websites, eBay, was compromised in 2014 by severe *cross-site scripting* attacks where malicious scripts in auction listings redirected clients to a phishing website [5]. eBay Sellers are able to embed dynamic content in listings to make it more appealing by using JavaScript, a programming language designed for dynamically manipulating the *Document Object Model* (DOM) in the client-side of a web application. The dynamic nature of JavaScript has been the source of several security vulnerabilities over the years.

The Open Web Application Security Project (OWASP), a worldwide non-profit software security organization maintains a Top Ten list[2] for the most common web application flaws. Several of the flaws involve the use of JavaScript.

The asynchronous model of JavaScript has shown to be useful to attain high concurrency and efficiency. A server-side platform, Node.js, utilise this model to solve a critical problem in enterprise-level business applications where I/O blocking is primarily a bottleneck for scaling large network and data-driven applications [12]. Node.js is rapidly gaining territory as a full-stack solution together with the *Node Packaged Modules* (npm[3]) registry that empowers a large ecosystem of third-party libraries.

[1] `http://www.internetworldstats.com/stats.html`

[2] `https://www.owasp.org/index.php/Top_10_2013-Top_10`

[3] `https://www.npmjs.org/`

npm deploys a liberal screening process in the publication phase of a candidate artifact to the repository. A unique identifier and a version number is the minimal requirement[4]. Practically, anything can be published as there is no quality assurance towards reviewing the supplied content. Neglecting a screening process of modules, could put developers to use unknowingly modules with severe vulnerabilities. This imposes challenges in the *trust* relationship between the user and third-party artifacts available in the npm registry. Reliance on third-party artifacts is a key factor in achieving a high-degree of software reuse in a cost-effective development.

The shift towards JavaScript as a full-stack solution and its immense popularity introduce new challenges and opportunities in understanding security practices and threats. Previous research has mainly concentrated on client-side JavaScript. Thus, the central theme of this thesis is to investigate and pinpoint of potential security problems in the Node.js ecosystem with particular interest on known vulnerabilities.

1.1 Problem Statement

The event-driven style of JavaScript induces many challenges for Node.js developers with a sequential programming mindset. The lack of rich debugging facilities for developers can have a negative impact on the functionality and user experience in Node.js applications.

The loose typing and dynamic code constructs such as *eval* in JavaScript are often a source of misuse, particularly with input validation that is a potential target for XSS and CSRF attacks [26]. Furthermore, the single-threaded event loop in Node.js is more vulnerable to Denial-of-Service attacks in contrast to multi-threaded environments [22]. There are several cases due to misunderstanding that cause unintentional Denial-of-Service (DOS) vulnerabilities.

A Sonatype survey [7] suggests that software reuse of *open-source software* (OSS) components is a *de facto* industry norm with 90% of the participants using pre-existing code. Moreover, deployed OSS policies are reported to have major shortcomings with 21% enforcing policies only to use secure OSS code, 63% having no active monitoring of vulnerabilities over time and lastly, 78% having never banned the usage of certain OSS components. This poor management of third-party artifacts in software systems could be compromised by intruders.

Considering the high usage of third-party source code, the lack of policies in the publication phase to the npm repository and dependency management in applications, this could potentially open up lucrative weaknesses for attackers to exploit. The OWASP Top 10 report on ninth position: *A9-Using Components with known vulnerabilties* as a critical risk in web applications.

The identified problems are:

- JavaScript has dynamic language properties with known misuse in the client-side.
- Deployed OSS policies has no strict limitations on third-party code usage.

[4] `https://docs.npmjs.com/files/package.json`

- npm has no screening of submitted artifacts to the repository.

These problems suggests that there is clearly a need to perform a risk assessment of third-party modules in the npm repository in order to gain empirical evidence of potential security weaknesses in the Node.js ecosystem. Moreover, the lack of security policies in dependency management could potentially be of advantage for attackers.

A few research publications [22, 31] have addressed potential vulnerabilities with Server-side JavaScript. However, there is no empirical data on the spread of these vulnerabilities.

1.2 Dependency Analysis

One possible approach to perform a risk assessment of third-party components is by analyzing third-party artifacts from a source code perspective. Analysis of the source code can identify potential security vulnerabilities in code sections of the software. However, the vast majority of approaches for program analysis in client-side JavaScript leverages a combination of static and dynamic analysis. The client-side approaches require execution of JavaScript code in the browser and a crawling strategy to explore all states in the web application. This is challenging in server-side JavaScript where dynamic analysis requires semi-manual intervention to explore all code execution paths, and pure static analysis has low accuracy on call-graph construction [13]. Another critical concern is the rate of false positives and false negatives in source code analysis [13].

Therefore, the context of this thesis is to study third-party components from a dependency perspective. Previous research has shown that historical-version analysis provides a useful early identification of reliability and stability issues in third-party components [25, 18, 24]. This form of analysis aid in determining insights and trends by analyzing metadata based on library maintainers practices and patterns. In the context of security, known vulnerabilities are actively reported and maintained in data sources such as The National Vulnerability Database[5] (NVD) and The Node Security Project[6] (NSP). These advisories help system maintainers to identify potential problems with affected dependencies in software systems.

1.3 Research Objective

The transparency of open-source projects and shallow screening of hosted projects on the npm registry could potentially be leveraged for attackers to gain strategic information to exploit systems based on weaknesses in third-party software libraries. The availability of known vulnerabilities on the Internet can be leveraged by attackers to identify and exploit modules where the library maintainer has not patched a known security hole in a dependency. An attacker can achieve this by using information from npm, GitHub, and security projects.

[5]`https://nvd.nist.gov/`

[6]`https://nodesecurity.io/`

The goal of this thesis is to establish the presence of modules depending on security vulnerabilities in the npm registry and security practices in dependency management to better understand why library maintainers are not patching or dealing with security problems.

1.4 Research Questions

In order to tackle our research objective, we define three research questions to identify security weaknesses and security malpractices with third-party modules in the npm repository. The target is on declared dependencies in third-party modules and known vulnerabilities:

RQ1 What is the prevalence of modules that use at least one dependency that is disclosed as vulnerable?

This research question tries to establish how widespread known vulnerabilities are used as dependencies in npm modules. Furthermore, the negligence among library maintainers concerning security advisories is further studied in identified npm modules to qualitatively pinpoint on areas of concern.

RQ2 What is the cascading effect of modules depending on at least one vulnerable module?

The second research question aims to determine the cascading effect of identified modules from **RQ1**. This determines whether library maintainers unknowingly depend on a module with a propagated vulnerability in the "dependencies of dependencies" chain.

RQ3 What is the time latency for updating to a non-vulnerable version range for a dependency?

The last research question investigates the time it takes for library maintainers to update dependencies to a vulnerable-free patch-fix. The Drupal security team detected a critical security flaw that required users to upgrade to a patched version within in seven hours to avoid being compromised [6]. Hence, this shows how critical the response time is for updating to a patch. A problematic factor is that attackers could potentially at the advent of a new advisory have the time window to perform attacks knowing that library maintainers require a long time window to resolve the vulnerable dependency.

In answering the research questions, statistical data is mixed with qualitative data in order to establish the presence of vulnerable modules with potential explanations.

1.5 Thesis organization

The remainder of the thesis is organized as follows. Chapter 2 introduces necessary concepts and background to the thesis. Chapter 3 presents the approach to tackle the research questions. Chapter 4 presents the implementation and the tool `rastogi.js`. Chapter 5 presents the descriptive statistics. Chapter 6 presents the quantitative results followed by Chapter 7 with the qualitative study. Chapter 8 includes answers to the research questions, discussion of the results, threats to validity and future work. Finally, Chapter 9 presents related work and Chapter 10 the conclusions of this thesis work.

Chapter 2

Background

This chapter introduces the relevant background information to understand the necessary concepts and ideas discussed throughout this thesis. Particular emphasis is put on the `Node.js` ecosystem.

2.1 JavaScript

The core technology that acts as a powerful catalyst in the sphere of rich web frameworks is JavaScript, a programming language designed for rendering dynamic content on the client-side of a web application. The asynchronous model of JavaScript is tuned for interacting with the DOM-tree in the browser and dynamically updating content based on user interactions and data transfers using `XMLHttpRequests`. This is commonly known as *Asynchronous JavaScript and XML* (AJAX), that sparked the development of rich and interactive web applications.

Initially, JavaScript runtime engines were merely interpreters with slow rendering capabilities. The urge for a faster and richer browsing experience has challenged browser makers to improve the performance of the JavaScript runtime environment. A milestone in this ongoing quest for faster JavaScript engines was reached in 2008 when Google introduced the JavaScript V8 engine that leverages Just-In-Time Compilation, a technique that instantly translates JavaScript code into native machine code [3]. This achievement had a significant impact on the JavaScript execution time. In 2007, to run the SunSpider JavaScript benchmark took 5.4 seconds. As of 2014, this has decreased to 153 milliseconds, 35 times faster in a time-span of seven years[1].

2.1.1 Node.js

The active development and performance improvements of JavaScript runtime engines shows that it can be applicable in a non-browser context. The asynchronous model of JavaScript provides a responsive and powerful interface for dynamically manipulating the *Document Object Model* (DOM) in the browser. The event-driven design of this model includes an

[1]`https://twitter.com/codinghorror/status/496451816331042816`

event loop, an important event-based language construct to achieve high concurrency and efficiency. A server-side platform, Node.js[2] is the result of leveraging this model together with Google's JavaScript execution engine V8 to achieve a light-weight and non-blocking I/O platform for scalable network applications [12].

Node.js is emerging in enterprise-level businesses where I/O blocking is primarily a bottleneck for scaling large network and data-driven applications. One of the largest payment processing companies, PayPal developed their account overview page using Node.js. In a report[3], it is stated that 33% fewer lines of code was written, built twice as fast but with fewer developers and 40% fewer files needed in comparison with a similar Java Application developed in parallel.

The advantages of using Node.js are three-fold:

- **Code re-use:** Node.js provides the option of a full-stack solution where code can be shared both on the client and server-side.
- **Less developers:** Front-end developers can be deployed on the back-end
- **High productivity:** Node.js is ideal for prototyping solutions.

Disadvantages with Node.js:

- **JavaScript:** Due to the lack of debugging facilities, asynchronous programming model can be challenging for inexperienced developers.
- **Event-loop:** The event-loop is ideal for scaling network-based applications. However, in compute-intensive operations it is a bottle-neck.
- **Maturity:** Node.js is recently being widely deployed and explored on the back-end. Technologies such as Java or PHP are mature with large community resources.

2.1.2 Security

The asynchronous design of JavaScript is manifested in the various code constructs and use-cases on the client-side for achieving a high-degree of flexibility in the language. For instance, the use of `eval()` can invoke and execute JavaScript code from user-input on the web page. A dangerous scenario is improper validation of the user supplied content where an attacker could compromise the website.

JavaScript has limited I/0 capabilities; the significant threat lies on the integrity of the data exchanged from the user and the domain. A particular area of concern is the interaction and integration with third-party data sources in the web application; a malicious third-party library could gain unauthorized access to data.

Traditionally, third-party JavaScript code is executed with the same rights and privileges as the code from the domain. Modern browsers are equipped with mitigation techniques to restrict execution rights of third-party JavaScript code. A significant contribution is the

[2]`http://nodejs.org/`

[3]`https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/`

same-origin-policy that isolates and limits the communication and interaction across domains in the browser [34]. In practice, a web page from TU Delft is not authorized to manipulate the DOM and read data from a different page such as Facebook on the same browser.

The *same-origin-policy* restricts access on Cookies, DOM Access, AJAX requests and internal data storage. However, there are several ways to by-pass this policy. In an *iFrame*, changing the property `window.document.domain` to the same domain as the parent would circumvent this policy. For AJAX Requests, using JSONP or having Cross-origin resource sharing can give access to an external remote file.

Cross-Site Scripting

The security weaknesses as mentioned earlier within JavaScript and the browser model is known as *Cross-Site Scripting (XSS)*. This allows a third-party source to gain access to content on another domain. Below is a description of the various types of XSS attacks [4]:

- **Stored XSS (Type-I)**: This form of XSS is triggered when a client retrieves tampered data from stored data content such as in databases and messaging forums where malicious JavaScript is stored and invoked in the client's browser.
- **Reflected XSS (Type-II)**: This is a non-persistent form of XSS where improper sanitizing is performed on URL parameters or HTML forms. Some form of social engineering is required to conduct this attack by providing a tampered URL including malicious code that will be executed in the website.
- **DOM Based XSS (Type-0)**: This form of attack is tightly connected with the client-side of the web application. The exploit is achieved where an attacker can control an execution point in the DOM (e.g., `document.baseURI`) from an input.

Node.js is designed for the server-side. The mentioned exploits are applicable on the server-side, in particular the injection and Denial of Service (DoS) attacks [26].

2.2 Centralized Software Repositories

Software reuse is a crucial pillar for a cost-effective and time-efficient development of a software system. The high dependence on external source code and the plethora of available software components introduced challenges in management and maintenance on dependencies. The challenges are two-fold: (1) *developers* are required to download manually and maintain the dependency. This can be a tedious task for outdated modules that are difficult to find on-line. Moreover, manual effort is required to update the library. (2) *publishers* need a distribution channel for a simplified way to release and update the software component.

This has lead to the emergence of centralized software repositories that provides a simplified and robust management and distribution center for software components. A notable example is MAVEN Central that hosts JVM-based projects. Developers that use Java as a development language can just create a `POM.xml` file where the needed dependencies are declared and automatically resolved. Examples of other central repositories are Rubygems for Ruby, nuget for .NET and CRAN for R.

2.2.1 npm

The *Node Packaged Modules* (npm[4]) registry empowers the large ecosystem of third-party web frameworks and libraries that serve as the backbone for most Node.js-based applications. As of 3 April 2015, npm has over 130,000 hosted modules, making it the largest software repository to date. npm has a liberal policy towards hosting packages on the registry, and the only requirement is a package identifier together with a semantic version number. This is declared in a `package.json` file, located in the root of a npm project.

In order to install a module (or use a dependency), a user is required to run `npm install MODULE_NAME`, the `package.json` in `MODULE_NAME` is read and the declared dependencies are resolved and downloaded. For publishing a module, the user needs first to register, create a *package.json* file with necessary details in the root folder and finally run in the package directory `npm publish`.

2.2.2 Semantic versioning

Semantic versioning is a scheme that provides guidelines for publishers of an artifact to indicate the changes made in a released software component. The numerical format is MAJOR.MINOR.PATCH and signifies the following (with example):

- **MAJOR:** The changes that are made need not to be backward-compatible with the previous version of the API.
- **MINOR:** The changes are made in a backward-compatible way.
- **PATCH:** Backward-compatible bug fixes.

npm employs semantic versioning[5] for releasing a package. Dependencies in npm modules have the option to be declared using advanced range syntax and is the default mode when automatically saving dependencies in the `package.json` by executing `npm install MODULE_NAME --save`. The following are the three advanced syntax ranges (with an example on the right-hand side of the comma):

- **Caret-Range:** `^1.2.3`
- **Tilde-Range:** ∼`1.2.3`
- **X-Range:** `1.2.x`

The tilde range matches the most recent minor version, an equivalent interval range is $>= 1.2.3 < 1.3.0$. The caret range will match the most recent major version, in the above

[4]`https://www.npmjs.org/`

[5]`semver.org/`

scenario the equivalent interval range is $>= 1.2.3 < 2.0.0$. npm maintains a utility library for semantic versioning, named `semver`[6]. `semver` includes features such as validation of version range and conversion of advanced syntax range to interval range.

[6]`https://github.com/npm/node-semver`

Chapter 3

Study design

3.1 Methodology

The research objective presented in Chapter 1 calls for a research strategy that uncovers security implications in dependency management of npm modules. Keeping the objective in mind, the research questions in section 1.4 attack this on two fronts:

- assessing the *presence* of modules using at least a vulnerable dependency in the npm registry
- security practices in dependency management for modules that are or were previously affected by a vulnerable dependency.

This guided our study towards an exploratory mixed-method approach where quantitative data are further explained by qualitative data. This is an exploratory study as it serves as an initial investigation of security practices in dependency management.

3.2 Approach

Our research approach consists of three phases:

Preliminary phase

1. Explore and identify security advisories compatible with npm modules.
2. Devise a crawling strategy to extract dependency information modules in the registry.
3. Create a snapshot of the npm registry in order to achieve reproducibility of the study.

Quantitative phase

1. implement a *vulnerability scanner* with the advisory data as the search criteria and the npm registry as the study object. In order to have reliable results, the scanner should have low *false positives* and *false negatives* rates.

2. collect the results and perform early statistical analysis; for example identify vulnerable modules with the highest downloads, most used as dependency in other projects and most starred project on GitHub
3. implement an extension of the vulnerability scanner to assess the depth and width of the “dependencies of dependencies” chain for the *cascading effect*.
4. perform similar data analysis operation on the results (as in step 2) and measure the depth of the chain and width (number of vulnerable modules for each level in the chain)
5. implement historical-version analysis features in order to inspect modules that used or still use a dependency that was flagged by a security advisory.
6. Measure the time latency of updating from a vulnerable version to a non-vulnerable version.

Qualitative phase

1. Modules based on attributes that express popularity such as *high-downloads*, *most dependent*, *GitHub stars* are selected for further inspection.
2. Inspect source code, communication, and discussion for each module in an explorative fashion to discover potential security malpractices; awareness of the vulnerable dependency and related actions (such as patch fixes, future plans) are particularity inspected.
3. compile the discoveries and report on the findings with supporting data from the qualitative and quantitative phases.

3.3 Metrics used

The following metrics are collected for each research question in the quantitative part of the study:

- **RQ1:** number of identified modules using at least one or more vulnerable dependencies
- **RQ2:** same as **RQ1** including the depth (length) of the dependency chain and width of vulnerable modules for each level in the chain.
- **RQ3:** the update latency from the publication day of an advisory to the first non-vulnerable version of the module is measured in terms of days between the two timestamps as shown below:

$$\textit{update\_latency} = \textit{timestamp\_vulnfree\_ver} - \textit{adv\_pub\_timestamp}$$

3.4 Data Collection and Model

The study is data-centered emphasizing on data processing and analysis. The approach requires multiple data sources and a data model for answering the research questions.

3.4.1 Data Sources

Security advisories

The National Vulnerability Database (NVD) maintains more than 60,000 Common Vulnerabilities and Exposures (CVE[1]) and would be the ideal candidate as a data source. However, only a handful of Node.js-related CVE's are available[2] Therefore, the Node Security Project (NSP)[3] is the data source for security advisories. The NSP maintains a public list with verified and brief descriptions of security exploits in the advisories.

npm registry

A naive approach is to scrape dependency information by extracting all the `package.json` files from downloaded modules in the npm registry. This approach would require extensive use of system resources and is not favorable. A closer look at the architecture of the npm registry[4] reveals that there is CouchDB database, named `skimdb` containing up-to-date meta-information for all available modules in the registry. This meta-information is essentially a copy of the `package.json` for all versions of a particular module.

