You Can Run But You Can’t Hide: Runtime Protection Against Malicious Package Updates For Node.js

Timo Pohl¹, Marc Ohm², Felix Boes³, Michael Meier⁴

Abstract: Malicious software packages are often used in software supply chain attacks. Detecting these packages is a top priority, and there have been many academic and commercial approaches developed for this purpose. In the event of an attack, it is essential to have resilience against malicious code. To address this issue, we introduce a runtime protection for Node.js that automatically limits the capabilities of packages to a minimum level. The implementation and evaluation of the detection and enforcement of necessary capabilities at runtime was conducted against known malicious attacks. Our approach successfully prevented 90% of historical attacks with a median install-time overhead of less than 0.6 seconds and a median runtime overhead of less than 0.2 seconds.

Keywords: Software Supply Chain; Policy Enforcement; Abstract Syntax Trees

1 Introduction

Modern software relies on reusing software components, due in large part to the abundance of Free and Open Source Software (FOSS). Utilizing pre-built building blocks has advantages, yet introduces security risks. Each added component increases the dependence on unverified code from untrusted developers. Thus, it is not surprising that a noticeable trend for software supply chain attacks involves maliciously manipulated software packages, as reported by Ohm et al. [Oh20]. Although one may carefully select a software, the use of dependencies remains nontransparent to the user. A common user will not realize should an attacker have introduced malicious code to a certain point in the supply chain.

To counteract such supply chain attacks, we leverage two observations. Firstly, the fact that a prominent attack vector is the introduction of malicious code at the patch level of a package, which is automatically updated on the victim’s system. Moreover, it has been observed that these additions noticeably alter the functionality of the respective software component [OSM20]. Our approach is predicated on the continuity of benign software and

¹ University of Bonn, Institute for Computer Science IV, Friedrich-Hirzebruch-Allee 8, 53115 Bonn, Germany, pohl@cs.uni-bonn.de
² University of Bonn, Institute for Computer Science IV, Friedrich-Hirzebruch-Allee 8, 53115 Bonn, Germany & Fraunhofer FKIE, 53115 Bonn, Germany, ohm@cs.uni-bonn.de, marc.ohm@fkie.fraunhofer.de
³ University of Bonn, Institute for Computer Science IV, Friedrich-Hirzebruch-Allee 8, 53115 Bonn, Germany, boes@cs.uni-bonn.de
⁴ University of Bonn, Institute for Computer Science IV, Friedrich-Hirzebruch-Allee 8, 53115 Bonn, Germany & Fraunhofer FKIE, 53115 Bonn, Germany, mm@cs.uni-bonn.de, michael.meier@fkie.fraunhofer.de

doi: 10.18420/sicherheit2024_015
232 Timo Pohl, Marc Ohm, Felix Boes, Michael Meier

aims to prevent the execution of intentionally added malicious code, which often demands extra capabilities. To address the challenge of employing dependencies with unknown capabilities, we introduce an automated approach.

Our approach automatically deduces necessary functionalities such as access to particular modules or functions by thoroughly analyzing the source code of a package in conjunction with that of its dependencies. In the initial stage, our approach infers a range of capabilities grounded in an authorized version of a package. Once the package is updated to a more recent version, it is executed using our customized Node.js interpreter, which enforces the predetermined capabilities during runtime. The newly added functionalities are not accessible in this manner, resulting in unsuccessful execution of the related code.

The rest of this paper is organized as follows: Sect. 2 presents and discusses related research. In Sect. 3, we describe our methodology and use case, while Sect. 4 illustrates our corresponding implementation. We then present the results from our experiments in Sect. 5. Finally, we draw a conclusion and provide perspectives for future work in Sect. 6.

