Identification of Dependency-based Attacks on Node.js

Brian Pfretzschner
Technical University of Darmstadt, Germany
brian.pfretzschner@stud.tu-darmstadt.de

Lotfi ben Othmane
Iowa State University, USA
othmanel@iastate.edu

ABSTRACT

Node.js executes server-side JavaScript-based code. By design Node.js and JavaScript support global variables, monkey-patching, and shared cache of loaded modules. This paper discusses four attacks that exploit these weaknesses, which are: leakage of global variables, manipulation of global variables, manipulation of local variables, and manipulation of the dependency tree. In addition, it describes the static code analysis that we implemented for T.J. Watson Libraries for Analysis (WALA) to detect the identified attacks and the evaluation of the analysis. The analysis is integrated into OpenWhisk, an open source serverless cloud platform.

KEYWORDS

Software security, Dependency-based attack, Cloud computing, Node.js

ACM Reference format:
Brian Pfretzschner and Lotfi ben Othmane. 2017. Identification of Dependency-based Attacks on Node.js. In *Proceedings of ARES '17, Reggio Calabria, Italy, August 29-September 01, 2017,* 6 pages.
https://doi.org/.1145/3098954.3120928

1 INTRODUCTION

Node.js is a popular open source server-side run-time platform for JavaScript applications [1]. Node.js applications use third-party JavaScript modules as dependencies. The community shares and distributes these modules through the Node Package Manager (NPM) repository [3]. The NPM lists about 500.000 packages;[1] thus, it is the largest language-specific package manager. The number of dependencies, which are often third-party dependencies available in NPM, of a typical Node.js application can easily exceed several hundred. (A dependency could use other dependencies.) The installation of all dependencies is a prerequisite for running the application. Node.js dependencies have the same access level to the environment as the main application.

[1]Source: http://www.modulecounts.com/, accessed on May 29, 2017.

ARES '17, August 29-September 01, 2017, Reggio Calabria, Italy

ACM ISBN 978-1-4503-5257-4/17/08...$15.00
https://doi.org/.1145/3098954.3120928

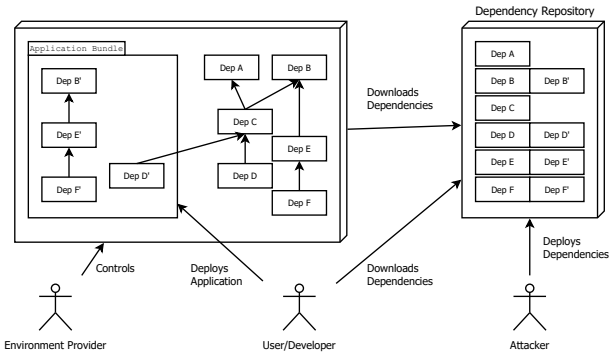

Figure 1: Parties involved in a dependency-based attack scenario.

Figure 1 illustrates the dependency ecosystem for Node.js. A user of the environment deploys an application bundle that includes all dependencies but also uses pre-deployed libraries managed by the environment provider. An attacker could use the dependencies to stage an attack on target Node.js applications: use of third-party dependencies allows for potential attacks because dependencies could be controlled by attackers [13, 16].

This paper addresses two questions:

(1) How attackers can exploit third-party dependencies in Node.js environment? and
(2) How to mitigate such attacks?

Existing work on the third-party dependencies ecosystem focused on the impact of vulnerabilities in these dependencies [10, 17, 18]. In contrast, this paper focuses on malicious code located in third-party dependencies and actively tries to attack the dependent applications by exploiting weaknesses of JavaScript and Node.js.

Performing attacks discussed in this paper requires that the attacker successfully manages to get its malicious dependency installed in the victim applications through e.g., distribution of the dependency through NPM or modifying a dependency available in NPM. The aim of the attacker is to overtake the host applications, to manipulate or leak data processed in the applications, to damage the execution environment, or to affect other tenants of the execution environment.

There are mainly two approaches to mitigate these attacks– besides writing code that avoids these attacks. The first approach is to modify Node.js environment to fix the underlying weaknesses discussed in Section 2.1. Unfortunately, these

Table 1: Comparison of dependency-based attacks in terms of their specific properties and underlying flaws.