Additional sources for data analysis

Information such as downloads per modules or number of Github Stars is gathered by using the official GitHub[5] and npm[6] RESTful APIs.

3.4.2 Data Model

A conventional RDBMS solution to access and store data provides the necessary means to conduct the required data activities. However, the data activities described in the approach primarily stress on viewing and treating dependencies as first-class citizens. A dependency graph $G = (V, E)$, where V is a set of npm modules and E a set of dependency relationships provides a closer domain-meaningful structure and reasoning of the data [1]. Therefore, mapping the `skimdb` as a graph database where the dependency relationship is viewed as a first-class citizen is more favorable over a non-graph database where the module is primarily seen as a first-class citizen.

[1] `https://cve.mitre.org/`

[2] `https://web.nvd.nist.gov/view/vuln/search-results?query=Node.js&search_type=all&cves=on`

[3] `https://nodesecurity.io/`

[4] `http://blog.npmjs.org/post/75707294465/new-npm-registry-architecture`

[5] `https://developer.github.com/v3/`

[6] `https://github.com/npm/download-counts`

We mirrored the `skimdb` as a neo4j graph database. A lacking feature in a neo4j database is native support for revision control of the graph. This is required to obtain historical version data of the modules. In order to overcome this obstacle, the `skimdb` is mapped as a time-based version graph where the domain-meaningful structure of the graph is preserved and states are introduced to represent the evolution of a module from a time and version perspective [8]. This data model is presented in Figure 3.1 showing two modules that depend on each other. The graph properties and relationships are introduced as follows:

Node (vertex)

- `Package`: The identity node contains immutable properties such as the creation timestamp and name identifier of a module.
- `PackageState`: The state node is the version release of a module and includes information such as the version number and unique combined version and name identifier.

Relationship (edge):

- `SNAPSHOT`: A `PackageState` is associated with a `Package`. `Package` is the domain-meaningful structure while the `PackageState` represents a particular version in a time context. The time context is defined in the relationship with a "from"-field (*release timestamp of the version*) and the "to"-field (*timestamp when the version was replaced with a newer version*). In order to specify the latest version in the "to"-field, the largest possible integer number in JavaScript is used.
- `DEPENDS_ON`: The dependency information from version to version evolves. Thus, the `PackageState` maps the dependencies it uses to `Package` node. This relationship includes the version range of the dependency. A drawback is that the version range cannot be used directly in the query interface; creating a relationship mapping for all valid version would drastically reduce the performance considering the large number of relationships needed to create.

The data collected for the quantitative analysis is stored in a MongoDB database. There are no particular advantages to using MongoDB concerning the problem domain. However, MongoDB provides a native database driver for Node.js to avoid performance bottlenecks in using HTTP for database transaction [2]. The SQL-based databases we investigated used a RESTful API to perform database transactions, which could pose performance constraints with processing a large dataset such as npm.

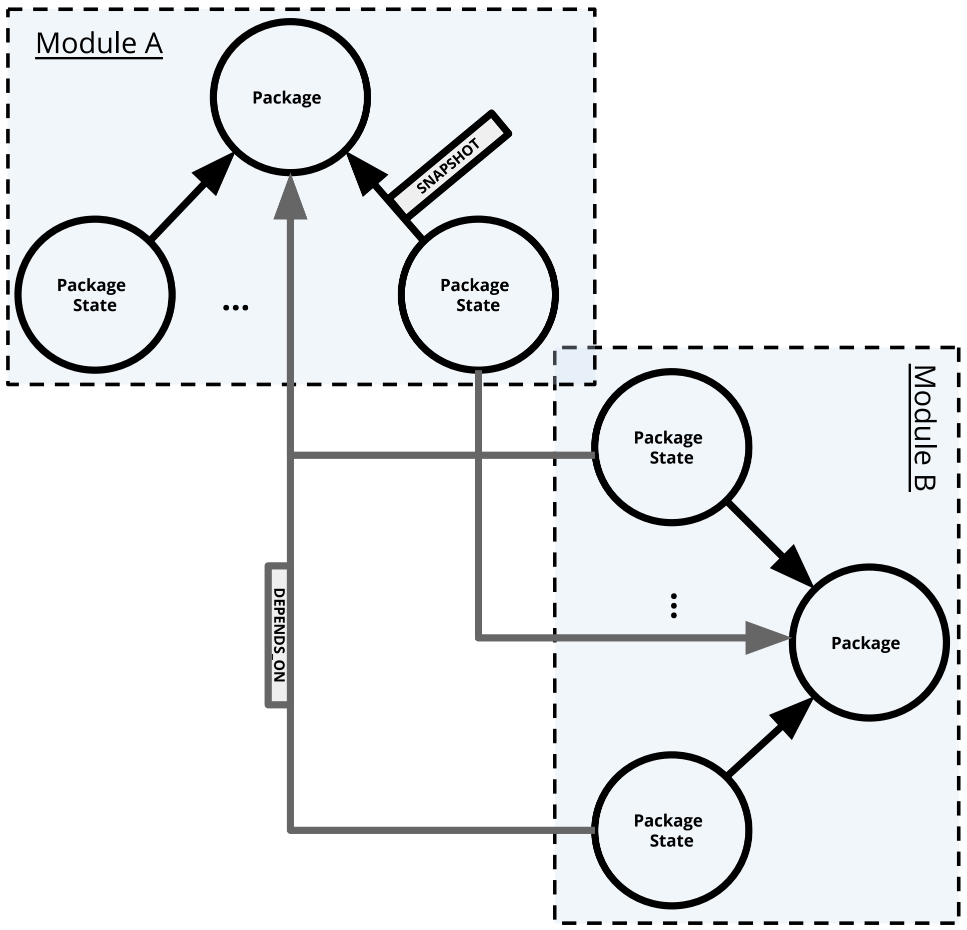


Figure 3.1: Data Model for neo4j database

Chapter 4

Implementation

This chapter describes the implementation details of the tool `Rastogi.js`. The first section uncovers the mechanism for recognizing vulnerable dependencies. The following sections presents the intuition and solution for gathering the necessary data for answering each research question. Finally, the architecture of `Rastogi.js` and the processing infrastructure is shown.

4.1 Recognizing vulnerable dependencies

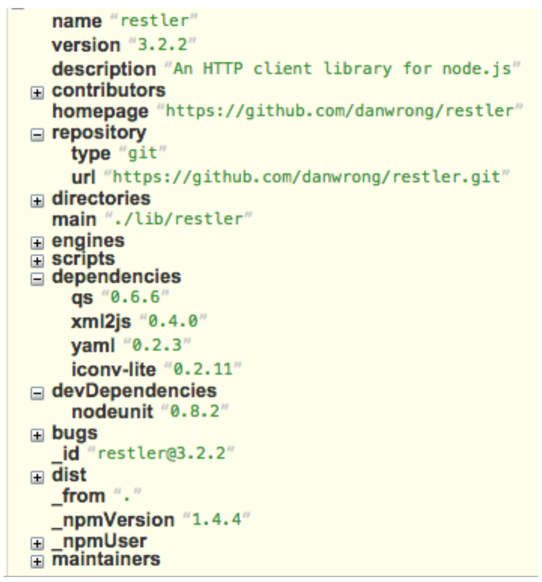
A vital component for detecting vulnerabilities is a mechanism that gathers the necessary and relevant information from vulnerability databases to recognize potentially vulnerable dependencies. The following subsections pinpoint on relevant information that enable this.

4.1.1 Structure of a `package.json` file

The minimally required file in a project for publishing to the npm registry is the `package.json` file. The JSON file act as a deceleration form for the supplied content in a module; a version, and a name identifier is minimally required. An example of this file is shown in Figure 4.1 for a module named *restler*. The official documentation provides full coverage of potential fields that can be declared in the file[1]. A selection of fields that are relevant for the detection mechanism with a brief explanation of the different dependency types is described below:

- `name`: npm module identifier
- `version`: semver version
- `repository`: details about the repository, such as projects hosted on Github
- `dependencies`: dependencies that are used in the source code
- `devDependencies`: dependencies intended in the development workflow

[1]`https://docs.npmjs.com/files/package.json`



```
   name "restler"
   version "3.2.2"
   description "An HTTP client library for node.js"
⊞ contributors
   homepage "https://github.com/danwrong/restler"
⊟ repository
     type "git"
     url "https://github.com/danwrong/restler.git"
⊞ directories
   main "./lib/restler"
⊞ engines
⊞ scripts
⊟ dependencies
     qs "0.6.6"
     xml2js "0.4.0"
     yaml "0.2.3"
     iconv-lite "0.2.11"
⊟ devDependencies
     nodeunit "0.8.2"
⊞ bugs
   _id "restler@3.2.2"
⊞ dist
   _from "."
   _npmVersion "1.4.4"
⊞ _npmUser
⊞ maintainers
```

Figure 4.1: Example of a package.json entry (in skimdb)

- `bundledDependencies`: dependencies that are not downloaded from npm, but included(bundled) with the module
- `peerDependencies`: dependency type designed for version compatibility of plugins used in the library.
- `optionalDependencies`: in the scenario of a failure for a dependency, an alternative dependency is used

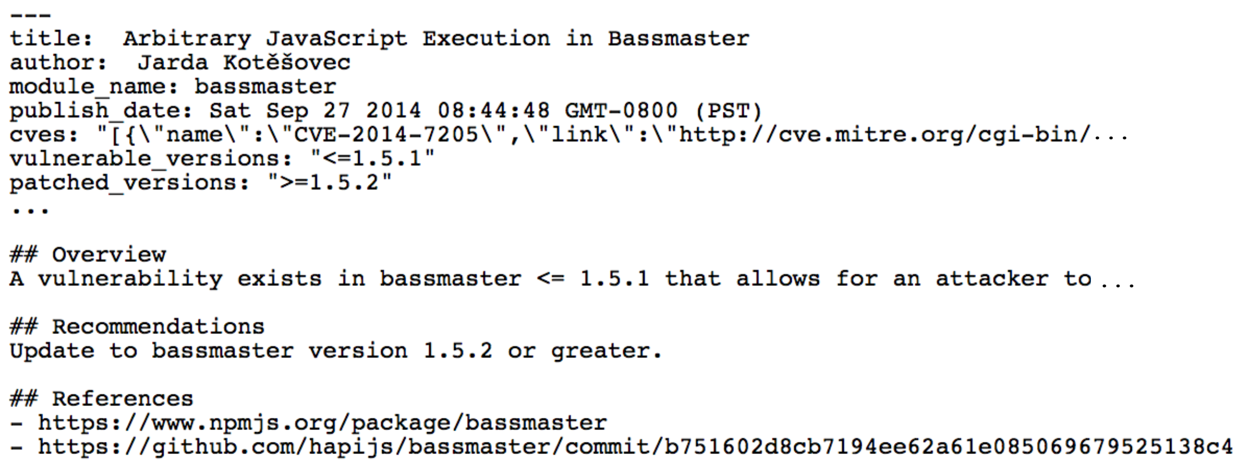
For all the dependency types, a semver version range is preferably specified. There are cases where a URL is specified instead of a version range.

4.1.2 Structure of an NSP advisory

The NSP is the data source for security advisories in this study. The advisories are retrievable in markdown format from the official github repository[2]. Figure 4.2 presents an NSP advisory for `bassmaster`. The structure of the advisories follow a common pattern and below is the derived anatomy of the advisories:

- **header information:**
  - `title`: suggesting title of the security flaw
  - `author`: the person reporting the vulnerability
  - `module_name`: npm module identifier

[2]`https://github.com/nodesecurity/nodesecurity-www/tree/master/advisories`



```
---
title:  Arbitrary JavaScript Execution in Bassmaster
author:  Jarda Kotěšovec
module_name: bassmaster
publish_date: Sat Sep 27 2014 08:44:48 GMT-0800 (PST)
cves: "[{\"name\":\"CVE-2014-7205\",\"link\":\"http://cve.mitre.org/cgi-bin/...
vulnerable_versions: "<=1.5.1"
patched_versions: ">=1.5.2"
...

## Overview
A vulnerability exists in bassmaster <= 1.5.1 that allows for an attacker to ...

## Recommendations
Update to bassmaster version 1.5.2 or greater.

## References
- https://www.npmjs.org/package/bassmaster
- https://github.com/hapijs/bassmaster/commit/b751602d8cb7194ee62a61e085069679525138c4
```

Figure 4.2: Example of an NSP advisory

  - `cves`: if available, the CVE for the vulnerability
  - `publish_date`: publication timestamp (compatible with JavaScript Date constructor)
  - `vulnerable_versions`: version range with semantic versioning
  - `patched_versions`: version range with semantic versioning

- **body:** description and reference to the vulnerability

The `module_name`, `publish_date`, `vulnerable_versions` and `patched_versions` are identified fields that are crucial in the detection phase of vulnerable dependencies.

4.1.3 Challenges for a matching formula

The `module_name` and `vulnerable_versions` are the key ingredients for recognizing vulnerable modules in the npm registry. A potential challenge in the matching phase is the degree of *accuracy* and *confidence* of the results. Poor accuracy in the context of security has implications in the reliability of the results; a high-rate of *false negatives* can leave security flaws unnoticed while a high-rate of *false positives* yields non-present security flaws in the reporting.

Module name: The module name for a npm module is unique and unambiguous. Therefore, the `module_name` field will precisely match the declared module in the dependency section of the `package.json`. The npm dependency resolver operates in a similar fashion to retrieve and download the dependencies defined in the `package.json` file. It could be argued that, for example, a dependency named “rastogi” (core package) and “rastogi.web” (plugin) should be matched (if the plug-in leverage code from the core). However, the npm dependency resolver will consider it as two separate modules and not the same; thus the same principle is applied in the matching process. On the other hand, if “rastogi” is defined as a dependency in the “rastogi.web” a vulnerability would be identified using this principle.

Version range: Applying the interval version range from `vulnerable_versions` on dependencies with a single version (such as in Figure 4.2) is trivial. However, a large number of dependencies are declared with an advanced version range syntax. This poses a problem to identify whether a potential version range for a dependency is vulnerable or not. In order to mitigate this, the dependency version range is resolved to the latest version and the version is tested against the `vulnerable_versions` range to see whether it is vulnerable or not.

4.2 Detecting vulnerable modules

The previous section touched on the necessary components and challenges to recognize vulnerable dependencies in npm modules. This is required for answering **RQ1**. The method for leveraging this information and scanning dependencies for vulnerabilities is shown in Algorithm 1. The interplay of the procedures is described below:

StartScan
The procedure parses the provided advisory files by extracting the `vulnerable_versions` and `module_name` fields (*line 27*). For each advisory, the extracted information is submitted to the `VulnerabilityScan` procedure for analysis and the results are stored in a database (*line 30*).

VulnerabilityScan
The `VulnerabilityScan` procedure searches for vulnerable dependencies based on the provided advisory information. The procedure returns the result of the analyzed modules.

- `getModulesUsingDep`: The function returns all modules of the latest version that use `module_name` as dependency.
- `getAllVersions` and `getAllVulnVersions`: The supplied `module_name` (*adv.modName*) is used to retrieve the released versions and the `vulnerable_versions`(*adv.verRange*) is applied on the released versions to get all vulnerable versions.
- For each *flagged* module, the dependency range is tested on the vulnerable version list (*line 13*). All matching versions are saved in *C*. If there are any matching versions, we define *vulnRange* to be a boolean that allow at least a vulnerable version and is set to true. In order to determine if the version range will resolve to a vulnerable version, this is done in *line 16*. If it resolves to a vulnerable version, *vuln* is set to true and is viewed as a vulnerable dependency.

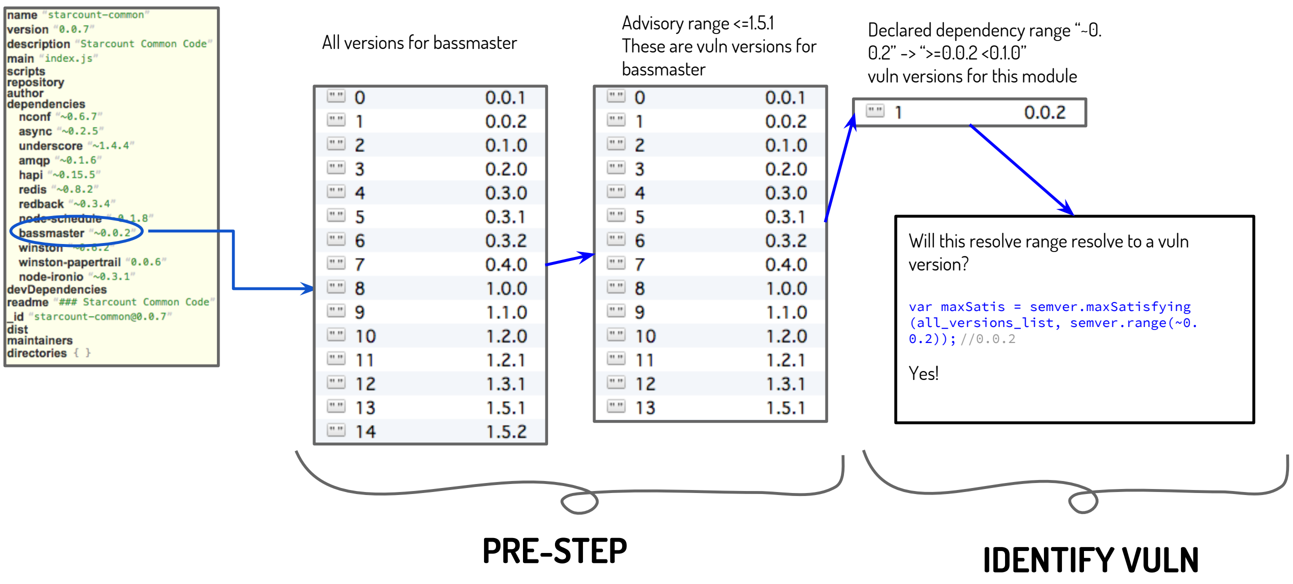
ResolveVuln
Based on the released (*allVer*) versions, distinct vulnerable versions (*vulnVer*)

from `module_name` and the version range from the *flagged* module are needed for mimicking the dependency resolver in order to get the particular version. All the released versions are required to identify what is the *highest* resolvable version based on the supplied version range. This is done in `highestResolvableVer`. The obtained version is tested to see if it is a vulnerable version (*line 3*).