2 Related Work

With increasing amounts of attacks on the software supply chain in recent years, research in that field has been thriving [OS23]. Several works focused on providing security by identifying malicious software packages in package repositories. This has been done both through code analysis [Li21, SS22, Oh22] and metadata analysis [Za21, Go21]. In addition, Taylor et al. have built a tool that protects against typosquatting attacks [Ta20]. Further research has specialized this task to not just detect malicious packages in general, but instead aiming to detect malicious updates to previously benign packages, leveraging different kinds of anomaly detection [OSM20, Du19, Ga19].

However, even if the mentioned detectors were widely deployed, they would inevitably miss some malicious packages. To account for this fact, there have also been approaches to allow the safe execution of malicious code.

Koishybeyev and Kapravelos [KK20] created a tool that reduces the attack surface of Node.js applications by removing unused code segments and blocking the ability to access built-in modules that are not statically referenced. While this protects against attacks where dependencies abuse certain vulnerabilities in adjacent dependencies, it does not provide protection against malicious packages in general.

Vasilakis et al. [Va18] built an alternative module system for Node.js, which allows spawning instances of modules in configurable “compartments” providing different levels of isolation. Only the compartment with the strongest isolation provides protection against malicious packages, but in turn requires extensive configuration for each module and introduces significant runtime overhead.
In 2022, Wyss et al. [Wy22] have developed a way to protect users against install-time attacks. Their system *LATCH* leverages a sandboxed cloud install to create a manifest of so-called “intents”, representing different capabilities. These intents are compared to a user-defined policy of allowed intents. If they do not match, the installation can either get blocked altogether, or the runtime is moved to an AppArmor environment, enforcing adherence to the policy. The evaluation suggests that this approach protects reasonably well against install-time attacks, but still suffers from overblocking, and does not protect against runtime attacks. Furthermore, it puts the burden of creating a meaningfully secure policy on the user.

For protection against runtime attacks, Ferreira et al. [Fe21] have created a permission system with the intent of allowing individual packages to have individual permissions, which can be applied to packages of the Node.js ecosystem. Package developers have to declare the permissions that their package needs, and the user has to accept the permissions given to a package. At runtime, these permissions are enforced by restricting access to built-in JavaScript modules, as well as certain global objects. Results are promising that capability-based approaches are able to reduce attack surface and capable of protecting against real-world attacks.

The presented approaches rely on the user to create or verify the capability choice. While they might be familiar with such a permission system from platforms like android, relying on the user to choose or verify permissions for a certain package bears risks. Current research suggests that users do not pay attention to requests for certain permissions, and if they do, they do not fully understand the implications [Fe12]. For these reasons, we present a similar approach, leveraging more granular permissions and a system to automatically infer and enforce policies.

## 3 Methodology

We have chosen JavaScript and the Node.js runtime as our ecosystem, as it has been a common ecosystem choice for related work [SS22, Oh22, Za21, Ga19, Wy22, Fe21] and shows the highest number of known malicious packages [Oh20]. It thus meets our criteria of comparability to other research in the same area, as well as practical relevance. We focus on the prevention of execution of malicious code despite how it was added to the targeted software. However, we assume that the malicious code is contained in a future update of the software which the user is currently running.

We develop an approach to automatically infer capabilities used by a program. In this context, capabilities refer to accessed global objects and built-in modules. The inferred capabilities need to be persisted to a policy file which will be used for the enforcement later on. To do so, we need to locate and understand the import functionality of the Node.js interpreter. Furthermore, that functionality needs to be enhanced in a way that allows it to
respect our generated policy. The actual selections and details of the implementation will be presented in Sect. 4. We evaluate our approach to answer the following research questions:

**RQ1** How exhaustive is the automated inference of capabilities?

**RQ2** To what extent are benign package updates affected?

**RQ3** Which historic attacks would have been blocked?

In order to answer these questions, we conduct several experiments. All experiments are run as “no-human-in-the-loop” style, i.e., all policies are generated and enforced fully automated. Nonetheless, our experiments provide an upper boundary for the approach’s shortcomings and a lower boundary for its inference performance.