	Scope	Aims for	Source	Weaknesses
Global Leakage	Global	Data	JavaScript	Global variable
Global Manipulation	Global	Data & Control	JavaScript	Global variable and monkey-patching
Local Manipulation	Local	Data & Control	Node.js	Loaded module cache and monkey-patching
Dependency Tree Manipulation	Local	Data & Control	Node.js	Loaded module cache and monkey-patching

weaknesses are features of JavaScript and Node.js and are design choices. Changing JavaScript interpreter and Node.js to address these weaknesses implies that most of existing applications will fail.

The second approach is to use code analysis [9] to identify such attacks. The analysis needs to be used such that the execution environment denies execution of applications that include dependency-based attacks. We applied this approach and developed a static code analysis for T.J. Watson Libraries for Analysis (WALA) to identify dependency-based attacks.

The contributions of this paper are:

(1) enumeration of dependency-based attacks,
(2) development of static code analysis using WALA to identify such attacks, and
(3) integration of the analysis into `OpenWhisk` [4], an open source serverless cloud platform.

The paper is organized as follows. First, we discuss in Section 2 the 4 dependency-based attacks on Node.js that we identified. Then, we discuss the code analysis that we developed using WALA to address these attacks (Section 3). Next, we discuss the related work (Section 4). We conclude the paper afterwords (Section 5).

2 DEPENDENCY-BASED ATTACKS

This section discusses 3 JavaScript and Node.js weaknesses, 4 attacks that use these weaknesses, and how to perform the attacks. Table 1 summarizes these attacks.

2.1 Weaknesses

We discuss in this subsection two Javascript weaknesses and one Node.js weakness.

2.1.1 Global variable. The JavaScript language functionalities are realized in functions and variables (aka objects) that are stored in the global namespace/scope. The elements of the global namespace are shared among all the modules of the given Node.js application including their dependencies. The use of global variables is considered a bad practice [14]. Programs that use global variables can neither control access to these variables nor protect them from external modifications.

JavaScript global namespace includes several sensitive elements including `String` and `RegExp`, which are responsible for string manipulation and evaluation of regular expressions. The variables can be accessed and manipulated from any module within the application, including the dependencies. Legitimate and malicious changes get immediately reflected in the entire application.

Listing 1: Basic monkey-patching example. Function `someFunction` of class `MyClass` is monkey-patched in function `performMonkeyPatch`.

```
1  function MyClass() {};
2  MyClass.prototype.someFunction = function () {
3      // Initial implementation
4  };
5
6  function performMonkeyPatch() {
7      var originalFunction = MyClass.prototype.someFunction;
8      MyClass.prototype.someFunction = function () {
9          // New monkey-patched implementation
10         // originalFunction can be invoked here
11     };
12 }
```

2.1.2 Monkey-patching. JavaScript supports dynamic modification of classes and functions at run-time, that is *monkey-patching* functions. Listing 1 shows a basic monkey-patching example. Let function `someFunction` be defined in lines 2 to 4. Function `performMonkeyPatch` defined in lines 6 to 12 monkey-patches function `someFunction`. First, the initial implementation is copied to variable `originalFunction`. Then, the function `someFunction` is overwritten. As long as the `performMonkeyPatch` function was not executed, the call to `someFunction` will invoke the original function. As soon as the monkey-patching statements are executed, the patched function will be invoked instead.

In JavaScript, there is no way to tell whether a function is monkey-patched or not. That means, Node.js applications have to trust their dependencies to not manipulate the original functions.

2.1.3 Loaded modules cache. A module must be explicitly loaded into the application scope using function JavaScript `require` in order to use its functions and variables. The `require` function expects the name of the dependency as a parameter and returns the *exported* functions and properties of the requested module. The `require` function comes with a `cache` property that caches loaded modules and their exported properties[2].

This cache is shared among all modules in a given Node.js application. It is somewhat comparable to global variables since there is exactly one cache instance shared throughout the entire application. Any module can manipulate the content of the application cache.

2.2 Dependency attacks

We identified 4 dependency attacks for the 3 weaknesses discussed above.