Algorithm 1 Algorithm to scan dependencies for known vulnerabilities

```
 1: procedure ResolveVuln(Set allVer, Set vulnVer, versionRange)
 2:     $ver \leftarrow$ highestResolvableVer(allVer, versionRange)
 3:     return containsVersion(ver,vulnVer)                          ▷ true or false
 4: end procedure
 5:
 6: procedure VulnerabilityScan(adv)
 7:     Set $M \leftarrow$ getModulesUsingDep(adv.modName)
 8:     if $M \neq \emptyset$ then
 9:         Set $A \leftarrow$ getAllVersions(adv.modName)
10:         Set $V \leftarrow$ getAllVulnVersions(A, adv.verRange)
11:         Set $result \leftarrow \emptyset$
12:         for $mod \in M$ do
13:             Set $C \in \{v \in V|$ isValid(v, mod.verRange) $\}$
14:             if $C \neq \emptyset$ then
15:                 $vulnRange \leftarrow true$
16:                 $vuln \leftarrow$ ResolveVuln(A, V, mod.verRange)
17:             else
18:                 $vulnRange, vuln \leftarrow false$
19:             end if
20:             $result \leftarrow result \cup \{(mod, vulnRange, vuln)\}$
21:         end for
22:         return result
23:     end if
24: end procedure
25:
26: procedure StartScan(Set advFiles)
27:     $advisories \leftarrow$ parse(advFiles)
28:     for $adv \in advisories$ do
29:         Set $res \leftarrow$VulnerabilityScan(adv)
30:         storeResult(res)
31:     end for
32: end procedure
```

An example using the *bassmaster* advisory is shown in Figure 4.3 to demonstrate the execution of the algorithm.


```
name "starcount-common"
version "0.0.7"
description "Starcount Common Code"
main "index.js"
scripts
repository
author
dependencies
   nconf "~0.6.7"
   async "~0.2.5"
   underscore "~1.4.4"
   amqp "~0.1.6"
   hapi "~0.15.5"
   redis "~0.8.2"
   redback "~0.3.4"
   node-schedule "~0.1.8"
   bassmaster "~0.0.2"
   winston "~0.6.2"
   winston-papertrail "0.0.6"
   node-ironio "~0.3.1"
devDependencies
readme "### Starcount Common Code"
_id "starcount-common@0.0.7"
dist
maintainers
directories { }
```

All versions for bassmaster

0	0.0.1
1	0.0.2
2	0.1.0
3	0.2.0
4	0.3.0
5	0.3.1
6	0.3.2
7	0.4.0
8	1.0.0
9	1.1.0
10	1.2.0
11	1.2.1
12	1.3.1
13	1.5.1
14	1.5.2

Advisory range <=1.5.1
These are vuln versions for bassmaster

0	0.0.1
1	0.0.2
2	0.1.0
3	0.2.0
4	0.3.0
5	0.3.1
6	0.3.2
7	0.4.0
8	1.0.0
9	1.1.0
10	1.2.0
11	1.2.1
12	1.3.1
13	1.5.1

Declared dependency range "~0.0.2" -> ">=0.0.2 <0.1.0"
vuln versions for this module

1	0.0.2

Will this resolve range resolve to a vuln version?

```
var maxSatis = semver.maxSatisfying
(all_versions_list, semver.range(~0.
0.2)); //0.0.2
```

Yes!

Figure 4.3: Example of how the detection works

4.3 Exploring cascading dependencies

A method that captures the propagation of a vulnerable dependency is required for answering **RQ2**. The procedure `VulnerabilityScan` in Algorithm 1 is utilized to recursively identify further dependencies that are vulnerable. The starting condition is the advisories, and the results of analyzing the advisories are piped back as a matching criteria to traverse deeper in the dependency chain until no further vulnerable dependencies are found. The algorithm for the cascading effect is presented in Algorithm 2.

The `Cascader` procedure takes a list of vulnerable modules and the *depth* of the traversal in the chain. The starting condition is a set of advisories (*line 3*). A global variable *marked* keeps track of modules that have already been explored in order to avoid a circulation scenario. The *nextLevel* variable contains modules that are vulnerable based on the results from the vulnerability scanner (*line 8 and 9*). Those modules are used as a search criteria to explore dependencies deeper in the chain (*line 18*). If no more vulnerable dependencies are discovered, the procedure returns and ends the recursion (*line 15*). Finally, for each explored module, the results are stored together with the *depth* in a database (*line 11*).

4.4 Time latency from the release of an advisory to a patch fix

For **RQ3**, the target is to calculate the time for an individual module to be vulnerable-free from a dependency management perspective. This requires a historical version analysis of the module where the dependency changes are tracked. Algorithm 3 presents an overview of the main components for the historical version analysis. The idea is that versions of a

Algorithm 2 Algorithm to measure the cascading effect

```
1: $marked \leftarrow \emptyset$
2: $depth \leftarrow 1$
3: $modules \leftarrow$ a set of advisories
4: procedure CASCADER(modules,depth)
5:     $nextLevel \leftarrow \emptyset$
6:     for $mod \in modules$ do
7:         if $mod \notin marked$ then
8:             Set $result \leftarrow$ VulnerabilityScan(mod)
9:             $nextLevel \leftarrow nextLevel \cup \{r \in result|$ isVulnModule(r) $\}$
10:            $marked \leftarrow marked \cup \{mod\}$
11:            storeResult(result,mod,depth)
12:        end if
13:    end for
14:    if $nextLevel = \emptyset$ then
15:        return
16:    else
17:        $depth \leftarrow depth + 1$
18:        cascader(nextLevel, depth)
19:    end if
20: end procedure
```

module is compared with each other to identify changes made for each version. Thus, this involves comparing the declared version range for each version over time. The key components are further explained:

- `getModulesUsingDep`: similar to the function in Algorithm 1, instead of retrieving the latest version of a module, all modules at the publication timestamp of the advisory is retrieved.
- `isVulnerable`: A modification to the `VulnerabilityScan` procedure where the version range is directly compared to see whether it is vulnerable or not. In addition to this, if the dependency is removed, it is considered as vulnerable-free.
- `timeCompare`: the number of days between the publication date of the advisory and the released version
- `store`: The results returned from the `CompareVersions` is stored in a database. The individual result of the version and the comparison with a previous version for a module is saved.

Algorithm 3 Algorithm to determine the time latency

```
1: $advisories \leftarrow$ a set of advisories
2: procedure VERSIONANALYSIS(advisories)
3:     for $adv \in advisories$ do
4:         Set $M \leftarrow$ getModulesUsingDep(adv.modName, adv.pubDate)
5:         for $mod \in M$ do
6:             Set $V \leftarrow$ getAllVersionsAndSort(mod)
7:             $result \leftarrow$ compareVersions(V,adv)
8:             store(result)
9:         end for
10:     end for
11: end procedure
12:
13: procedure COMPAREVERSIONS(set V,adv)
14:     $prev \leftarrow previous\_version$
15:     $result \leftarrow \emptyset$
16:     for $ver \in V$ do
17:         if $not first release in V$ then
18:             $vulnCompare \leftarrow$ compare(isVulnerable(prev,adv),isVulnerable(ver,adv))
19:             $result \leftarrow result \cup \{(vulnCompare, typeCompare)\}$
20:         end if
21:         $timeDiff \leftarrow$ timeCompare(adv.pubDate,ver.releaseDate)
22:         $result \leftarrow result \cup \{timeDiff\}$
23:         $prev \leftarrow ver$
24:     end for
25:     return result
26: end procedure
```

4.5 Dependency Analysis Tool: Rastogi.js

All the concepts presented in this chapter are implemented in a tool called Rastogi.js, an asynchronous dependency analysis tool. The tool is built for running on Node.js and available for download at `https://github.com/jhejderup/rastogi.js`. A process description of rastogi.js is depicted in Figure 4.4 where the vulnerability and cascading analysis block focus on the detection of vulnerabilities, while the *Evolution Analyzer* block deals with comparing historical version data and identifying changes over time.

Rastogi.js is developed with a pluggable *analyzers* feature for performing various analyses of npm dependency data. Currently, the following analyses are supported:

- **vulnAnalyzer**: inspects modules to find vulnerable dependencies (to answer *RQ1*)
- **cascadeAnalyzer**: analysis the cascading effect of known vulnerable modules (to answer *RQ2*)
- **evoAnalyzer:** for analyzing the version-evolution of modules (to answer *RQ3*)

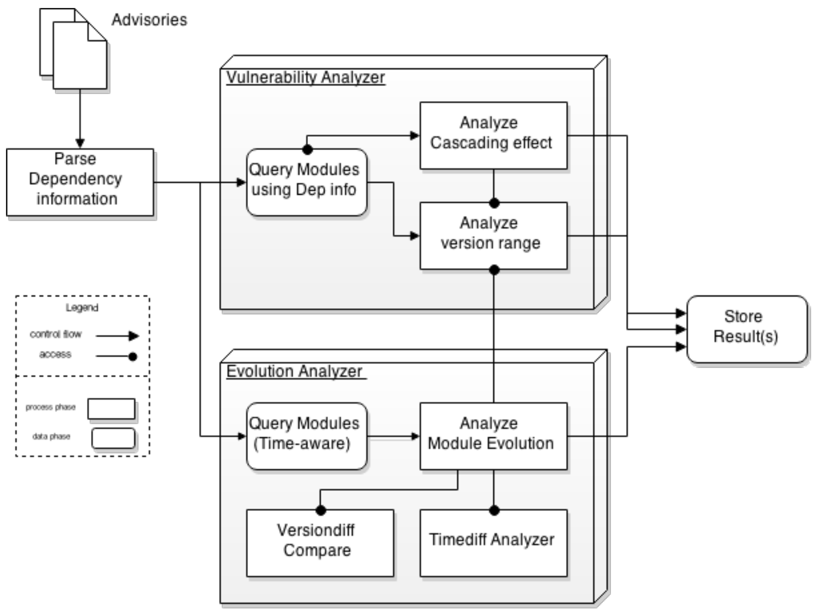


Figure 4.4: Process view of Rastogi.js

The prerequisites for using Rastogi.js are a mongodb database and a neo4j database to retrieve and store data. There are certain third-party modules that play an important role in enabling rastogi.js:

- **semver:**[3] The library plays a central role in validating and resolving version ranges.
- **commander**[4]: provides a CLI for interfacing with the core features in rastogi.js

4.5.1 Development workflow

The tool is hosted on GitHub and uses *gulp.js*[5] as a build system and task-runner. The project currently enforces a consistent code style and unit testing of the source code. The following gulp.js plug-ins are used in the build pipeline:

- **JSCS**[6]: code style linter using the airbnb[7] javascript style guide
- **JSHint**[8]: JavaScript code quality tool for detecting errors and potential problems
- **Mocha**[9]: A JavaScript test runner to execute the unit tests

[3] `https://github.com/npm/node-semver`
[4] `https://www.npmjs.com/package/commander`
[5] *`gulpjs.com/`*
[6] `jscs.info/`
[7] `https://github.com/airbnb/javascript`
[8] `jshint.com/`
[9] `mochajs.org/`

4.6 Infrastructure for processing the npm registry

In order to create the graph database needed for the experiments, close to 100,000 modules from the npm registry needs to be processed in a manageable time frame. The procedure for processing and storing the data in neo4j is the same for each module. Therefore, the goal is to achieve a high level of task parallelism. A distributed system using the Work-Stealing pattern [10] was developed and is shown in Figure 4.5. The system uses ZeroMQ[10] for communication between the nodes and fault tolerance is supported out-of-the-box. The master node retrieves all modules from the `skimdb` database and puts it into a work queue. The workers retrieve the tasks from the master, process it, and send it to the collector node for storing it in the graph database. A mongodb database is also used for creating a snapshot of the `skimdb` for reproducibility.

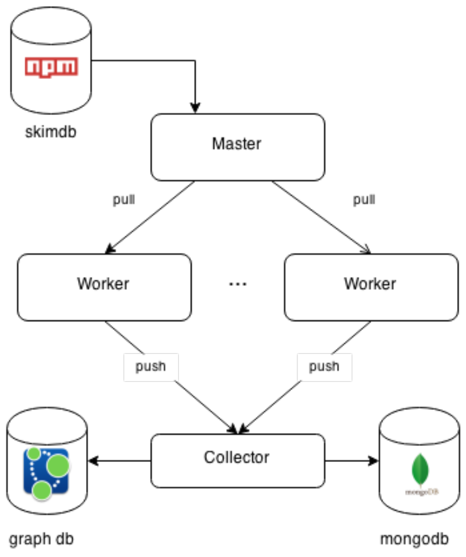


Figure 4.5: Overview of the distributed process for handling skimdb entries

[10]`zeromq.org/`

Chapter 5

Descriptive Statistics

5.1 Dataset

We used a snapshot of the npm registry dated 12 October 2014 where the total number of database entries in `skimdb` were $104,675$. Out of these entries, $99,679$ have time and version fields. Modules that have missing version or time information from the snapshot were not included. Furthermore, data cleaning was performed to remove modules that deviated from the intent of the study. An example is `bigben`[1], a project that released a version every 15 minutes, published over 690 versions before it was stopped. Another example is the `mikolalysenko-hoarders`[2] module with over 260 dependencies. The snapshot serves as a reference to reproducing the graph database.

A descriptive overview of the graph database is presented in Table 5.1. The graph has in total $99,631$ modules with $406,555$ dependency relationships. Dependencies intended for use in the code base constitute $201,892$[3] out of $406,555$; the remainder are other dependency categories. There are in total $16,662$ nodes that do not use any dependency and serve as a dependency for other modules.

The snapshot and the graph database are available for download on GitHub[4].

5.2 Properties of the Dependency graph

Before delving into analyzing vulnerabilities in dependencies, we need to understand the general dependency usage in the npm registry.

5.2.1 Overview of the dependency types

Section 4.1 in Chapter 4 introduced five categories to declare dependencies. An overview of the usage for the dependency categories is shown in Table 5.2. Production (*dep*) and development (*dev*) dependencies are dominantly the most declared form of dependencies while

[1]`https://skimdb.npmjs.com/_utils/document.html?registry/bigben`

[2]`https://skimdb.npmjs.com/_utils/document.html?registry/mikolalysenko-hoarders`

[3]See Table 5.2

[4]`https://github.com/jhejderup/rastogi.js/tree/master/dataset`

`Package` nodes	99,631
`PackageState` nodes	544,663
graph diameter (dep)	11
Dependencies(all ver)	2,797,600
Dependencies(latest ver)	406,555
Isolated Nodes	16,662

Table 5.1: Statistics of the graph database

Dep type	All	Dep	Dev	Peer	Optional
avg	4.08	2.02	1.98	0.065	0.01
max	83	77	59	46	16
use >=1 dep	77,829	61,508	51,044	5,364	590
tot Dep. rel.	406,555	201,892	197,009	6,559	1,095

Table 5.2: Statistics of the dependency types

the usage of peer and optional dependencies is substantial. Furthermore, 78% $(77,829/99,631)$ of the modules include at least one dependency (row 3), 67% $((61508+5364)/99,631)$ are declared for production code. This clearly indicates that dependencies is a standard practice while on average (the mean value) two dependencies are declared as either *dev* or *dep* in a module.

5.2.2 In and out-degree of modules

In order to gain an understanding of the *usage* and how *dependent* modules are in the graph, a distribution of the in-degree and out-degree of the modules is shown in Figure 5.1. The out-degree serves as the *usage* of external modules while the in-degree is the *dependence* of the module (external modules that *depend* on the module).

We can identify that both dependency types have a similar shape of the exponential decay curve with a skewed data distribution. It can be clearly seen that for both the in-degree and out-degree, a strong majority of the modules use one dependency and have one *dependent*. Furthermore, it can be observed that the out-degree curves have a more steady decrease compared to the in-degree curves where it drastically decrease to degree two before it decays. This suggests that modules are less likely to have multiple dependent modules while there is a steady *usage* of more than one dependency for a module. A closer inspection of the out-degree in Figure 5.2, reveals that 91% of the modules uses five or fewer dependencies.

5.2.3 Path distance distribution for the npm graph

The path distance captures the indirect *dependency-of-dependency* relations in the graph which is of relevance for the *cascading effect*. The *graph diameter* in Table 5.1 shows that the longest possible path is 11 between two modules in the graph. The distribution of the

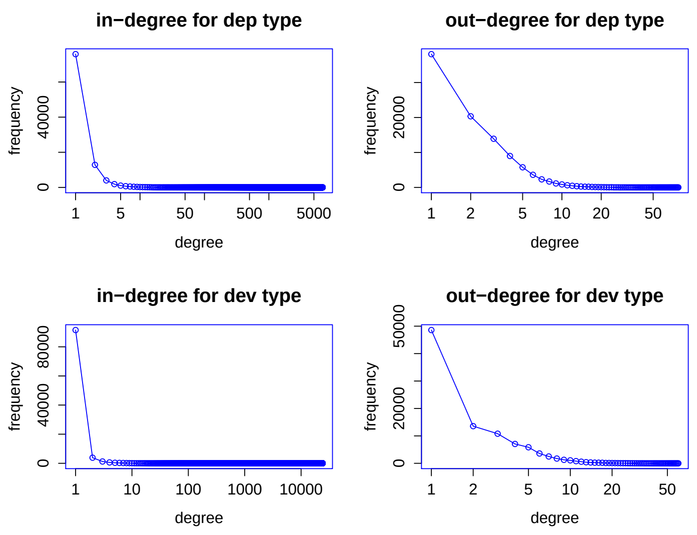


Figure 5.1: In-degree and out-degree distribution for the npm graph

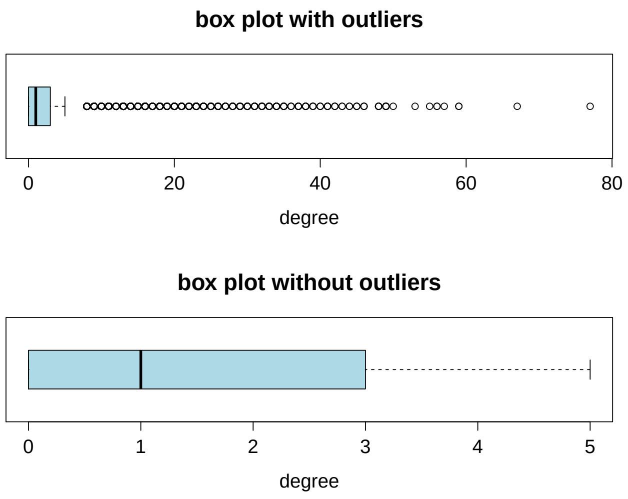


Figure 5.2: Percentile (9th to 91th) Box-plot of the out-degree distribution for *dep* dependencies

shortest path lengths is presented in Figure 5.3. Overall, $1,210,833$ combinations of the possible shortest path between modules were discovered. The average path length for the *dep* dependency type is 2.67 for the registry, the most frequent paths are between two to three hops of length. This suggests overall that the dependency chain is not long. However, looking at the frequency, more than 17,900 paths were discovered for six hops, over 3,000 paths for eight hops and 115 paths for ten hops indicating that there is a significant number for long dependency chains.