For RQ1 we use the 200,000 randomly chosen packages again, and compare the dependencies they use to the third party modules our approach is able to infer.

As a reference of dependencies a package uses, we consult the list of runtime dependencies in the package’s `package.json` manifest. Since the list of dependencies has to be manually maintained by the developer, it is prone to errors. For example, these lists can contain development dependencies incorrectly declared as runtime dependencies, or deprecated dependencies from previous versions that have not been removed from the dependency list. For this reason, we only consider those dependencies whose names also appear within at least one JavaScript file of the package. Additionally, we filter out all dependencies starting with `@types`, as those are just the type definitions needed to transpile code from TypeScript to JavaScript, and thus won’t appear in JavaScript files.

The change of required capabilities caused by historic updates is recorded for RQ2. We conduct the experiment on the 1,000 most depended upon software packages from npm because these are most likely to be benign. This allows us to provide an upper bound of false positives and true negatives of our approach.

To answer RQ3 we generate and enforce the policy on the last benign versions of a set of known malicious packages taken from the Backstabber’s Knife Collection [Oh20], and subsequently trigger the malicious behavior of the following malicious version in a sandboxed environment. This allows us to determine whether our approach would have prevented these attacks.

We also conducted all experiments for a more fine-grained definition of capabilities, namely for individual members of global objects and modules. However, we observed that using the fine granular policy does not provide any benefit in stopping the given attacks, as none of them used the required modules or global objects in their benign versions. Other performance metrics were very similar or a bit worse than for the coarse grained capabilities. For the sake of brevity, we thus only present the results for the coarse grained capabilities.
The source code of our approach as well as the software packages’ names used for the experiments are available on GitHub\(^5\).

## 4 Implementation

After explaining the general concept of our approach, this chapter presents our reference implementation. We detail the implementation of the policy generation and show how the generated policy is enforced at runtime.

Our policy is a mapping of a package name to its capabilities. As described by Ferreira et al., Node.js has very limited functionality without importing any of its built-in modules [Fe21]. Additionally, the modules are inherently grouping abilities that belong together. For example, the `fs` module allows access to different kinds of file system operations, like reading or writing files. Therefore, we are selecting the built-in modules as one part of the capabilities.

Furthermore, JavaScript and the Node.js interpreter expose a set of global objects which also allow access to certain abilities. For example, the `Buffer` global object allows manipulation of byte buffers, and allows performing various en- and decoding operations. As we are aiming to provide a more fine granular approach than Ferreira et al., we are also considering global objects to be part of the capabilities. Thus, the set of capabilities we are choosing for our implementation is the union of the set of built-in modules and the set of global objects.

To identify a package’s capabilities, we generate an Abstract Syntax Tree (AST) for each file within the package with one of the JavaScript file extensions `.mjs`, `.cjs` or `.js`, using `acorn` [HnLHS]. Each file that `acorn` fails to parse is considered invalid JavaScript and is thus ignored. Given the AST we extract the used modules and global objects. Imported modules are identified as the arguments to calls to either the `require` or `import` function or the `import` statement. After extracting the capabilities for every JavaScript file of a package, the package’s capabilities are set to the union of all those files’ capabilities. Since Node.js is natively able to parse JSON, the policy is stored in the JSON format. The capability inference is visualized in Fig. 1.

The generated policy is enforced through a patch in the Node.js runtime. This enforcement is split into two parts. The first part is the enforcement of the module restrictions. This is done by patching the `makeRequire` function in the file `lib/internal/modules/cjs/helpers.js`, which is responsible for providing the `require` function to every loaded module.\(^6\)

We replace every module that is not contained in the requiring packages’ allowlist with a dummy object containing the same members as the required module, but where the corresponding value is a function that returns itself when it’s called. This way, a package that is not allowed to access a certain module can still require it and call its member functions without crashing, but with no risk of performing malicious actions through that module.