Listing 2: Code snippet that leaks environment variables which are stored in `process.env`.

```
1  //definition of the leak method
2  function leak(data) { ... }
3
4  //call of the method
5  leak(process.env);
```

Listing 3: Demonstration of a Global Manipulation attack that alters the functionality of the `validator` dependency.

```
1  > var validator = require('validator');
2  > validator.isEmail('name@example.de"␣hello="world');
3  false
4  > require('./index.js'); //Contains the Manip. attack
5  > validator.isEmail('name@example.de"␣hello="world');
6  true
7
8  //Content of ./index.js
9  var validator = require('validator');
10 var isEmail = validator.isEmail;
11 validator.isEmail = function(val){
12       if (val==='de"␣hello="world') return true;
13       else return isEmail.apply(this, arguments);
14 };
```

2.2.1 Global leakage. A dependency could access global variables and leak their content. To do so, it needs to implement a `leak()`-like function that sends the provided parameter, which could, for example, be the list of environment variables, to an attacker-controlled server. It only takes one line of code to leak the values of all environment variables as shown in listing 2.

For example, in AWS Lambda, the security credentials of the role that is used to execute the given application are provided in environment variables.[2] A leak method in a third-party dependency could leak this sensitive information [15]. Depending on the configuration of the victim, these credentials can be misused to perform the actions that the application is also allowed to perform including take over control of the victim's account.

2.2.2 Global manipulation. The key idea of this attack is to manipulate a globally accessible variable or function to alter the application behavior. For instance, the behaviour of a function could be changed using monkey-patching.

Listing 3 shows an example of manipulation attack. The application loads the `validator` library in line 1. The dependency checks correctly the user input in line 2; the result of this invocation is `false` as shown in line 3. The malicious code located in file `./index.js` (content is in lines 8 to 14) is loaded in line 4, which changes the behavior of the `isEmail()` function. In line 5, the same string is tested again. This time, the invalid Email address is recognized as a valid Email address as indicated by the returned value `true` in line 6.

Note that file `./index.js` is loaded to use functions other than `isEmail()` function. However, this operation overwrites the original function `isEmail()`. This change of the validator library could obviously concern security-sensitive functions such as the functions that validate input against XSS or SQL Injections.

[2]These credentials can be used to "sign programmatic requests that could be made to AWS using AWS SDKs, REST (REST), or Query APIs" [7].

Listing 4: Basic Dependency Tree Manipulation attack example.

```
1  require('./malicious-lib');
2  require.cache[require.resolve('victim-lib')]
3   = require.cache[require.resolve('./malicious-lib')];
```

Listing 5: Dependency Tree Manipulation attack to manipulate a dependency that exports a function instead of an object.

```
1  var original = require('victim-lib');
2  require.cache[require.resolve('victim-lib')].exports
3   = function () {
4      // Monkey-patch function body
5      return original.apply(this, arguments);
6  };
```

2.2.3 Local manipulation. Variables in Node.js could have a local scope that is accessible within the context of the caller, not from outside as in global variables. However, Node.js has the loaded module cache weakness. This property can be misused to manipulate other dependencies.

By exploiting this weakness, the control flow of the given application can be altered in favor of the attacker. Assume that the application and one of its dependencies `A` use a dependency `B`. Dependency `A` can modify the value of an exported variable of dependency `B` to alter the behavior of the main application. It can also monkey-patch an exported method of the dependency to alter the behavior of the main application.

2.2.4 Dependency-tree manipulation. It is possible to use the `cache` object of the `require` function directly rather than manipulating the object returned by the `require` function, as in the previous attack. This attack uses the `require.resolve` function and gives the file name that implements the module to attack as a parameter.

There are two variations of this attack. In the first variation, a malicious dependency is loaded instead of a victim dependency, as demonstrated in listing 4. Any modules of the given application that loads the victim dependency, would use the malicious dependency available in the cache instead of the original one. In the second variation, the function of the victim dependency is called using the `resolve` method. Then, it is modified. The modified method is stored in the dependency file. Listing 5 demonstrates this variation.

This attack could be used to bypass security controls. For example, it is possible to modify the behavior of the `requireSafe` module,[3] which is used to receive notifications about outdated or vulnerable loaded dependencies. Such attack would allow to avoid updating the environment with recent versions of the dependencies.