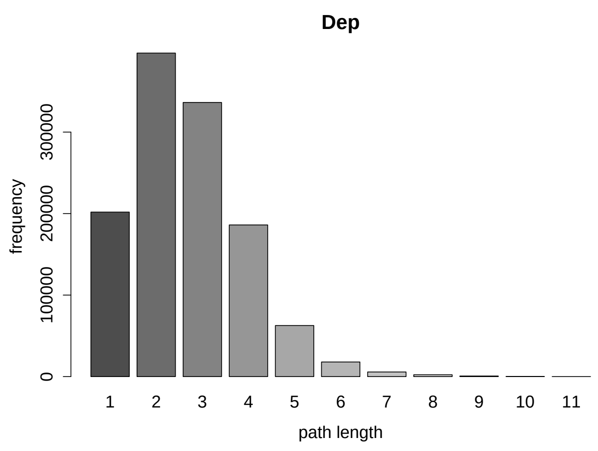


Figure 5.3: Distribution of the dependency chain

5.3 Advisories

A total of 22 public advisories published between July 2013 to October 2014 were used in the study. Table 5.3 presents an overview of the advisories. The dominating type of vulnerabilities is injection attacks and DoS vulnerabilities. Overall, there are 19 unique modules with known vulnerabilities reported. Three of the modules have been advised twice with a different vulnerable version range. An observation is also made on *hapi-v2* and *qs* where the suggested update is not a patch but a minor version update. This implies that the changes made should be backward-compatible.

5.3.1 Usage overview of the advised modules

Roughly 3.2%[5] of the npm modules use the advisory modules as dependencies. The usage of the advisory modules as dependencies is depicted in Figure 5.4 and Table 5.4. *printer*, *tomato* and *codem-transcode* are not used as dependencies and therefore omitted. The most used modules (*dep*) are *connect*, *marked*, *qs* and *validator* with more than 300 dependent modules. 12 of 16 advised modules comprise of more than 80% of *dep* dependencies while *hapi*, a highly used module, has the least fraction of *dep* dependencies. Furthermore, *hapi* and *yar* has the highest percentage of *peer* dependency use. There is a strong percentage of *dev* dependency use for *hapi*, *connect* and *st*. *optional* dependencies had almost non-existent use. Overall, the overview of the modules suggests that there is a strong case that the advised modules are used as dependencies in npm modules.

5.3.2 Download statistics of the advised modules

The download counts for each npm module provides an insight to the popularity of the module and complement the usage statistics in the previous subsection. Between the period

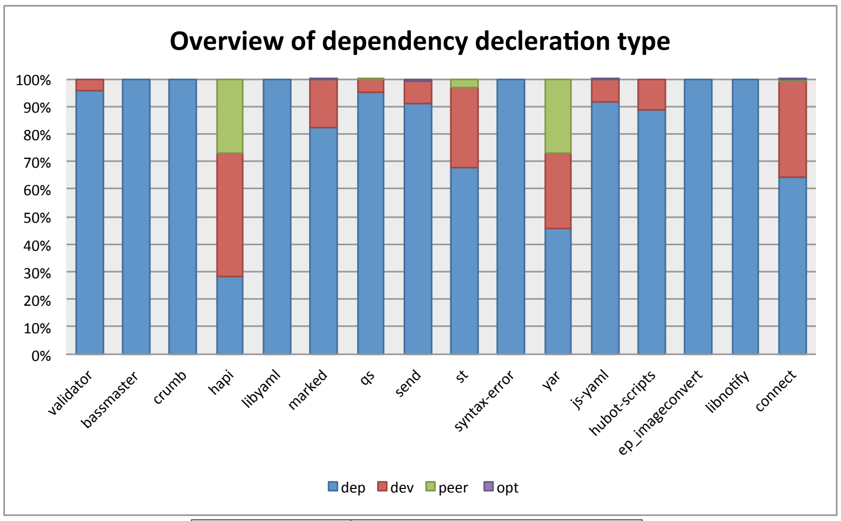
[5] from Table 5.2 where modules that use the *dep* type (3195/61508)

Advisory	**Desc**	**Published**	**Module**	**Vuln Range**	**Patched Range**
ADV1	XSS Filter Bypass	Oct 27 2014	validator (v2)	<2.0.0	>=2.0.0
ADV2	Arbitrary Code Execution	Sep 27 2014	bassmaster	<=1.5.1	>=1.5.2
ADV3	CORS Token Disclosure	Aug 1 2014	crumb	<3.0.0	>=3.0.0
ADV4	DoS Vulnerability	Feb 14 2014	hapi (v2)	2.0.x || 2.1.x	>= 2.2.x
ADV5	JSONP Vulnerability	Jul 08 2014	hapi (v6)	<6.1.0	>= 6.1.0
ADV6	heap-based buffer overflow	Feb 04 2014	libyaml	<0.2.3	>=0.2.3
ADV7	Content Injection	Jan 31 2014	marked	<=0.3.0	>=0.3.1
ADV8	Command Injection	Mar 06 2014	printer	<=0.0.1	>0.0.1
ADV9	DoS Event Loop Blocking	Aug 6 2014	qs	<1.0.0	>= 1.x
ADV10	DoS Memory Exhaustion	Aug 6 2014	qs	<1.0.0	>= 1.x
ADV11	Directory traversal	Sep 12 2014	send	<0.8.4	>= 0.8.4
ADV12	Directory traversal	Feb 06 2014	st	<0.2.5	>=0.2.5
ADV13	Script Injection	Jul 15 2014	syntax-error	<1.1.1	>= 1.1.1
ADV14	DoS Vulnerability	Jun 16 2014	yar	<2.2.0	>=2.2.0
ADV15	Arbitrary Code Execution	Jun 23 2013	js-yaml	<2.0.5	>= 2.0.5
ADV16	Command Injection	May 15 2013	hubot-scripts	<= 2.4.3	>2.4.3
ADV17	Authentication Weakness	Mar 07 2013	tomato	<= 0.0.5	>= 0.0.6
ADV18	Command Injection	Jul 07 2013	codem-transcode	<0.5.0	>=0.5.0
ADV19	Command Injection	May 06 2013	ep_imageconvert	<=0.0.2	>=0.0.3
ADV20	Command Injection	May 15 2013	libnotify	<= 1.0.3	>= 1.0.4
ADV21	XSS Filter Bypass	Jul 05 2013	validator (v1)	<1.1.0	>=1.1.0
ADV22	Reflected XSS Attack	Jul 01 2013	connect	<=2.8.0	>=2.8.1

Table 5.3: Overview of advisories

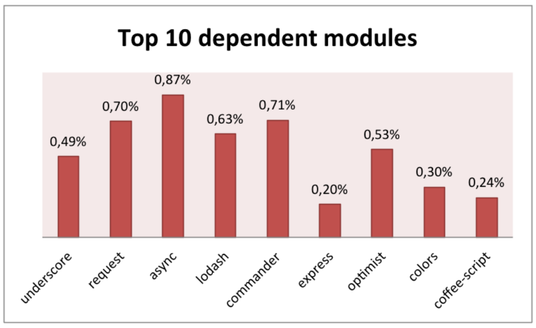
18 January to 18 February 2015, a total of $1,023,062,868$[6] downloads were made from the npm registry. In order to assess the total number of downloads from the time period as mentioned earlier, a reference of the top ten used modules is presented in Figure 5.5a. We can identify that module *async* (6200 dependents) has the highest download share of 0.87% followed by *commander* (3642 dependents) with 0.71%. Figure 5.5b shows the top seven downloads from the advisory modules. The top downloaded module is *qs* with 0.87% download share followed by *send* with 0.33%. Overall, the download counts of four modules from the advisories have a download share that is similar to the top used modules of the npm registry. The omitted advisory modules have a percentage that is less than *hapi*.

[6]`https://api.npmjs.org/downloads/point/last-month/`

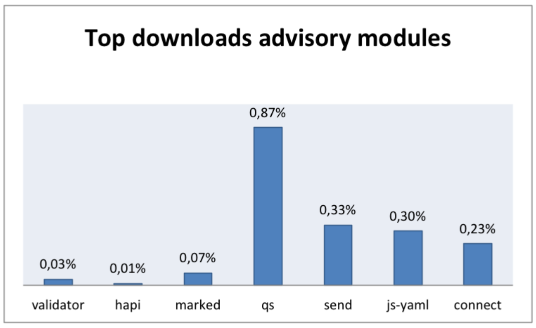


module	dep	dev	peer	opt	all
validator	304	14	0	0	318
bassmaster	1	0	0	0	1
crumb	5	0	0	0	5
hapi	87	139	84	0	310
libyaml	11	0	0	0	11
marked	750	155	1	2	908
qs	303	15	1	0	319
send	142	13	0	1	156
st	42	18	2	0	62
syntax-error	19	0	0	0	19
yar	5	3	3	0	11
js-yaml	573	52	1	1	627
hubot-scripts	8	1	0	0	9
ep_imageconvert	1	0	0	0	1
libnotify	4	0	0	0	4
connect	940	516	7	2	1465
Total	3195	926	99	6	4226

Figure 5.4 & Table 5.4: Overview of the usage for advisory modules in the npm registry



(a) Top 10 Dependent Modules



(b) Download share of advisory modules

Figure 5.5: Download Statistics of advisory modules

Chapter 6

Known vulnerabilities captured in dependencies of npm modules

This chapter presents the obtained quantitative results needed for answering the research questions presented in Chapter 1.

As mentioned in Section 4.1, there are five categories to declare dependencies depending on their purpose. The primary focus as defined in the research objective in Section 1.3 on dependencies that serve as a potential risk in an operational software system. Therefore, the dependency category *dev* which is intended for development purposes is omitted. The *dep* and *peer* dependencies are analyzed, and the remaining dependency categories have little use as shown in Section 5.3.

6.1 Overview

6.1.1 Volume of vulnerable modules for each advisory

Table 6.1 presents descriptive statistics for the number of modules depending on a vulnerable version of the advisory modules. The *cases* rows in the table include all identified cases and modules that match for more than one advisory. Previously, as mentioned in the third point of the `VulnerabilityScan` procedure in Section 4.1, two notions were introduced to view the version range of a dependency:

- *vulnerable range*: the version range includes one or more valid vulnerable versions, but the latest version does not resolve to a vulnerable version. This is considered a *warning*.
- *resolvable range*: the version range resolves to a vulnerable version and is considered as a direct *vulnerability*.

Overall, 1,678 unique modules have a *vulnerable range* for the *dep* category, and 77 modules have a *vulnerable range* for the *peer* category. Out of these, 1,029 (*dep*) and 18 (*peer*) modules resolve to a vulnerable version of the dependency. These numbers constitutes roughly 1% of the npm registry. In Table 5.4 in Section 5.3, the usage of advisory

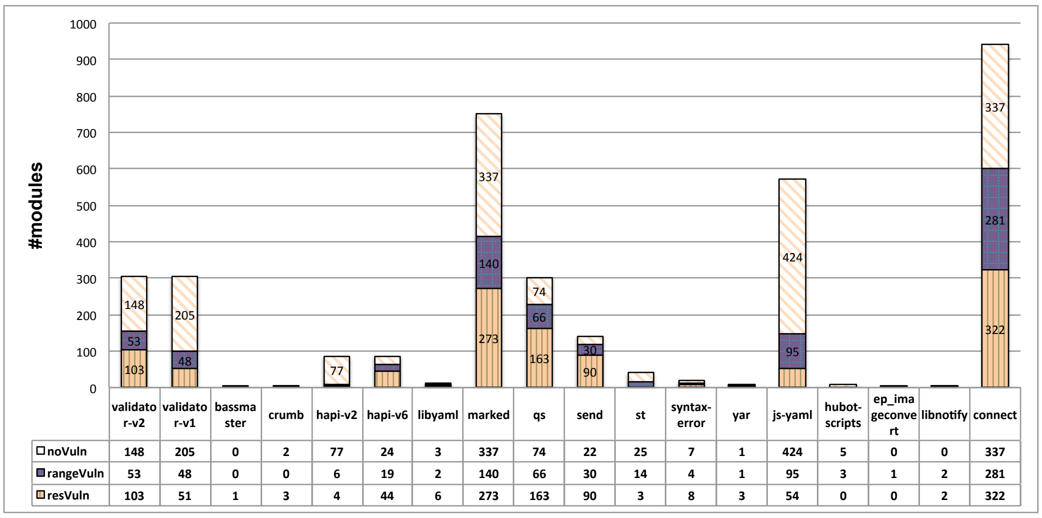
Range	**dep**	**peer**
cases	1895	114
unique	1678	77
Resolve		
cases	1130	19
unique	1029	18

Table 6.1: Overview of identified vulnerabilities

modules amounted to 3,195 for the *dep* category. We can establish that about 32% of the modules that use the advisory modules are vulnerable, and roughly 52% of the modules have vulnerable version range.

Figure 6.1 and 6.2 shows the distribution of vulnerable and non-vulnerable modules for each advisory in numbers and percentage. Roughly 44% of the advisories have 50% or more vulnerable modules while *hubot-scripts* and *ep_imageconvert* have no vulnerable modules. *hapi-v2*, *st*, and *js-yaml* have the lowest rate with 10% or less vulnerable modules. Top used advisories with more than 300 dependent modules[1] have a proportion of more or equal to 30% vulnerable modules in Figure 6.2. *js-yaml* stands out among the top used advisories for having more than 50% non-vulnerable modules while also being one of the most downloaded module.

The results suggest that the number of identified vulnerable modules is substantial, considering the relation between number of identified vulnerable modules with the total number of modules that use the advisory as dependency in the npm registry.



	validator-v2	validator-v1	bassmaster	crumb	hapi-v2	hapi-v6	libyaml	marked	qs	send	st	syntax-error	yar	js-yaml	hubot-scripts	ep_imageconvert	libnotify	connect
□ noVuln	148	205	0	2	77	24	3	337	74	22	25	7	1	424	5	0	0	337
■ rangeVuln	53	48	0	0	6	19	2	140	66	30	14	4	1	95	3	1	2	281
▥ resVuln	103	51	1	3	4	44	6	273	163	90	3	8	3	54	0	0	2	322

Figure 6.1: Distribution in numbers

[1]See Section 5.3.1 for top used modules

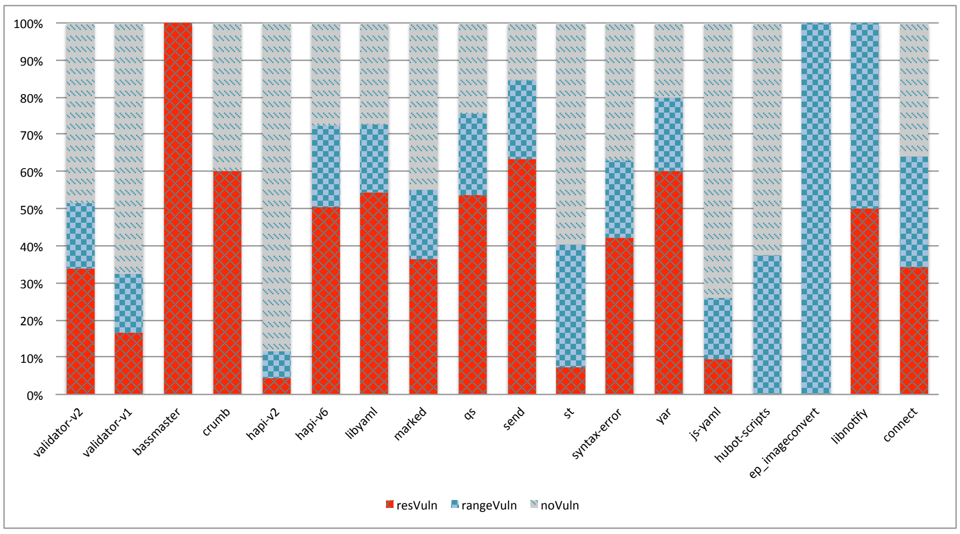


Figure 6.2: Distribution in percentage

6.1.2 Proportion of modules that are outdated

Advisories that are published in 2013 are *js-yaml, validator-v1 and connect*, and the remainder of the advisories are published in 2014. Figure 6.3 presents the distribution of the top dependent modules. We can identify that there is a significantly large number of vulnerable modules released in 2014 for advisories published more than one year ago. This can be seen for each of the three advisory modules published in 2013.

For six of the seven advisories in Figure 6.3, the modules have their latest known version released in 2011. *connect* has the highest number of 102 modules from 2011. Furthermore, a majority of the advisories has the highest number of outdated modules released in 2013 compared to previous release years.

Overall as shown in the pie chart for Figure 6.4, 26% of the vulnerable modules have not been updated since 2011-2012. Modules that are released in 2014 amount to 36%. The findings indicate that almost 2/3 of the vulnerable modules are outdated.

6.2 Indirect dependency propagation of vulnerable modules

6.2.1 Overview

Table 6.2 presents the depth and width of the *cascading* propagation in the dependency chain for the *dep* category. In the previous section, 1029 modules have been identified vulnerable. The analysis of the cascading dependencies obtained an addition of 461 cases with the longest traversed length reaching a depth of five. The *cascading* effect decreases dramatically *the further the* dependency chain is traversed. From depth two to three, the decrease is 79.25% of vulnerable modules.