\(^5\) [https://github.com/cybertier/npm-dependency-guardian](https://github.com/cybertier/npm-dependency-guardian)

\(^6\) Policy enforcement for ECMAScript modules is currently not supported.
The second part is the enforcement of global object restrictions. To do this, we inject a new local object into each module, which holds a reference to all global objects that are present in the allowlist. Similar to the module enforcement, references to objects that are not part of the policy will be replaced with dummy objects.

Additionally, when the file is loaded, we alter the source code, replacing each reference to a global object with a reference to the corresponding member of our new injected local object. This way, we can ensure that only those global objects present in the allowlist can actually be accessed.

5 Evaluation

In this section, we first briefly discuss the performance of our approach. We then present the results of our experiments, which are described in Sect. 3.

When considering performance overhead, there are two areas to address. First, the capability inference before the runtime and second, the policy enforcement during the runtime. The capability inference time is measured on 200,000 randomly chosen packages from npm. On median our approach requires 0.52 (\(\mu = 2.73\), \(\sigma = 6.08\)) seconds to generate a policy. The actual enforcement of the policy requires an additional 0.2 s at startup time. Overall, we consider this a small performance overhead.

The rest of this section describes the results of our experiments regarding our three research questions.
5.1 Exhaustiveness (RQ1)

To analyze the exhaustiveness, i.e., does our approach comprehensively infer the set of required capabilities, we performed the experiment as described in Sect. 3. Recall that we compare the set of the detected external modules to the dependencies used by the package.

Again, we conducted the experiment on the 200,000 randomly chosen packages from npm. However, 3,311 packages had corrupt meta information and 12 crashed during AST generation. An additional 17,706 packages had invalid JavaScript files, for example incorrectly named `.jsx` files. From the remaining packages we collected 541,538 dependencies in total. Our tool correctly detected 499,587 (92.25 %) dependencies. Of the 109,198 packages with at least one dependency, we correctly inferred all dependencies for 91,992 (84.24 %) packages.

As mentioned in Sect. 3, the validity of these results depends on the correct declaration of dependencies. Even our naive verification process of looking for the dependency name within the package’s JavaScript files is not guaranteed to only result in dependencies that are actually used. Therefore, we manually inspected 100 randomly selected packages where at least one dependency was counted as undetected by our automated approach, to estimate the quality of this dataset. We found that 80 of those packages did not actually use the declared runtime dependency, and the text matches were usually found as variable names or within comments. In the remaining 20 cases we were not able to confirm that the respective module is imported, but from the given occurrence we could also not confidently deny that the module may be imported through some indirections.

This highlights that even with our additional verification measures, there are a lot of dependencies that are not actually used. However, it is unlikely that there are dependencies that are used but not declared in the `package.json` file, as this would in most cases break the package for all users. Therefore, we consider our findings as lower bounds for the amount of correctly found modules.

**Response to RQ1:** Our approach correctly detects 92 % of used imports, and for 84 % of packages all imports are correctly detected.

5.2 Specificity (RQ2)

Being able to automatically infer, describe, and enforce capabilities for a software, we will now investigate its performance on benign software updates. We investigated how the set of capabilities per package changes across historic updates.

We conducted this experiment for the 1,000 most depended upon packages from npm according to Sect. 3, assuming that all currently published versions of these packages are
benign. Furthermore, we distinguish between major, minor, and patch updates according to Semantic Versioning [PW13] as well as if the updates required new modules, new global objects, or both, in order to get more granular insights.

<table>
<thead>
<tr>
<th></th>
<th>New module and global object</th>
<th>New global object</th>
<th>New module</th>
<th>No change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Major</td>
<td>39.4%</td>
<td>22.4%</td>
<td>0.7%</td>
<td>37.5%</td>
</tr>
<tr>
<td>Minor</td>
<td>16.7%</td>
<td>21.9%</td>
<td>4.0%</td>
<td>59.5%</td>
</tr>
<tr>
<td>Patch</td>
<td>8.7%</td>
<td>21.9%</td>
<td>85.3%</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 2: Changes of capabilities required without member access tracing, sorted by version level.