[3]https://www.npmjs.com/package/requiresafe

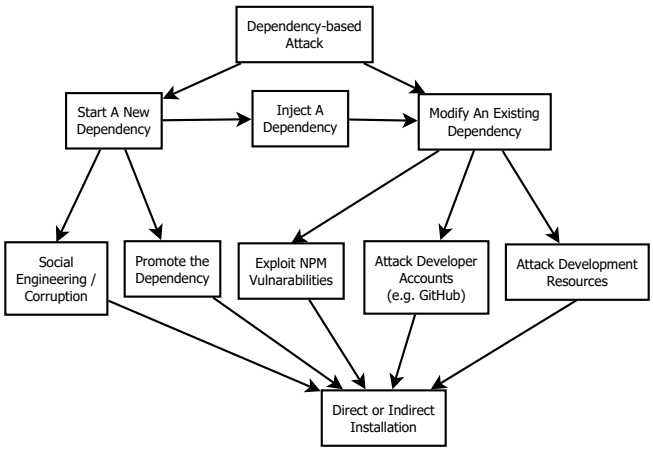

Figure 2: Visualization of options an attacker has to deploy malicious code as part of a third-party dependency using NPM.

2.3 Discussion

The dependencies used in an application can either be bundled by the user/developer and uploaded as a package, or automatically resolved by the environment platform based on the `package.json` file. In either case, the malicious dependency gets downloaded from a public repository, for instance NPM or GitHub.

Figure 2 visualizes how malicious dependency can get to be used. One option is to start a new dependency. The attacker could promote it publicly or through social engineering. They may add malicious behavior at later version of the library to avoid detection by the reviews that some companies do before the first use of the dependencies.

The second option is to manipulate an existing dependency that is already widely used. This could be through (1) exploiting NPM vulnerabilities such as the NPM Package Install Vulnerability [6] or (2) attacking the development resources using e.g., developers' leaked credentials–many popular dependencies contained credentials that allows attackers to publish or update dependencies [8].

A dependency could easily evade attention of testers by detecting the environment in which it is executed. It could invoke the malicious payload only when it identifies that it is executed in a production environment. For instance, it could use the value of the environment variable `NODE_ENV`, which is available in the dependency `express`, to decide to execute (the value of the variable is `production`) or not–if the value of the variable is not `production`.

For a running application, it is very difficult to realize that the application behavior is manipulated using a dependency attack. Therefore, we propose the use of code analysis to verify the absence of such attacks before executing Node.js applications.

3 MITIGATION OF DEPENDENCY-BASED ATTACKS

This section discusses the mitigation strategies for dependency-based attacks and the static code analysis that we implemented to detect these attacks.

3.1 Mitigation strategies

The first mitigation strategy for this category of attacks is to perform code review of all dependencies, which is time consuming. Manual code review is not practical when updated versions of these dependencies should be adopted frequently.

The second mitigation strategy is to address the three weaknesses discussed in Subsection 2.1. However, these weaknesses are design choices. Design changes, such as changing the cache mechanism of Node.js, may be made to address the weaknesses. However, such design changes would break existing applications.

The third strategy, which is more realistic, is to use static code analysis t detect these attacks, which we adopted. Next subsections discuss our implementation.

3.2 Static code analyses for dependency-based attacks

We implemented code analysis that detects the attacks described above using the code analysis tool T.J. Watson Libraries for Analysis (WALA) [5]. WALA is a package of libraries for analysis of JavaScript source code–and Java binary code. WALA is a fast, efficient, and extensible analysis framework. The framework transforms the source code to an Intermediate Representation (IR) (a kind of a simple language) [12] form that we use in the analysis.

A short description of the code analysis to detect the 4 attacks follows.

3.2.1 Global leakage. The identification criterion for this attack is: invoke a leaking function where one of the parameters is a global variable. The problem with this attack is the difficulty to enumerate all the leaking functions–this is easier in the case of Android as Apps need to call specific system services.

We use the implementation of `HTTP` protocol as an example of leakage mechanism. The analysis that we developed assess the use of `HTTP request` object to leak values of global variables. The analysis identifies calls to the `write` method of `HTTP request`. We use data flow analysis [12] to trace whether the parameter of the method is linked to a global variable.

3.2.2 Global manipulation. There are two varieties for this attack: (1) a global object is overwritten or (2) any of its properties is modified. The first variety is easy to detect. In WALA, overwriting a global variable is represented by a single IR `AstGlobalWrite` instruction. If an IR instruction is of type `AstGlobalWrite` and the variable is a built-in JavaScript global variable, than an attack is found. Two criteria needs to be fulfilled for the second variety: a property of a variable is written and that variable is linked to one of the 32

Table 2: Evaluation of the static code analyses in terms of number of findings and required analysis times.