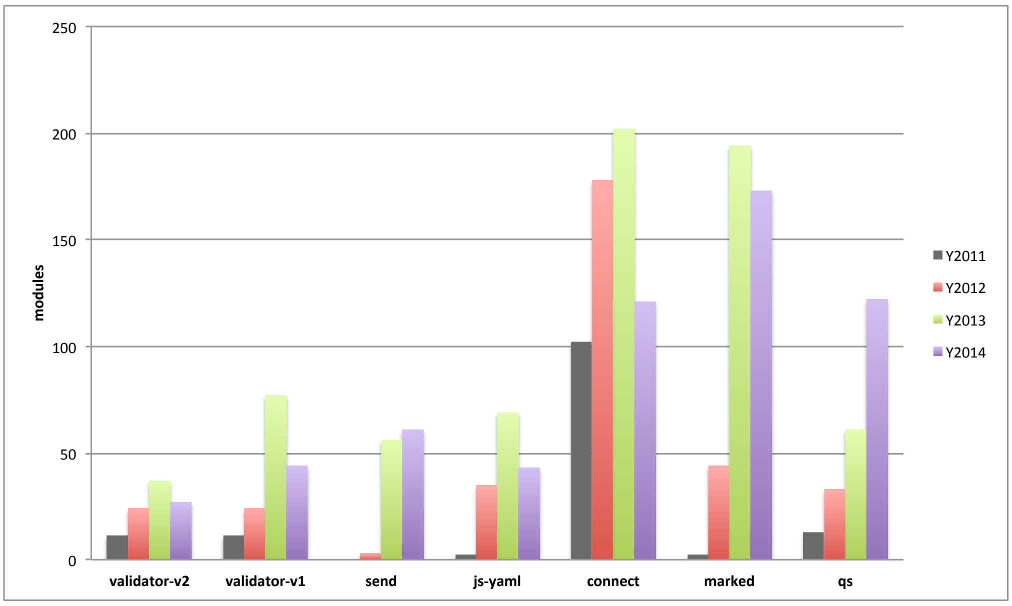


Figure 6.3: Year distribution for top dependent modules

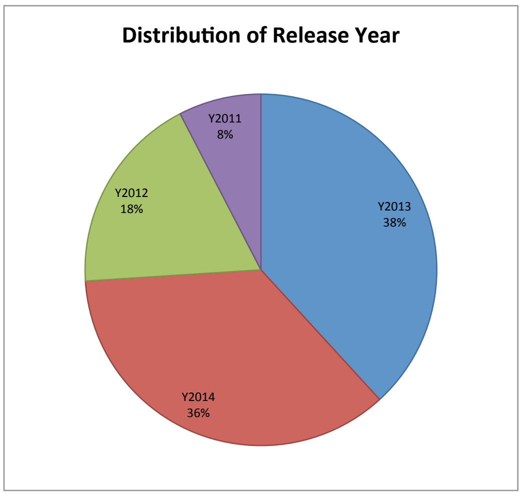


Figure 6.4: Overall distribution of the release year

A total of 1591 cases is identified for modules with direct and indirect vulnerable dependencies. The number corresponds to 1434 unique modules, an increase of 39.36% vulnerable modules from the direct 1029 vulnerable modules.

A dependency graph is constructed of the obtained results and shown in Figure 6.5 to understand the connectedness of the vulnerable modules. The graph consist of six connected components, *libnotify*, *syntax-error*, *st*, *libyaml* and *crumb* form their own independent network (bottom of the figure). The size of the node and the color scheme (yellow to red) is determined based on the out-degree where a large node size and the red color represents a

Depth	**Cases**
1(direct)	1130 (1029 unique)
2	371
3	77
4	11
5	2
Total	**1591 (1434 unique)**

Table 6.2: Overview of the *cascading effect* for the vulnerable modules

high out-degree.

The direct vulnerable modules, *connect*, *marked* and *qs* have the highest out-degree and edge connectedness. There is a single in-direct vulnerable module, *restler* that depends on *qs* that has the highest out-degree and edge connectedness with 108 dependent modules.

Finally, the dependency graph shows no nodes with substantial in-degree that are clearly visible as a result of the *cascading effect* except for the case of the *restler* module.

6.2.2 Cascading effect per advisory

Figure 6.6 presents the distribution of the depth and width of the cascading effect of each advisory. A total of 12 advisory modules have a cascading effect, suggesting that roughly half of the advisories have a cascading effect. The observed traversed length is the following: eight advisories traversed a depth of 3, five advisories traversed a depth of 4 and two advisories traversed a depth of 5.

qs has the highest share of indirect vulnerabilities with roughly 53%, which overtakes the number of direct vulnerabilities. This could partly be of the large number of dependent modules from the *restler* module in Figure 6.5. *validator-v2* is the only case where there is an increase in the number of identified modules from depth 2 to 3. The remainder of the modules has a decrease *the further the* dependency chain is traversed. For the majority of the advisories, the average increase of modules is 21% from depth 1 to 2, the average increase drops to roughly 7% from depth 2 to 3.

Although a single case had an increase of vulnerable dependencies in the traversal of the dependency chain, the majority of the cases suggests that the further the dependency chain is explored, the less likely more vulnerable dependencies are identified. Considering that most cases were identified at depth 2, this is in line with Section 5.2 where the most frequent shortest path length is two for the npm graph.

6.3 Time-to-update from vulnerable to vulnerable-free dependency

6.3.1 Overview

Table 6.3 shows an overview of the dependency changes in identified vulnerable modules from the publication date of the advisory to 12 October 2014[2]. This is the time window

[2]in this section #days is compared to this date

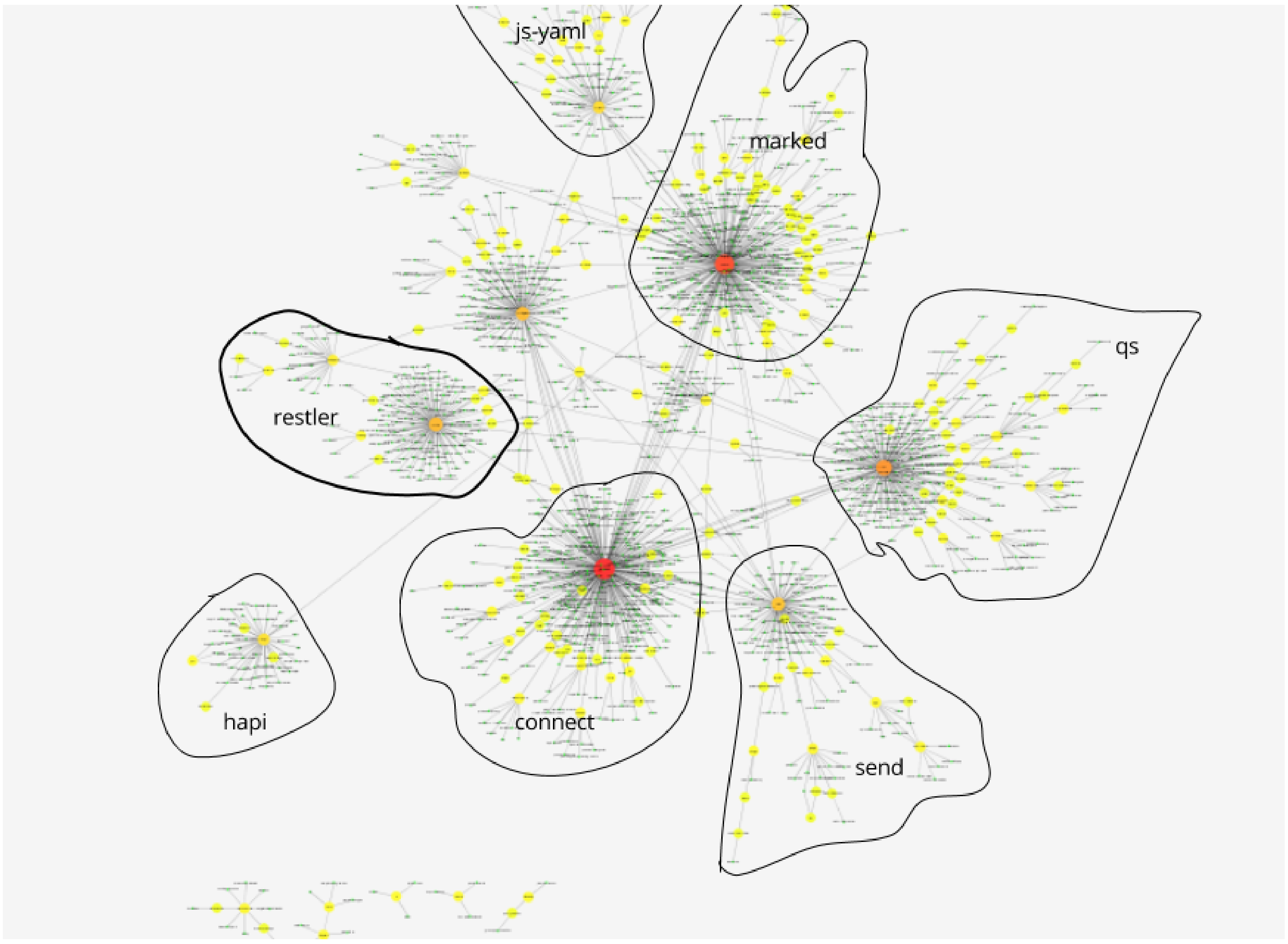


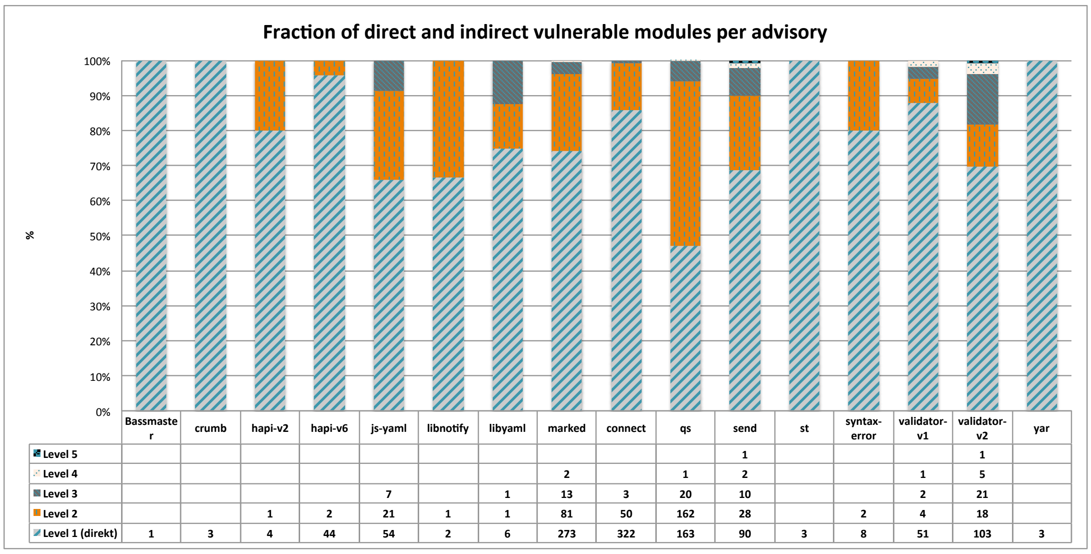
Figure 6.5: A *dependency-of-dependency* graph with direct and indirect vulnerable modules

used for identifying vulnerable modules and analyzing the version history. The left side of the delimiter in Table 6.3 presents the detected vulnerable modules on the publication date of the advisory, the right-side shows the changes that have been made in the vulnerable dependencies.

We can identify that for 11 of 15[3] advisories, the version range for those modules was updated to be completely vulnerable free (*depVulnFree*). Only 6 of 15 advisories had modules that removed the dependency (*depRemoved*). *connect* had the highest number of removals with 19 modules conducting this action.

qs had the largest reduction of modules with vulnerable dependencies with 20.32% less vulnerable modules since the publication day, followed by *js-yaml* with 17%, *marked* and *hapi-v6* with roughly 14%. *validator-v1* had the lowest rate with 8%. *bassmaster, libyaml, libnotify, yar and crumb* had only one or no changes at all.

[3]*validator-v2* was not included as it was released after the snapshot was taken



	Bassmaster	crumb	hapi-v2	hapi-v6	js-yaml	libnotify	libyaml	marked	connect	qs	send	st	syntax-error	validator-v1	validator-v2	yar
Level 5											1				1	
Level 4								2		1	2			1	5	
Level 3					7		1	13	3	20	10			2	21	
Level 2			1	2	21	1	1	81	50	162	28		2	4	18	
Level 1 (direkt)	1	3	4	44	54	2	6	273	322	163	90	3	8	51	103	3

Figure 6.6: Findings of vulnerable dependencies for each depth level

Considering cases with vulnerable version ranges, the highest observed reduction is modules with the *js-yaml* dependency with 18.5%, followed by *marked* with 16.5%. The lowest rate is modules using *qs* or *send* with a reduction of only one module. The *hapi-v6* had an increase with three modules; those are modules that resolved to a vulnerable version but is now updated to a vulnerable range instead.

Finally, we can establish that from an overall perspective the reduction of vulnerable dependencies from the publication is less than 20% and shows at large that vulnerable dependencies are not resolved as consequence of a published advisory.

Advisory	nonVuln	rangeVuln	**vuln**	depVulnFree	depRemoved	**depVuln**	depVulnRange
bassmaster	0	0	**1**	0	0	**1**	0
crumb	0	0	**2**	1	0	**1**	0
hapi-v2	26	5	**4**	3	0	**3**	3
hapi-v6	0	9	**50**	2	2	**43**	12
js-yaml	18	54	**54**	18	1	**45**	44
libnotify	0	2	**2**	0	0	**2**	2
libyaml	1	1	**6**	0	0	**6**	1
marked	7	109	**279**	52	6	**239**	91
connect	11	215	**299**	29	19	**264**	202
qs	7	59	**187**	37	2	**149**	58
send	6	26	**99**	11	1	**88**	25
st	0	15	**3**	4	0	**3**	11
syntax-validator	1	4	**9**	1	0	**8**	4
validator-v1	15	32	**48**	7	0	**44**	29
yar	0	0	**4**	0	0	**3**	1

Table 6.3: Overview of the changes made from the publication date of the advisory to 12 October 2014

6.3.2 Time taken for updating

Figure 6.7 shows a series of box-plots for advisories in three different publication date ranges. The box-plots present the number of days taken for a library maintainer to update the version range or remove the dependency in order to make it vulnerable-free.

- **Advisories released roughly 14 months ago**: We can identify that the middle 50% of the update time for a vulnerable dependency is between 4 to 9 months for *connect*, 4.4 to 11 months for *js-yaml* and 2 to 4.6 months for *validator-v1*. The box-plot for all three of them reveals that the distribution is positively skewed, the distance from the median and first quartile is smaller than the distance to the third quartile. 25% of the library maintainers that used *connect* and *js-yaml* updated in roughly 130 days or less, while for *validator-v1* it is 55 days or less.
- **Advisories released roughly 8 months ago**: We can see that from an overall perspective the middle 50% of the update time is much less compared to the advisories published 14 months ago, for *hapi-v2* this is between 57 to 84 days, *marked* between 1.28 and 6.5 months, *st* between 22.5 days to 84 days. The distribution is positively skewed for *st* and *marked*, while *hapi-v2* has reasonably symmetric distribution.
- **Advisories that were released approximately 3 months ago**: We can see a trend that the update time is relatively short, the middle 50% of the update time for *hapi-v6* is between 13 and 44 days, roughly within a span of a one month. On the other hand, the update time is between 5.5 and 14 days for *qs* and *send* is between 5 to 16 days, this shows that the time response is within two weeks. 25% of the library maintainers that used *qs* and *send* updated in five or less days.

The results compare advisories published at a similar time window; thus we can identify that despite the popularity of the module, the interquartile range is about same in each of the three scenarios. Furthermore, although some advisories have fewer data points than others, the interquartile range still holds. We can also identify that there is a trend with faster time response for advisories published 100 days ago.

An objection would be that the advisory should be compared from a one year time interval. However, we can see in Section 6.3.1 that the reduction of vulnerable modules for *qs* is more than advisories that were released more than one year ago such as *js-yaml*; thus there are factors that suggest a different form of awareness in updating modules for recent advisories.

6.4 Summary

The quantitative findings report on the volume and spread of modules depending on vulnerable dependencies. The following is inferred from the quantitative analysis:

Section 6.1.1 The vulnerable dependencies is 1/3 of the number of advisory modules used as dependencies in the npm registry.

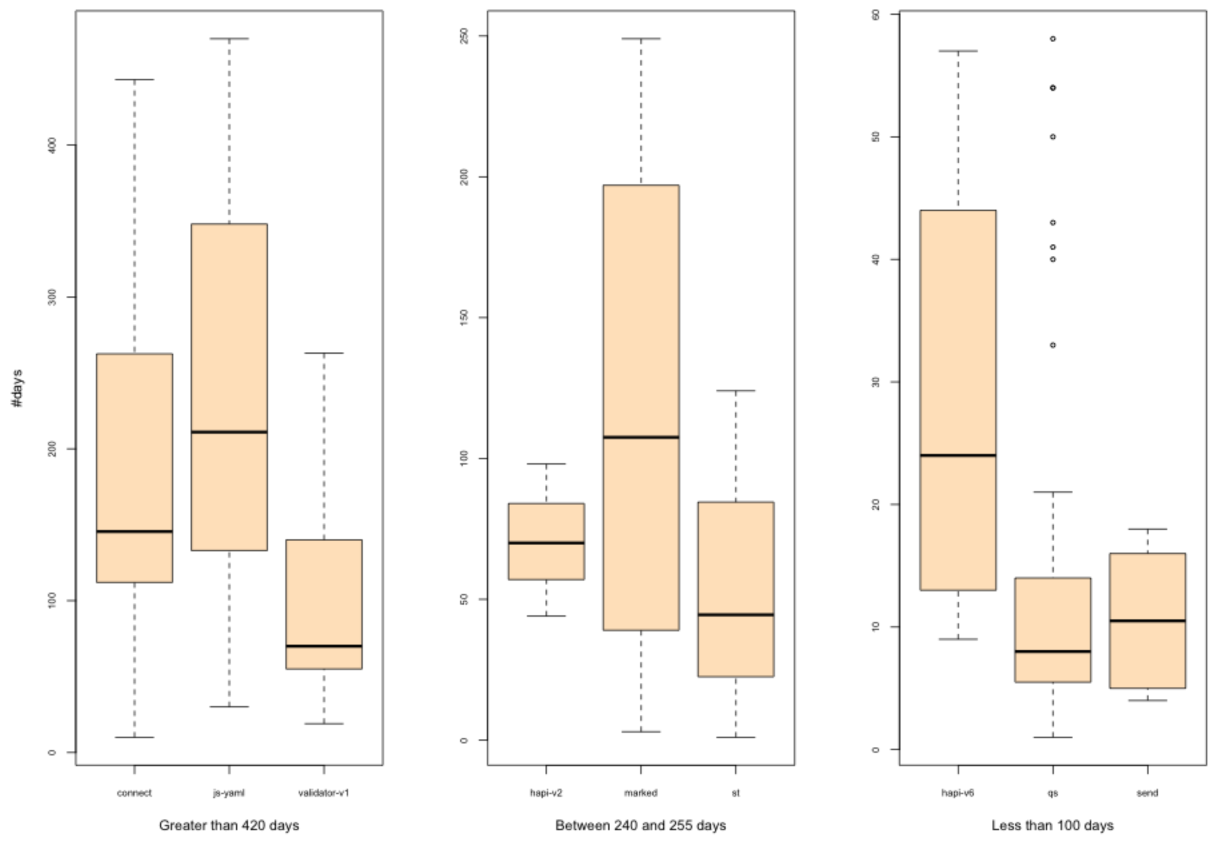


Figure 6.7: Boxplots showing the *resolve-time* distribution over three time periods

Section 6.1.2 Around 2/3 of the vulnerable modules are released in 2013 or earlier. This indicates that a majority of the vulnerable modules are outdated.

Section 6.2 The cascading effect for 12 advisories showed a small increase of indirect vulnerable modules. A single case, *qs* have more indirect vulnerable dependencies than direct dependencies. This confirms the intuition that there is a cascading effect of modules depending on vulnerable dependencies.

Section 6.3 Advisories published more than 100 days ago[4] require several months to resolve the vulnerable dependency, while recent advisories have a faster rate of resolving the vulnerable dependency. Furthermore, the reduction of vulnerable modules from the publication date to 12 October 2014 shows at large that vulnerable dependencies are not resolved (less than 20% are resolved for most advisories).

[4]counted from 12 Oct 2014

Chapter 7

Management of advisory-related vulnerabilities in npm modules

The quantitative findings in Chapter 6 provide an initial insight into the security implications of dependency management of npm modules. In order to better understand the dependency management practices and shortcomings in npm modules, we performed an exploratory qualitative investigation on npm modules hosted on GitHub. This also included code inspection of the modules to confirm the findings of the quantitative results.

7.1 Overview

The qualitative investigation consisted of 85[1] modules hosted on GitHub that were selected based on the criterion defined in Section 3.2. A mix of popular and random projects was selected, 44 modules with a high-degree of popularity and 31 random modules with issues and pull requests. Moreover, 43 of the projects have cascading vulnerable dependencies. There are projects that overlap in popularity, particular with GitHub stars and forks, where one project had a higher number of forks compared to stars. An overview is shown in Table 7.1.

type	**#projects**
depends	4
GitHub stars	31
downloads	18
GitHub forks	1
random	31

Table 7.1: Selected projects

[1]https://github.com/jhejderup/rastogi.js/blob/master/dataset/GithubStudy.pdf

7.2 Comparison of the `package.json` on npm and GitHub

Comparing the `package.json` in the GitHub projects with the information available on the npm registry, we identified six cases where the vulnerable dependency is resolved on GitHub while the npm registry has an outdated version of the *package.json* file. This is shown in Table 7.2[2]. We can observe that for both *phpjs* and *restler*, the version numbers are identical while the version numbers of the dependency differ. The remaining cases have the latest release on GitHub where both the release number of the module and the dependency version have a higher number than on npm. The only exception is for *node-linkedin-simple* where the npm version, 0.0.3 is higher than the GitHub version, 0.0.1. However, the vulnerable dependency version is not resolved on npm.

Module	**Dependency**	**npm version of dep**	**npm latest update**	**Github version of dep**	**Github latest update**
phpjs	send	0.1.0	1.3.2 (2014-03-05)	ˆ0.12.1	1.3.2 (2015-03-07)
restler	qs	0.6.6	3.2.2 (2014-04-24)	1.2.0	3.2.2 (2014-09-09)
couchtato	bagofcli (validator)	∼0.0.5 (∼1.5.1)	0.16 (2013-09-02)	∼0.2.0 (validator removed)	0.2.0 (2014-10-16)
datagen	bagofcli (validator)	∼0.0.8 (∼1.5.1)	0.0.10 (2013-10-23)	∼0.2.1 (validator removed)	0.0.11 (2015-02-10)
cabbie	request-sync (qs)	0.0.5 (0.0.6)	0.0.9 (2014-04-14)	dep removed	1.0.0 (2015-01-15)
node-linkedin-simple	qs	∼0.6.5	0.0.3 (2014-05-13)	∼1.2.1	0.0.1 (2014-08-14)

Table 7.2: Comparison of the module version and dependency version for certain modules

In the dependency graph in Section 6.2, we identified a module with a cascading dependency, *restler* to have the highest out-degree with 108 dependents. *restler* depends on *qs* version 0.6.6. The project has 1471 stars and 303 forks. In the Github version of the `package.json`, the *qs* module is updated to version 1.2.0. Thus, there is an inconsistency with the dependency version on GitHub and npm. If the library maintainer updated the version on npm, 108 modules depending on this module would be able to migrate to a vulnerable-free version.

The cases above indicate inconsistencies with the release management of new versions on two distribution channels; only one channel, GitHub has the latest changes including resolved vulnerable dependencies while npm lacks behind.

7.3 Code inspection of identified vulnerable modules

The quantitative findings were derived from the declared dependency information entered by library maintainers in the *package.json* files. A concern is the use of the dependency in the source code and whether the identified cascading relationships hold from a code execution perspective. This verifies the *accuracy* of the dependency checker by cross-checking the results in the source code. We conducted a shallow code-review to assert this. The complete list of all remarks and studied modules is available online[3]. The following sub-sections investigate five of the advisories.

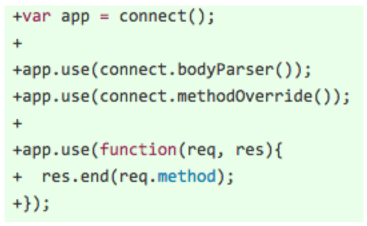
[2]indirect dependencies are specified with the direct dependency in parenthesis on the right hand side

[3]`https://github.com/jhejderup/rastogi.js/blob/master/dataset/CodeReview.pdf`

7.3.1 connect

The *connect* advisory describes a potential XSS vulnerability where the *methodOverride* middleware is enabled and does not properly validate user POST requests. The test cases for the patch[4] of this vulnerability reveal how the functionality is used. This is shown in Figure 7.1 on how the functionality is triggered. In total 13 modules are depending on the *connect* advisory both as in-direct and direct dependency.

In all instances where *connect* is used, the `connect.methodOverrride()` was not enabled. This suggests that the inspected modules are not vulnerable to this attack. An overall conclusion cannot be drawn whether this applies for the modules that are not inspected. However, this demonstrates that advisories may trigger a large number of false positives due to plug-in functionality that is not widely used.

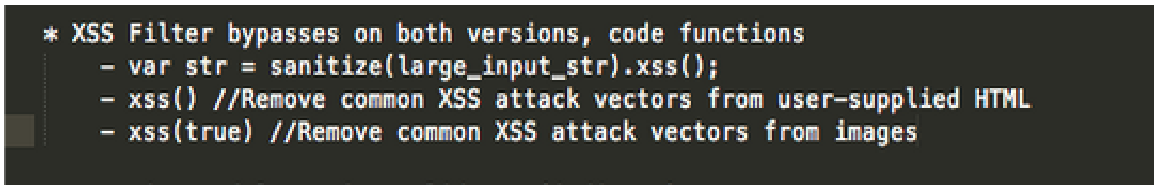


```
+var app = connect();
+
+app.use(connect.bodyParser());
+app.use(connect.methodOverride());
+
+app.use(function(req, res){
+  res.end(req.method);
+});
```

Figure 7.1: Code snippet on how to invoke the methodOverride middleware

7.3.2 validator

The *validator* module has two security advisories that report on potential XSS vulnerability by exploiting the XSS Filter functionality. Since 31 October 2013, the XSS filter in the *validator* module has been removed. In order to keep using the XSS filter functionality, migration to a new dependency is required[5]. Figure 7.2 presents how the functionality can be invoked in three ways.



```
* XSS Filter bypasses on both versions, code functions
    - var str = sanitize(large_input_str).xss();
    - xss() //Remove common XSS attack vectors from user-supplied HTML
    - xss(true) //Remove common XSS attack vectors from images
```

Figure 7.2: XSS filter functionality in validator.js

Six modules depend on the *validator* module, half of the modules are indirect dependencies. The following subsections present the findings for each of these modules.

[4]`https://github.com/senchalabs/connect/commit/126187c4e12162e231b87350740045e5bb06e93a`
[5]`https://github.com/chriso/validator.js/commit/2d5d6999541add350fb396ef02dc42ca3215049e`

direct dependents

- `pump.io`: A social stream server project that is starred 1,184 times on GitHub. The use of the XSS functionality from the *validator* module was discovered in the `scrubber.js`[6] file.
- `frontail`: A module that streams logs in real-time on the browser. The project is starred 146 times. The XSS filter functionality is used in the `index.js`[7] for sanitizing incoming data.
- `nodeportal`: A portal platform developed using mongodb and Node.js. The project is starred 24 times. The use of *validator's* XSS filter was discovered in two files: `ThreadCommentsController.js`[8] (plugin) and `index.js`[9] (core).

in-direct dependents

- `feedparser`: feedparser depends on `resantize` that in turn depends on *validator*. `feedparser` uses the `resantize` module to execute the statement `resanitize.stripHtml(data);` for removing HTML angle brackets. The code inspection of `stripHtml(data);` found no code related to the *validator* function. Thus, this is not vulnerable.
- `couchtato and datagen`: Both modules uses `bagofcli`. By inspecting the code in `bagofcli`, there was no use of the XSS filter function. Thus, this is not a vulnerability.

We can identify for the `validator` module that the vulnerability is invoked in the studied cases for the direct dependencies. An interesting case is the `pump.io` module, a relatively popular social stream server where an attacker could exploit the XSS filter vulnerability. However, the studied cases for indirect dependencies did not show any positive sign for propagation of the vulnerability.

7.3.3 qs

The `qs` advisories are vulnerable to potential Denial of Service attacks due to a memory overflow with sparse arrays in the parsing process. In total 38 modules depend on `qs`: 10 are direct, and the remainder are indirect dependencies.

For the studied cases with direct dependencies, 9 of 10 cases invoke code from the `qs` module. In the cascading dependencies, all modules except 3 had functions calls that

[6] `https://github.com/e14n/pump.io/blob/1029ac71b94dec2b51f6b2aa461b5b2a514e585a/lib/scrubber.js`

[7] `https://github.com/mthenw/frontail/blob/654a62c522f76ebc41094e6f804617c7ea1aa634/index.js`

[8] `https://github.com/saggiyogesh/nodeportal/blob/c0b96f1dad788acd790e222f4091bf22314786e9/plugins/threadComments/ThreadCommentsController.js`

[9] `https://github.com/saggiyogesh/nodeportal/blob/c0b96f1dad788acd790e222f4091bf22314786e9/lib/FormBuilder/index.js`

invoked the *qs* module. This suggests that *qs* is invoked widely in both scenarios and is vulnerable.

In most of the cases, the code sections using *qs* are for performing API calls to various services. This could be a potential problem for user input that is used in combination with API calls. In a few of the cases, the vulnerability can be discarded as the purpose is used for testing.

7.3.4 marked

`marked` is vulnerable to content injection attacks where GitHub flavored markdown (gfm) code-blocks and JavaScript URLs are used. The vulnerability is contextual. The vulnerability is exploitable in the context of processing user input and outputting the result for the attacker in for example a browser window.

Seven modules with direct dependencies and four modules with cascading dependencies were inspected. For the direct dependencies, `marked` is used in six of seven cases, where in one single case it is only declared and not used. The following projects were identified to produce an output in the browser in order to trigger the vulnerability:

- *cleaver*[10]: A project that uses markdown for creating presentations in HTML format. The project is starred over 1,765 times on GitHub. In the `index.js`[11] file, gfm is set to true and the markdown files are rendered in a browser window. This fulfills all the conditions except for a user input location. Therefore, full read and write access to the markdown files is required to exploit the vulnerability. Considering that the module is intended for generation of presentations. The modules does not pose a major threat.

- *styledocco*[12]: A module developed for producing documentation from cascading style sheets (css). The project is starred over 1,004 times. The marked module is set to use gfm in the `styledocco.js`[13]. The possibility of exploiting the vulnerability is low as it is intended for development purpose.

- *glog*[14]: A git push blog server that uses the git model for making changes and blog posts are written in markdown. *marked* is used for parsing to HTML in `index.js`[15]. The module is vulnerable if an attacker manage to get access to the publish functionality of the module. Hence, the risk is low.

For the modules with cascading dependencies, functionality from the *marked* library is invoked in three of four cases. In one case, it is only declared. However, these modules does not fulfill the conditions for invoking the vulnerability. Hence, these are not vulnerable.

[10] *https://github.com/jdan/cleaver*

[11] https://github.com/jdan/cleaver/blob/92b43f9fe6d9710d65bc867f5aca4c90fabbaff2/lib/index.js

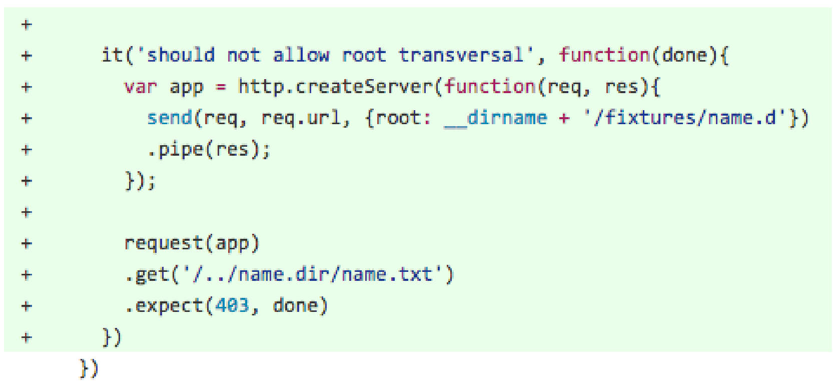
[12] *https://github.com/jacobrask/styledocco*

[13] https://github.com/jacobrask/styledocco/blob/9f12e71a02c5193ed0427aecba7929b230cf0975/styledocco.js

[14] *https://github.com/substack/glog*

[15] https://github.com/substack/glog/blob/257b0b9c17d6164131455279fe330e77002dcd4c/index.js#L492

7.3.5 send



```
+
+     it('should not allow root transversal', function(done){
+       var app = http.createServer(function(req, res){
+         send(req, req.url, {root: __dirname + '/fixtures/name.d'})
+         .pipe(res);
+       });
+
+       request(app)
+       .get('/../name.dir/name.txt')
+       .expect(403, done)
+     })
    })
```

Figure 7.3: Test case showing how the vulnerability was patched in the `send` module

The vulnerability is related to unauthorized access to files with root permission. The test cases for invoking the vulnerability can be found in the following commit[16]. This is also shown in Figure 7.3. We inspected five modules with direct dependency and three with indirect dependencies. For the modules with cascading dependencies, send is not used and only declared. `send` is used in four of five modules:

- *node-xpush*[17]: A real-time communication server intended for messaging or as a push system. *send* is used in `channel-server.js`[18] in the server directory. The functionality is leveraged in a download API point. Therefore, an attacker could abuse this API function to access other files.

- *gitbook*[19]: A command-line tool for creating books using Git and Markdown. Gitbook has over 8,200 GitHub stars. *send* is used in the `server.js`[20] file in the *utils* directory. The risk low as this intended for creation of books in a local machine.

- *phpjs*[21]: `send` is used for testing and thus not in the code-base. Hence, not vulnerable.

- *shiny-server*[22]: Shiny server is used for deploying Shiny applications for the web. Shiny is a web framework for R. The vulnerable function is used in `directory-router.js`[23]. The `DirectoryRouter` has access to the root directory

[16] https://github.com/pillarjs/send/commit/9c6ca9b2c0b880afd3ff91ce0d211213c5fa5f9a

[17] *https://github.com/xpush/node-xpush*

[18] https://github.com/xpush/node-xpush/blob/cbd13716f7287cce01135d4ad7e5f016af761507/lib/server/channel-server.js#L360

[19] *https://github.com/GitbookIO/gitbook/*

[20] https://github.com/GitbookIO/gitbook/blob/34fc2831e0cf0fed01c71cec28d93472d87f455b/lib/utils/server.js#L68

[21] *https://github.com/kvz/phpjs*

[22] *https://github.com/rstudio/shiny-server*

[23] https://github.com/rstudio/shiny-server/blob/393c7b1613580017dd4e227dd4531fe0458ccbe0/lib/router/directory-router.js#L171

where content and applications are stored. It is not very clear how the vulnerability can be exploited. However, this would be a major threat if an attacker could inject a malicious R application that uses the Shiny framework.

7.4 Security discussions within the vulnerable projects

We inspected GitHub issues related to the published advisories or security problems to understand the importance, obstacles and measures taken by library maintainers in the projects.

7.4.1 Mentioning of advisory-related vulnerabilities in projects

In total, 21 of 85 modules have issues involving security problems, and 15 of 21 are directly related to the security advisories. Table 7.3 presents an overview of the findings. We can identify that *qs* and *marked* have the most advisory-related security discussions. The overwhelming majority of projects did not have any issue related to security.

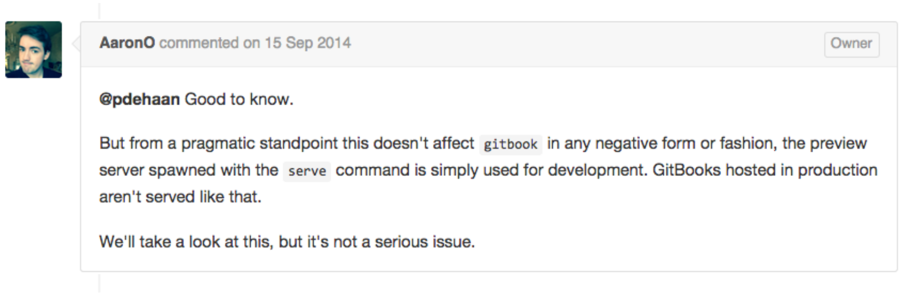
Advisory	**non-NSP**	**NSP (direct)**	**NSP (cascading)**	**Total**
qs	2	3	2	7
marked	0	3	2	5
send	0	1	0	1
connect	3	2	0	5
syntax-error	0	1	0	1
st	0	1	0	1
validator	1	0	0	1
Total	6	11	4	21

Table 7.3: All GitHub projects with security discussions

In 6 of 15 modules, mentioning the vulnerable dependency resulted in resolving and updating the dependency to a vulnerable-free version. A notable example is the *restler* module where a user reported the vulnerable dependency in an issue resulting in the dependency being updated. However, as previously detected, this was not updated on the npm registry.

For 3 of 15 modules did not update due to the intended *purpose of the module*: For example, the modules are specifically built for use in a development environment, or the library maintainer may consider the risk minimal for the intended purpose of the module. One notable example is *gitbook* with over 8,200 stars. One of the library maintainers wrote the following in Figure 7.4 to inform the issuer that the dependency code sections invoke a preview server intended for development and therfore is not a major threat.