Attack	# cases	Global Leak		Global Man.		Local Man		Dep. Tree Man.	
		#	Time	#	Time	#	Time	#	Time
Global Leak	3	2	133 s	147	138 s	1	5 s	0	0.484 s
Global Man.	9	0	0.063 s	2	0.519 s	0	0.019 s	0	0.027 s
Local Man.	4	0	0.430 s	0	2.611 s	1	0.081 s	0	0.067 s
Dep. Tree Man.	4	0	0.364 s	0	2.676 s	0	0.060 s	1	0.069 s

Note: The analysis times are an average over three executions on a standard computer (Intel Core i5, 2.2 GHz; 8 GB RAM).

JavaScript global variables. The analysis identifies all (IR) `AstGlobalRead` instructions for the 32 variables and propagate them over the module to identify linked variables. Next, IR instructions `JavaScriptPropertyWrite` that manipulate variable from this list are identified.

3.2.3 Local manipulation. The identification criteria of the attack are: the dependency is loaded using function `require` in the scope of a module and manipulated in the scope of another module. To detect such attack, we first identify all invocations of `require` function, considering that function names can be parameters for other functions. Next, a data flow analysis is performed over loaded objects and manipulations of the objects or of their properties are identified; that is, we track IR instructions `JavaScriptPropertWrite` and `SSAPutInstruction`.

3.2.4 Dependency-tree manipulation. The identification criterion of the attack is: the interface of a dependency is manipulated in the scope of another module *without loading* it or the dependency module is replaced by another file. To detect such attack, all invocation of JavaScript statement `require.cache` are identified. The objects returned by these invocations are tracked using data flow analysis to check for manipulations.

3.2.5 Discussion. The analyses are integrated into open-source serverless platform `OpenWhisk` [4]. The integration is designed such that, when a code is provided for execution, `OpenWhisk` creates a Docker container, analyzes the code, and either proceeds with executing the code or declines to execute it if it contains a dependency-attack.

The analyses are published to the public WALA repository on March 12, 2017.

3.3 Evaluation of the solution

We did not find online real-world exploits of dependency-based attacks. Therefore, we developed 20 test cases to verify the analysis. We used in these exploits JavaScript features for obfuscation and hiding malicious actions. Table 2 provides the number of test cases for each type of attack along with the number of findings for each analysis and the analysis times. For each test case (in the columns) we report the number of findings of each of the 4 analyses (in the rows). We observe that the number of global manipulation findings for each global leak case is 147. The analysis returns 146 false positives. The precision issue is due to over approximations. (Improvement can be made to the analysis to address the over approximation.)

We note that we found in file `domain.js`, which is part of Node.js core libraries, at line 29 what we call a local manipulation attack. The `flag EventEmitter` is initialized to `false` in file `event.js`, which is also part of Node.js core libraries. This is a legitimate use of the weakness as it is not for malicious purpose.

The analysis has currently several limitations including the limitation of the size of applications that could be analyzed (Max. 9 MB.), dependencies that contain binary code, and use of specific functions such as the `eval`. Future work will address these problems.

4 RELATED WORK

To the best of our knowledge, all existing work on security and third-party dependencies is focused on vulnerabilities included in dependencies.

For instance, Hejderup [10] investigates the time to fix vulnerabilities in dependencies and the frequency of updating dependencies. They investigated dependency management practices, such as how vulnerabilities in dependencies are resolved and how much time it takes for the dependency developer to publish a fix.

Plate et al. [17] studied the question: Is a given vulnerability that was found in an OSS library and is used in given product, is indeed exploitable in the given product? They developed an automated solution to answer that question for a specific use of a OSS library in a product and a given vulnerability in that library.

Jensen et al. [11] introduce the term Cloud Malware Injection Attack. The idea is that the adversary needs to create a malicious service (SaaS or PaaS) or a virtual machine instance (IaaS) and add it to the Cloud system. Then, they have to trick the Cloud system so that it considers the service they created as a valid instance for the particular service they want to attack.

Tellnes argues [18] that allowing dependencies to run in an application allows for exploitation of the vulnerabilities they contain because these dependencies have the same access levels as the applications that use them. They propose to address the problem at the package manager infrastructure. As a second countermeasure they suggest to implement security wrappers to isolate dependencies from one another and from the host application.