In 2 of 15 modules, there is a report of *breaking changes* and as a consequence delaying the time to update the dependency version. The two cases are related to the `marked` module. An example is *harp* with over 2,500 stars that uses *terraform* as dependency, that in turn uses *marked*. In this case, a pull request was created in September 2014, and it was resolved



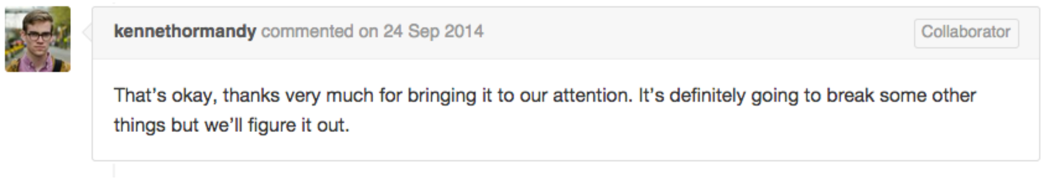
AaronO commented on 15 Sep 2014 Owner

@pdehaan Good to know.

But from a pragmatic standpoint this doesn't affect `gitbook` in any negative form or fashion, the preview server spawned with the `serve` command is simply used for development. GitBooks hosted in production aren't served like that.

We'll take a look at this, but it's not a serious issue.

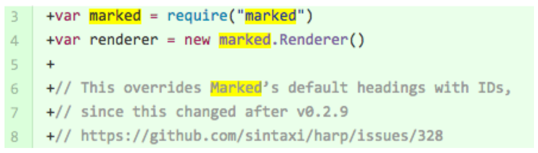
Figure 7.4: Discussion in issue #438



kennethormandy commented on 24 Sep 2014 Collaborator

That's okay, thanks very much for bringing it to our attention. It's definitely going to break some other things but we'll figure it out.

(a) Discussion in pull request #65



```
+var marked = require("marked")
+var renderer = new marked.Renderer()
+
+// This overrides Marked's default headings with IDs,
+// since this changed after v0.2.9
+// https://github.com/sintaxi/harp/issues/328
```

(b) The added code to fix the breaking change

Figure 7.5: Discussions in terraform module that harp depends on

in February 2015. Figure 7.5a shows one of the messages from the collaborator and the final fix in Figure 7.5b.

Finally, in the studied modules, the response is generally positive towards reporting vulnerable dependencies.

7.4.2 Data triangulation

The findings present potential reasons for library maintainers not updating the dependency. Runeson *et al* [28] suggests data triangulation to compensate for enhancing the precision of the empirical study. This implies to take a different angle of the studied object by means of other data sources. However, npm modules are extensively being used in npm and GitHub and finding an external data source is challenging. Therefore, GitHub is still used as the data source. The different angle is to investigate modules that are not in the studied data by finding relevant GitHub projects. The findings are to confirm whether *breaking changes* and *purpose of the module* can be observed in modules outside our studied data. The following is observed:

Breaking changes

In the findings for the `marked` module, we observed discussions around *breaking changes*. The following projects had similar reports:

- `marked-extras`[24]: A non-security related Pull request, User *lukeapage* wrote the following: *"makred-extras references marked and uses it to create a renderer.. This means that still no highlighting is done on less-docs because marked made a breaking change."*

- `remarked`[25]: The following is found: *"3. Breaking changes are introduced to marked.js without bumping the minor version"*

- `remarkable`[26]: User *jonschlinkert* comments the following: *"...1) after months of trying to get some responses (and even offering to help maintain) on marked, I got zero responses, and 2) marked.js has a history of making breaking changes without bumping the minor - kind of frustrating to put it mildly. In fact, the less.js site was down for a couple of days because we had to re-write plugins around marked.js's breaking changes..."*

The intercepted discussions in other pull requests and issues suggests that there are projects outside the scope of the study experiencing problems with *marked.js* breaking changes.

Purpose of the module

As discovered previously, the purpose of the module or the functionality it uses may determine whether the library maintainer is keen on updating the version of the dependency. Below are comments from the repository owners that touch on the purpose or need to update the version of the dependency:

- `apostrophe`[27]: *"Thanks for flagging these! I don't think we are actually using validator's XSS filter. We use sanitize-html for that sort of thing. We'll need to go through these. If you get a chance to test the results of upgrading feel free to send pull requests."*

- `frewil`[28]: *"It's important to note that express-form doesn't actually use the xss filter from validator doesn't use the xss filter unless you explicitly use it on a field. Also, the request is only a devDependency used in the tests. Anyways - I'll take a look at upgrading these anyways and look at some of those package.json errors."*

[24]`https://github.com/assemble/marked-extras/pull/4/commits`
[25]`https://www.npmjs.com/package/remarked`
[26]`https://github.com/jonschlinkert/remarkable/issues/81`
[27]`https://github.com/punkave/apostrophe/issues/170`
[28]`https://github.com/freewil/express-form/issues/20`

- `ember-cli`[29]: *"Ugh, yeah, we should probably fix this. The likelyhood of someone hitting this vulnerability is very low, as the only thing implementing it currently is the model blueprint, but if people start dropping a bunch of garbage like this into their blueprint's HELP.md file, it would be an issue"*
- `supertest`[30]: *"Cool, but FWIW having a vulnerability in your internal tests isn't really a big deal :-)"*
- `yuidoc`[31]: *" updated dependencies in YUIDoc 0.4.0, but security risk I think low because YUIDoc is a developer tools. I hope that this problem resolves now."*
- `yogi`[32]: *"This is a build tool, it doesn't serve production traffic, and it uses the server internally to serve the docs during development."*
- `Ghost`[33]: *"We upgraded the packages (express and body-parser) affected by the qs vulnerability which are used in parts of Ghost that are user-facing and could be subject to the DoS attacks. The packages listed above are either development or test specific and don't pose any real risk as far as I can see. As far as the npm outdated listing goes, as I'm sure you know, with the rate at which new versions come out, it's not practical to test and update everything all the time. However, If you have specific recommendations and rationale for upgrading something it'd be great to get that information."*

These findings confirm the intuition that there are cases where the purpose of the module is a reason for keeping a vulnerable version of the dependency. Furthermore, this also suggests that the vulnerable functionality is not used and the advisories only circle the entire module as vulnerable.

7.5 Summary

A strong majority of the inspected modules have no public discussion concerning security, in general; only 21 modules have security discussions with 15 modules related to NSP-advisories. This could potentially imply that there is a lack of awareness. In one particular case, *pump.io*, a popular module which has the vulnerability open in the code without being mentioned in Github issues. This opens the door for attackers to exploit the XSS vulnerability. *pump.io* was updated latest on 22 June 2014 which could suggest that it is outdated or has a slow development cycle. As previously established, $2/3$ of the modules are outdated.

The code inspection suggests that both the direct and indirect vulnerable dependency are used in the studied modules. However, looking at whether the vulnerability is triggered or not gives mixed results. There are clear cases that show based on the advisory description

[29] `https://github.com/ember-cli/ember-cli/issues/3455`
[30] `https://github.com/visionmedia/supertest/pull/166`
[31] `https://github.com/yui/yuidoc/issues/254`
[32] `https://github.com/yui/yogi/issues/100`
[33] `https://github.com/TryGhost/Ghost/issues/3795`

and test cases how the vulnerability can be exploited. However, in other cases such as *connect*, a middleware is vulnerable, and the core code-base is not affected. This implies that dependency checkers may in some instances generate a high false-positive rate.

Lastly, for the modules with security discussions, there is an indication that dependency checkers are useful and library maintainer resolve it based on reporting. However, for a majority of the projects there is no discussion taking place. Furthermore, the findings also present challenges with dependency management such as with the *purpose of the module* and *breaking changes*.

Chapter 8

Discussion

8.1 Revisiting the research questions

The research objective presented in Section 1.3 targets to establish the *volume* of modules with vulnerable dependencies and explore library maintainers practices with security adversaries and reasons for not updating. In this section, we try to answer the research questions based on the quantitative and qualitative findings from Chapter 6 and 7.

RQ1 What is the prevalence of modules that use at least one dependency that is disclosed as vulnerable?

In Section 5.3, 61508 of 99631 modules use at least one dependency, and around 3192 (3.2%) of those modules use at least one of the advisories as dependency. We identified that 1678 of 3192 (52%) used dependencies with a *version range* with vulnerable versions. Furthermore, 1029 of 3192 (32%) use a version range that directly resolves to a vulnerable version. The vulnerable modules constitute to 1% of the npm registry, shows at large that the npm registry is not substantially affected with the current security advisories.

However, looking to the usage of the advisory dependencies, we can establish that one-third of the modules is vulnerable and potentially exploitable. Half of the modules use a version range that includes at least a vulnerable version. This shows that there is a substantial number of modules with security implications.

The second part of the question looks at library maintainers not resolving the vulnerable dependency. At first, the latest release date of the modules reveals that 36% are released in 2014, and that the remaining 64% of the modules have not been in active development. This suggests that inactive projects could be a potential reason for modules not being updated to the latest version.

Secondly, 85 modules were selected to understand dependency management practices by inferring information from source code, issues and, pull requests on GitHub. The majority of modules lack discussion about security issues. We identified only 21 modules with security discussions, where only 15 are directly related to the nodesecurity project. The findings suggest that there is a low rate of discussions and reports of the advisories.

For almost half of the inspected modules with NSP adivsories, the library maintainer updated the dependency as a consequence of reporting. In the remaining modules, there is an indication of the *purpose of the module* and *breaking changes* as a reason for not updating the dependency. Data triangulation was applied to confirm the findings in other modules.

Answering the question: *"the prevalence of vulnerable modules in the npm registry is minimal. However by analyzing the use of the advisory dependencies, we can see that there is a significant proportion that is vulnerable. Furthermore, a majority of modules lacks security measurements in dependency management where discussions are non-existent in GitHub projects"*

RQ2 What is the cascading effect of modules depending on at least one vulnerable module?

In total, 405 additional modules with vulnerable dependencies were identified by traversing the dependency chain. This is an increase of 39.36% vulnerable modules. Looking at the modules with a cascading effect, the average increase of modules is 21% from depth 1 to 2, while the average increase drops to roughly 7% from depth 2 to 3. *qs* has the highest share of indirect vulnerabilities with around 53%, that is more than direct vulnerabilities.

These cascading effect results are in line with the average shortest path length of 2.67. Furthermore, we investigated the presence of security discussions and use of vulnerable functionality in the source code. This is to establish whether the cascading effect has a vulnerable effect. The results show that there are instances where GitHub users have reported vulnerabilities that are cascading. In the majority of cases, we could identify that the source code was indeed invoked. For *send*, *validator*, *connect* there was no code involved at all. The mentioning of security advisories only appeared in cascading modules that depend on *qs* and *marked*.

Answering the question: *"A cascading effect appears for twelve advisories with a minor growth of modules and the longest observed depth reaching five hops. The GitHub data confirms instances where the vulnerability propagates in the dependency chain by code inspection. Thus, cascading dependencies could serve as a potential threat for modules depending on it unknowingly."*

RQ3 What is the time latency for updating to a non-vulnerable version range for a dependency?

Section 6.3 analyzed the time taken for updating a vulnerable dependency to a vulnerable-free dependency by either updating the version or removing the dependency. This was conducted by dividing advisories into three time windows: advisories published later than 420[1] days or later, between 240 and 255 days and less than 100 days.

Advisories published 14 months ago, the middle 50% of the update time for *js-yaml* and *connect* is between 4 to 9 months, which is a significant period for patching a vulnerable

[1]days are counted from 12 Oct 2014

dependency. On the other hand, for advisories published three months ago, the middle 50% is between 5 to 16 days. This is a substantial change in the responsiveness to the advisories published in 2013. Considering the reduction rate of vulnerable modules from the start of the publication date to 12 October 2014, the reduction is much higher for *qs* than for *connect*. Furthermore, the reduction of vulnerable modules from the publication timestamp of each advisory to 12 October 2014 showed that less than 20% resolved the vulnerable dependency. This particularly shows that security vulnerabilities are not resolved at large.

Answering the question: *"Looking at advisories published at three different time windows, there is a positive indication in recent advisories that modules are patched within in two weeks. This is a substantial improvement to advisories published in 2013 where several months were taken to update. Considering this, further investigation is needed to establish what factors influence the time taken for patching"*

8.2 Threats to validity

The validity of the study is discussed using the four classes of validity as described by Runeson *et al* [28].

8.2.1 Construct validity

This class of validity puts the mapping of the inferred research variables to the research questions under scrutiny.

Investigating the prevalence of modules using vulnerable modules was conducted by measuring the number of identified modules. **RQ1** is formulated with respect to *known vulnerabilities* and thus the results solely reflect the prevalence of modules using disclosed vulnerable dependencies. The cascading effect (**RQ2**) is measured by the *number of modules (width)* per *depth* level. This reflects the *propagation* of vulnerable dependent modules.

Potential construct threats relate to the qualitative part where modules with direct and indirect vulnerable dependencies were analyzed. The form of solely using GitHub data for performing the qualitative analysis may raise potential threats. The expertise and background of the users participating in the discussions are unknown and may provide incorrect or biased information. The library maintainers could be biased in the reporting that a vulnerable dependency is not harming the module because security problems may give negative publicity. A second form of bias is that the library maintainer may state an opinion or negative experience with the vulnerable dependency as a consequence of delaying the update. The threat is of minor character since the discussion is taken place mostly on popular projects and progress/actions from the discussions is visible in the code/refactoring work.

The information digested from projects in GitHub comes from an essentially *uncontrolled* environment. Therefore, interviews with the library maintainers could provide better insight into the dependency management and how security is dealt with. A concern is with projects that have no security discussion; it could be possible that this is done *internally*. This would also be a more useful strategy to perform data triangulation.

The last form of threat is with the researcher bias; the majority of the work was inspected and interpreted by one researcher. For a few projects in the qualitative phase another researcher was involved. Ideally, an interview with the library maintainer to confirm the understanding of the discussions will enhance the confidence of the interpreted data.

8.2.2 Internal validity

The internal validity concerns with investigating whether any additional third-party factors could influence the studied factors.

A potential threat is the matching formula for detecting modules with vulnerable dependencies from the implemented tool, *Rastogi.js* that is described in Chapter 4. Furthermore, the obtained results are solely inferred by declared dependency data. The usage of the dependency in the source code is not known; thus the results may be *optimistic*.

In order to mitigate these threats, the implementation is covered with unit tests to ensure the functionality to our best ability. Secondly, the results were subject to manual inspection and cross-checked with the npm registry data on the *skimdb* to confirm the findings. Likewise, the same was conducted on modules with cascading vulnerabilities. Finally, the qualitative study included a code review to inspect whether the dependency is used in the source code. However, this does not compensate for static code analysis to analyze a larger set of modules.

8.2.3 External validity

The external validity looks at how the result can be generalized. The study that is limited to the npm registry and JavaScript as a language. Therefore, the results may not be valid for other large central software repositories such as Maven or NuGet. The studied projects are written in JavaScript that may have different security problems and developer practices in contrast to other programming paradigms such as Java or Python.

All studied modules are open-source projects. Hence, proprietary projects may produce different and contradicting results to this study.

8.2.4 Reliability

A primary concern is the replicability of the study. A major constraint is the rapid pace the npm registry is evolving with constant changes. In order to mitigate this, a snapshot of 12 October 2014 was constructed to make it reproducible. Likewise, the *Rastogi.js* is available for download for conducting the analysis performed in this study.

A minor threat to the reliability is with the snapshot of *skimdb*. The objective of the snapshot was to store necessary metadata where missing information is obtained from *skimdb*. Information such as the author is not stored and requires to be retrieved from *skimdb*. A problem that could arise is that this information is altered in the *skimdb* and could produce different results.

8.3 Implications

The qualitative study touched on aspects such as the *usefulness* of the security advisories in the context of dependencies. Although there is a positive signal towards reports of vulnerable dependencies in Github issues or pull requests, the role of security advisories and dependency checkers for security assessment raise certain questions.

The vast majority of dependency checkers have a simple process of categorizing dependencies as vulnerable or not by just relying on the library identifier and version. The findings of our study suggest that this information is not *sufficient* to highlight a weakness in a software system. Several of the discussions in Github issues and code inspections show: *(1)* the vulnerable functionality as described in the advisory is never used or is a plug-in functionality, *(2)* the runtime context is for development use, *(3)* the severeness of the vulnerability or the usage in the source code cannot be triggered by user supplied content.

In some cases, there are valid reasons specified by library maintainers for not resolving the vulnerable dependency. This could cause confusion for end-users relying solely on dependency checkers in build processes or Continuous Integration (CI) systems.

An identified concern for library maintainers to resolve a vulnerable dependency is *breaking changes*. In our findings, a module relying on the *marked* dependency required roughly four to five months to adjust to the changes. This shows that the refactoring work and the severeness of vulnerability could be problematic in software developed at a rapid pace. A grading scale of the vulnerability in the context of the software system could aid library maintainers to plan and prioritize patchwork. There is a grading standard called Common Vulnerability Scoring System (CVSS[2]) for quantifying the severeness of a vulnerability. However, the proposed grading scale specifically grades the threat based on how the vulnerable functionality is used in the software system.

Many of the concerns pointed out above above call for an intelligent dependency checker where the library maintainer plays an important role in providing relevant security information in the context of the library. Based on a security advisory, the library maintainer can determine whether it poses a threat to use it in a software system or not. Furthermore, some form of analysis of the source code with reference to the vulnerability described in the advisory could give a more informed time schedule for the patchwork. Although a security advisory is graded with highest severeness, the library maintainer should not rush with patch work in case minor functionality is used in the dependency and an update of the version introduces breaking changes.

[2]`https://nvd.nist.gov/cvss.cfm`

Chapter 9

Related Work

Examining large and scalable software ecosystems has sparked attention in the software research community in recent years. Significant contributions have been made to understand different aspects of third-party source code artifacts and dependencies from a version or time-based dimension. The majority of the work has been conducted on Java third-party libraries and the MAVEN repository. This chapter presents previous research in related areas of the research topic.

9.1 Software Security

9.1.1 Third-party libraries and dependencies

Mitropoulos *et al* investigated security vulnerabilities in the MAVEN Central Repository from a software evolution viewpoint and presented a dataset with vulnerability reports of all hosted MAVEN projects [19]. From the 115,214 JARS, 45,559 had at least one report of "malicious code" and 5,341 projects had at least one report of "security bugs" related to user-input validation. The authors particularly examined the correlation between number of security bugs and the JAR size as the projects evolve. The correlation was not strong and suggest that early introduction of security bugs may not necessarily correspond to an increase in the number of bugs over time.

Cadariu leverages the OWASP Dependency Checker to analyze known security vulnerabilities in MAVEN POM files in open-source and industrial applications [11]. His work included an empirical study on 12,100 projects in the Maven Central Repository and he developed a tool to alert system maintainers. Main findings included that almost one out of five open-source projects have one or more vulnerable dependencies. However, the precision rate was relatively low at 0.14 inducing a high false-positive rate with a recall rate of 80%. In order to show the usefulness of the alert system, the author evaluated the system at several companies where the outcome suggested it is very useful. Xia *el* mitigates vulnerability issues in reuse of outdated third-party libraries in software systems. The study indicates that a large number of open-source software systems relied on outdated software dependencies and investigated a number of reasons for not upgrading to newer versions [33].

The work in the publications mentioned above is closely related to our work; we differ in the approach to analyzing dependencies and modeling dependencies. Secondly, we target the practices of dependency management from a security perspective by conducting a qualitative study.

9.1.2 Client-side JavaScript

There is a plethora of security research in client-side JavaScript on Cross-site scripting (XSS) and Cross-site request forgery (CSRF) attacks in browsers. One of the primary concerns with including third-party JavaScript libraries in a web application is the privileges it enjoys in the browser context; a relative problem is that external code has same execution rights as their code. Thus, this allows for cookie hijacking and stealing credit-card information.

Nikiforakis *et al* performs a large-scale study of remote inclusions of third-party JavaScript libraries in the top 10,000 Alexa Website [21]. A metric was proposed to evaluate the quality of maintenance by the library providers; the metric takes into account the security measures to enforce a secure remove inclusion of the library and ranks accordingly. For instance, aspects such as usage of SSL/TLS, Anti-XSS and Anti-Clickjacking protocols are in use. The evaluation showed that a prominent number of top-ranking domains such as `paypal.com` are relying on high-maintenance providers while around 60% of the low-maintenance providers use worst-ranked providers. The work also features evolution aspects such as the number of inclusions that are deleted or added over time; this is of interest in our work as well.

A few publications describe mitigation techniques to counterattack web tracking of third-party libraries. Agten *et al* presents JSand, a novel sandboxing approach of remote JavaScript inclusions that does not require browser modification [9]. This is achieved by using an object-capability model, that acts as a secure subset of JavaScript with strict security policies. The prototype itself is a library and provides a sandbox environment for remote inclusions. A different approach is to perform dynamic, static or hybrid analysis of third-party libraries. Hedin *et al* [15] presents a prototype that performs information flow analysis on JavaScript code. Information flow analysis is essential to track how data flows in the application and is an important tool to enforce strict data policies. Similarly, mitigation strategies on dealing with web security are proposed in [17, 27].

9.1.3 Server-side JavaScript

Server-side JavaScript is a relatively unexplored research area with a small number of publications. Ojaama *et al* investigates potential security pitfalls in the Node.js ecosystem [22]. The authors acknowledge that there are security problems inherited with the architectural design and JavaScript as a language. The single-threaded event loop is more vulnerable to Denial-of-Service attacks in contrast to multi-threaded environments. The development model of Node.js is challenging for many developers as its entirely asynchronous. Thus, there are several scenarios due to misunderstanding that cause unintentional Denial-of-Service vulnerabilities.

Finally, Sullivan demonstrates how client-side vulnerabilities are inherited in server-side JavaScript with a particular focus on Node.js and MongoDB [31]. One such example is the severe consequences of bad user input validation at the server-side, since injection attacks at the server-side can cause more harm in comparison with client-side.

9.1.4 Vulnerability Database

A hand-full of publications leverage vulnerability databases such as the NVD to assess security problems in software systems and components. Zhang *et al* attempted to create a practical prediction model for zero-day vulnerabilities using information from NVD. The authors investigated the challenges in the data source and concluded that the information from NVD is not sufficiently refined for predicting zero-day vulnerabilities [35].

9.2 Software Quality Assurance

Reliance on third-party libraries is a critical factor in achieving a high-degree of software reuse in cost-effective software development. A challenge with software reuse is mitigating potential risks involved in depending on the third-party source code. Few papers have proposed quality indicators for third-party libraries by studying the usage of third-party dependencies in software systems.

9.2.1 Wisdom of crowds

Raemaekers *et al.* proposed a rating formula for risk assessment of third-party libraries in software systems [24]. The risk assessment is based on a *wisdom of the crowds* approach with the following two dimensions: a popular library is perceived safer due to being more widely accepted and common, and a software system with many dependencies is conceived unsafe as it is exposed to higher number of external threats. The authors presented two metrics by extracting dependencies from the source code: *commonality rating* and *isolation rating*. The commonality score takes into account the frequency rating of a library divided by the number of dependencies. The isolation rating measures the distribution of third-party library imports e.g., 20% of the files in a system uses this dependency. The empirical evaluation consisted of 106 open-source systems and 178 proprietary systems written in Java. The authors point out that the metrics provide insightful indications of potential risks. The study does not consider the library version used.

Mileva *et al* uses a similar approach on *wisdom of the crowds*, and the authors leverage the knowledge of the crowd to aid developers in making an informed decision on which version of a library to use [18]. Establishing the trends and usage patterns of different library versions is achieved by performing a historical version analysis of dependency usage in software projects. The authors examined dependencies of 250 APACHE projects and studied the usage evolution of 450 different library versions for a period of two years. The empirical study imposes two important quality parameters: the number of times a library version was changed in a software project is a strong indication of instability and the popularity of a particular version. Thus, these two parameters provide insight to the stability of a particular version of a library. If it is shown that a newer version, for instance, has bugs, users tend to switch to an older as its considered reliable. The authors developed a tool called AKTARI that recommended library versions based on these two quality indicators.

Previous studies have shown that software visualization can aid in decision-making of refactoring jobs as well as in understanding the current state of a software system. Software dependencies can be represented as a graph in order to assess network properties and relations. This has shown to be useful in predicting issues in dependency updates. Kula *et al* visualizes the evolution of software systems and its dependencies from two viewpoints [16]. The *System-centric Dependency Plot* (SDP) viewpoint presents an overview of the evolution of the dependency changes within a software system. The second viewpoint, *Library-centric dependents Diffusion Plots* (LDP) is based on *wisdom of the crowds* and explores usage patterns of third-party libraries in other systems. This is similar to the work in [18, 24]. However, this work presents visual aids to assess better migration paths in dependency management.

9.2.2 Migration and stability

Raemaekers *et al* introduced four metrics to measure the stability of third-party libraries [25]. The paper presents challenges with evolving third-party components with emphasis on breaking backward compatibility. Neglecting maintenance of dependencies over time can cause a considerable amount of code rework when upgrading dependencies. Therefore, a third-party library that ensures backward compatibility is essential for the stability of a software system. The authors studied the stability of libraries by examining dependencies in MAVEN build files from a set of industrial projects and performed source code analysis for each version of the dependency. The stability is composed of four metrics that look into evolving changes of the method (e.g., added or removed methods over time, the percentage of new methods, etc.).

Schoenmakers *et al* presents a framework that evaluates the complexity of upgrading a software module in order to minimize the potential for a large-scale refactoring [30]. The framework analyzes the dependencies of the module as well as the module's public interface to determine the minimal upgrade path by searching for possible valid update configurations. A configuration denotes the required dependency upgrades in other modules of the system. However, the problem of finding an optimal solution is NP-complete. The authors apply approximate heuristics to search for valid configurations. Optimal solutions were found in 58% of the scenarios.

In [32], Teyton et al presents a prototype to suggest possible migration libraries for outdated dependencies in Java. Dependencies are extracted from the source code, and potential library migration paths are suggested. The algorithm for suggesting libraries is based on trends of mining existing migration paths in VCSs. For instance, a current dependency uses *org.json* and *jackson* library is suggested based on previous migration patterns. However, the recommendation part has low precision and requires manual intervention to remove incorrect migration paths in the mining process.

9.3 Analyzing Software Ecosystems

There are several publications geared towards analyzing complete software ecosystems, and most of the work has been done on the Maven Central Repository.

Grechanik *et al* performed a large-scale empirical study on 2080 Java projects from the Sourceforge repository [14]. The work presents statistics on trends and usage of Java source code elements. Understanding code patterns and developer practice can be useful in enhancing program analysis and compiler optimization. A relevant aspect of our topic is that the authors used SQL queries to state the research questions in order to answer them. This is a practical example of using a database model to formulate and examine the source code.

In a follow-up paper to [25], Raemaekers *et al* presents a complete dataset of a snapshot of the MAVEN Central repository in [23]. The Maven Dependency Dataset comprises of calculated information of source code elements, evolution metrics and call graph information. The dataset is useful for conducting empirical studies of evolution in third-party software libraries.

Recently, work has been performed to examine the different types of software bugs in the MAVEN repository [20, 29]. In both papers, *FindBugs* is used to analyze the Java bytecode of third-party libraries in order to generate bug reports. The findings presents common bugs related to, for instance, bad code practices, malicious code and security issues. A useful application of this data is library stability from a code quality perspective. Schwittek *et al* investigated the software re-use of third-party components in enterprise applications. The findings suggest that previous empirical evaluations and assumptions indicate that software re-use is widely used in enterprise systems. A study on 36 Java applications, which in total depend on 70 third-party libraries, shows that on average a project depends in a minimum of 16 to a maximum of 161 components.

Chapter 10

Conclusions and Future Work

The central theme of the thesis is to investigate the presence of modules depending on vulnerable dependencies in the npm registry and to understand the interplay between security advisories and library maintainers in modules that are not updated. To the best of our knowledge, this is the first study that explores a large-scale JavaScript repository from a dependency and security perspective. This chapter presents the scientific contributions that were made throughout the study.

10.1 Contributions

- **Rastogi.js:** Dependency analysis framework developed for analyzing npm modules with the possibility to extend with other analyzer functionality.
- **Datasets including results and a snapshot of the npm registry:** The datasets and the results are publicly available for researchers to reproduce our results.
- **Data about the modules with vulnerable dependencies in the npm registry:** The obtained results provides the presence of the number of modules with vulnerable dependencies in the npm registry.
- **Looking into practices of library maintainers:** Studied data from Github to infer practices and challenges with resolving vulnerable dependencies.
- **New insights in the treatment of vulnerabilities in the Javascript ecosystem:** In particular, we have seen that one-third of the modules that use the advisories as dependencies are vulnerable. A cascading effect was observed in most advisories and could serve as a potential threat for modules depending on it unknowingly. We have also seen that the time for updating vulnerable dependencies is substantially faster for recent advisories compared to advisories published in 2013. Studying GitHub issues and pull request showed an overall lack of security discussions. In modules with security discussions, there is an indication that *breaking changes* and *purpose of the module* are reasons for not resolving the vulnerable dependency.

10.2 Conclusions

The research objective of this thesis is to establish new knowledge into dependency management of security advisories and to understand why we have vulnerable modules remaining unresolved in JavaScript projects. The formulated research questions in Chapter 1 tackled this objective by first establishing the volume of npm modules with vulnerable dependencies by the use of Rastogi.js. This was followed by an investigation of the response time and a qualitative study to identify practices with managing dependencies.

The prevalence of modules depending on vulnerable dependencies is substantial in relation to the total number of npm modules depending on the advisory modules. However, two-third of these modules are outdated based on the last release date and latest commit on GitHub. This is explained by our finding that the majority of the modules are abandoned; thus no action is taken to resolve the vulnerable dependency.

Advisories published in 2013 required several months before the vulnerable dependencies were resolved; on the other hand, there is an indication that newer advisories have a faster resolve time where the middle 50% of the update time is within one month. Although this is a positive trend, further factors have to be explored in order to understand why certain modules are updated faster than others.

The dependency management practices highlight both positive and negatives outcomes: Advisories are seen as a positive indication to address potential security weaknesses. On the other hand, reports of a vulnerable dependency are not an immediate sign of a security weakness in a module. There are several factors to this: the module is used in a development environment, the vulnerable functionality of the dependency is not used, or there is a little risk that the vulnerability can be triggered.

These findings points to the need for further research into dependency management practices of vulnerable dependencies in order to enhance the vulnerability assessment of a software system.

10.3 Future work

This is an initial study of dependency management from a security perspective. There are a number of directions that can be explored further:

- **Intelligent Dependency Checker:** We proposed in Chapter 8 a dependency checker that leverages information from the security advisories and library maintainers. A potential prototype can be built by using advisories from NSP and data from GitHub to aid developers making informed decisions about the security situation.
- **Grading the severeness of a security advisory from the context of a module:** Classifying modules based on the actual code usage of the dependency in the software system and the vulnerability description of the advisory. This requires taint analysis to assess where the dependency code is leveraged and whether this can be exploited with the advisory description. An automated process of this type of assessment could assist library maintainers evaluate the urgency and time required for performing a patchwork (in case of backward compatibility issues).

- **Security Advisories and Breaking Changes:** How do breaking changes impact the work of updating security vulnerabilities and how much of an advantage is this for an attacker? A study of the software architectures and the amount work required for patching a vulnerability is of interest. How can we develop best practices that allows us to resolve security issues with backward compatibility in a manageable small timespan?

Bibliography

[1] From relational to neo4j. http://neo4j.com/developer/graph-db-vs-rdbms/.

[2] Node.js Mongodb Driver. http://docs.mongodb.org/ecosystem/drivers/node-js/.

[3] Introduction to Google V8. https://developers.google.com/v8/intro, September 2012.

[4] Types of cross-site scripting. https://www.owasp.org/index.php/Types_of_Cross-Site_Scripting, 2013.

[5] ebay security flaw has existed for months. http://www.bbc.com/news/technology-29279213, 2014.

[6] Faq on sa-core-2014-005. https://www.drupal.org/node/2357241, 2014.

[7] Sonatype Open Source Development and Application Security Survey. http://img.en25.com/Web/SonatypeInc/%7Becc5f531-1f0b-4a9c-8f52-6514c6516056%7D_2014_Survey_Final_Results.pdf, 2014.

[8] Time-Based Versioned Graphs. http://www.neo4j.org/graphgist?608bf0701e3306a23e77, 2014.

[9] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H Phung, Lieven Desmet, and Frank Piessens. Jsand: Complete client-side sandboxing of third-party javascript without browser modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 1–10. ACM, 2012.

[10] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[11] Mircea Cadariu, Eric Bouwers, Joost Visser, and Arie van Deursen. Tracking known security vulnerabilities in proprietary software systems. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 516–519. IEEE, 2015.

[12] Ryan Dahl. Node.js. `http://nodejs.org/jsconf.pdf`, 2009.

[13] Asger Feldthaus, Max Schafer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for javascript ide services. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 752–761. IEEE, 2013.

[14] Mark Grechanik, Collin McMillan, Luca DeFerrari, Marco Comi, Stefano Crespi, Denys Poshyvanyk, Chen Fu, Qing Xie, and Carlo Ghezzi. An empirical investigation into a large-scale Java open source code repository. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 11. ACM, 2010.

[15] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. Jsflow: Tracking information flow in Javascript and its APIs. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1663–1671. ACM, 2014.

[16] Raula Gaikovina Kula, Coen De Roover, Daniel German, Takashi Ishio, and Katsuro Inoue. Visualizing the Evolution of Systems and their Library Dependencies. 2014.

[17] Jonathan R Mayer and John C Mitchell. Third-party web tracking: Policy and technology. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 413–427. IEEE, 2012.

[18] Yana Momchilova Mileva, Valentin Dallmeier, Martin Burger, and Andreas Zeller. Mining trends of library usage. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, pages 57–62. ACM, 2009.

[19] Dimitris Mitropoulos, Georgios Gousios, Panagiotis Papadopoulos, Vassilios Karakoidas, Panos Louridas, and Diomidis Spinellis. The Vulnerability Dataset of a Large Software Ecosystem. In *Proceedings of the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*. IEEE Computer Society, sep 2014.

[20] Dimitris Mitropoulos, Vassilios Karakoidas, Panos Louridas, Georgios Gousios, and Diomidis Spinellis. The bug catalog of the maven ecosystem. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 372–375. ACM, 2014.

[21] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: Large-scale evaluation of remote JavaScript inclusions. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 736–747. ACM, 2012.

[22] Andres Ojamaa and Karl Düüna. Security Assessment of Node.js platform. In *Information Systems Security*, pages 35–43. Springer, 2012.

[23] Steven Raemaekers, Arie van Deursen, and Joost Visser. The maven repository dataset of metrics, changes, and dependencies. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 221–224. IEEE Press, 2013.

[24] Steven Raemaekers, Arie van Deursen, and Joost Visser. Exploring risks in the usage of third-party libraries. *Software Improvement Group, Tech. Rep*, 2011.

[25] Steven Raemaekers, Arie van Deursen, and Joost Visser. Measuring software library stability through historical version analysis. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 378–387. IEEE, 2012.

[26] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do. In *ECOOP 2011–Object-Oriented Programming*, pages 52–78. Springer, 2011.

[27] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. Detecting and Defending Against Third-party Tracking on the Web. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 12–12, Berkeley, CA, USA, 2012. USENIX Association.

[28] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131–164, 2009.

[29] Vaibhav Saini, Hitesh Sajnani, Joel Ossher, and Cristina V Lopes. A dataset for maven artifacts and bug patterns found in them. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 416–419. ACM, 2014.

[30] Bram Schoenmakers, Niels van den Broek, Istvan Nagy, Bogdan Vasilescu, and Alexander Serebrenik. Assessing the complexity of upgrading software modules. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 433–440. IEEE, 2013.

[31] Bryan Sullivan. Server-side Javascript injection. *Black Hat USA*, 2011.

[32] Cédric Teyton, Jean-Rémy Falleri, Marc Palyart, and Xavier Blanc. A study of library migrations in Java. *Journal of Software: Evolution and Process*, 2014.

[33] Pei Xia, Makoto Matsushita, Norihiro Yoshida, and Katsuro Inoue. Studying Reuse of Out-dated Third-party Code in Open Source Projects. *Information and Media Technologies*, 9(2):155–161, 2014.

[34] Michal Zalewski. Browser security handbook. *Google Code*, 2010.

[35] Su Zhang, Doina Caragea, and Xinming Ou. An empirical study on using the national vulnerability database to predict software vulnerabilities. In *Database and Expert Systems Applications*, pages 217–231. Springer, 2011.

Appendix A

Glossary

DOM: Document-Object-Model

XSS: Cross-site scripting

CSRF: Cross-Site Request Forgery

OSS: Open-source software

DoS: Denial-of-Service

NVD: National Vulnerability Database

NSP: Node Security Project

AJAX: Asynchronous JavaScript and XML

HTML: HyperText Markup Language

I/O: Input/Output

SQL: Structured Query Language

CVE: Common Vulnerabilities and Exposures