5 CONCLUSION

By design, Node.js and JavaScript support global variables, monkey-patching, and loaded modules cache. These features could be used to perform 4 dependency-based attacks: leakage of global variables, manipulation of global variables, manipulation of local variables, and manipulation of the dependency tree. Currently, developers are required to review the dependencies they use for malicious behavior–a difficult task given the number of dependencies that one needs to use in a given application. As an alternative solution, we developed static code analyses using T.J. Watson Libraries for Analysis (WALA) to identify the 4 attacks. The performance of the analysis is satisfactory except for global leakage attack, which takes more than 2 min. The analyses are integrated to OpenWhisk, an open source serverless computing platform. Before accepting an uploaded NodeJS applications, the extended OpenWhisk platform triggers the analyses. The application is denied execution if an attack is identified.

ACKNOWLEDGMENTS

We thank Monika Illgner-Kurz (IBM DE) and Julian Dolby (IBM US) for assistance and directions.

REFERENCES

[1] [n. d.]. Node.js. https://nodejs.org/en/. ([n. d.]). Online; accessed on May 2017.
[2] [n. d.]. Node.js v4.8.3 Documentation. https://nodejs.org/dist/latest-v4.x/docs/api/modules.html#modules_caching. ([n. d.]). Online;accessed on May 2017.
[3] [n. d.]. npm. https://www.npmjs.com/browse/keyword/repository. ([n. d.]). Online; accessed May, 2017.
[4] [n. d.]. OpenWhisk. https://developer.ibm.com/openwhisk. ([n. d.]). Online; accessed on May 2017.
[5] [n. d.]. T.J. Watson Libraries for Analysis. http://wala.sourceforge.net/wiki/index.php/Main_Page. ([n. d.]). accessed in May 2017.
[6] 2016. Package install scripts vulnerability. http://blog.npmjs.org/post/141702881055/package-install-scripts-vulnerability. (march 2016). Online; accessed on May 2017.
[7] Amazon Web Services, Inc 2016. *Amazon Web Services: General Reference* (Version 1.0 ed.). Amazon Web Services, Inc, http://docs.aws.amazon.com/general/latest/gr/aws-general.pdf.
[8] "ChALkeR". [n. d.]. Do not underestimate credentials leaks. https://github.com/ChALkeR/notes/blob/master/Do-not-underestimate-credentials-leaks.md. ([n. d.]). Online; accessed on May 2017.
[9] B. Chess and J. West. 2007. *Secure Programming with Static Analysis* (first ed.). Addison-Wesley Professional.
[10] J. Hejderup. 2015. *In Dependencies We Trust: How vulnerable are dependencies in software modules?* Master's thesis. Delft University of Technology, Delft, the Netherlands.
[11] M. Jensen, J. Schwenk, N. Gruschka, and L. Iacono. 2009. On Technical Security Issues in Cloud Computing. In *Proc. of the 2009 IEEE International Conference on Cloud Computing (CLOUD '09)*. Bangalore, India, 109–116.
[12] U. Khedker, A. Sanyal, and B. Karkare. 2009. *Data Flow Analysis: Theory and Practice* (1st ed.). CRC Press, Inc., Boca Raton, FL, USA.
[13] P. Krill. 2016. The battle for Node.js security has only begun. http://www.infoworld.com/article/3029218/javascript/battle-for-nodejs-security-has-only-begun.html. (Feb. 2016).
[14] R. Parkhe. 2013. Global Variables Are Bad. http://c2.com/cgi/wiki?GlobalVariablesAreBad. (2013). Online; accessed in June 2016.
[15] B. Pfretzschner. 2016. *Detection of dependency-based attacks on NodeJS envirnment*. Master's thesis. TU Darmstadt, Darmstadt, Germany.
[16] B. Pfretzschner and L. b. Othmane. 2016. Dependency-Based Attacks on Node.js. In *2016 IEEE Cybersecurity Development (SecDev)*. 66–66. https://doi.org/10.1109/SecDev.2016.023
[17] H. Plate, S. E. Ponta, and A. Sabetta. 2015. Impact assessment for vulnerabilities in open-source software libraries. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 411–420. https://doi.org/10.1109/ICSM.2015.7332492
[18] J. Tellnes. 2013. *Dependencies: No Software is an Island*. Master's thesis. University of Bergen, Bergen, the Netherlands.