As depicted in Fig. 2, 37.5 % of major updates, 59.8 % of minor updates, and 85.3 % of patch updates did not require new capabilities. This drastic reduction of newly required capabilities is in line with the procedure proposed by Semantic Versioning [PW13].

Arguably, major updates should not be performed fully automatically as it is expected to be not backwards compatible and hence might break things. Most often, patch updates do not require new capabilities. However, as observed by Ohm et al. [OSM20], malicious code is frequently introduced on patch level — a commonly allowed level for automatic updates — and alters the program’s functionality. Our approach would impede such an attack.

**Response to RQ2:** Applying our approach to patch updates still results in 14.7 % false positives. This motivates further improvements.

5.3 Sensitivity (RQ3)

Arguably, the most important question is RQ3, whether the presented approach protects against attacks, i.e., malicious updates. We are considering previously benign software packages that turn malicious. This may be achieved by adding malicious code to the package itself, adding malicious code to one of its dependencies, or by adding a new dependency containing malicious code [La23].

As there exists no method of formally proving that we will be able to prevent all future attacks, we evaluate whether we would have prevented past attacks on benign packages. To do so, we need samples of malicious packages that were infected in one of the presented manners. The Backstabber’s Knife Collection [Oh20] is a curated collection of such samples. It also has a well maintained package index, which allows querying individual packages by several metadata attributes. For these reasons, we use the Backstabber’s Knife Collection7 to gather samples for our evaluation.

---

7 We are using the commit 22bd768, which was the most up-to-date commit at the time of our evaluation.
Packages appropriate for our attack model have to meet the following criteria: (1) Published in the npm registry, (2) Infected an existing package, (3) Malicious action at runtime, (4) Applicable to Node.js. Using the package index metadata mentioned above, we can create a query that preselects only packages that meet the first three criteria. However, not all of them are applicable to Node.js, as some of them leverage browser exclusive JavaScript APIs to perform their malicious actions. This characteristic is not reflected in the dataset’s metadata and hence we removed such packages by hand. These remaining packages as well as the malicious version and the last benign version for reference are listed in Tab. 1.

Tab. 1: List of package versions used for our experiment, if the attack was averted, and if the program flow crashed because of our interference.

<table>
<thead>
<tr>
<th>Package</th>
<th>Last benign version</th>
<th>Malicious version</th>
<th>Averted</th>
<th>Crashed</th>
</tr>
</thead>
<tbody>
<tr>
<td>conventional-changelog</td>
<td>1.1.24</td>
<td>1.2.0</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>eslint-config-eslint</td>
<td>5.0.1</td>
<td>5.0.2</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>eslint-scope</td>
<td>3.7.1</td>
<td>3.7.2</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>event-stream</td>
<td>3.3.5</td>
<td>3.3.6</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>kraken-api</td>
<td>0.1.7</td>
<td>0.1.8</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>leetlog</td>
<td>0.1.1</td>
<td>0.1.2</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>load-from-cwd-or-npm</td>
<td>3.0.1</td>
<td>3.0.2</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>mariadb</td>
<td>2.5.6</td>
<td>2.13.0</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>opencv.js</td>
<td>1.0.0</td>
<td>1.0.1</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>rate-map</td>
<td>1.0.2</td>
<td>1.0.3</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

To evaluate whether our approach would have prevented an attack, we create a policy for the respective preceding version of the package and afterwards trigger the malicious behavior of the malicious version in a sandboxed environment. We examined each package manually and confirmed that the blocked capabilities were necessary to execute the malicious functionality.

The proposed approach averts 9/10 of the investigated attacks. Regarding the package load-from-cwd-or-npm, we were unable to trigger the malicious behavior. An analysis of the code suggests that our approach would have averted the attack, but still crash the program. Furthermore, we see that our naive implementation of the mock object, which is returned if a prohibited capability is requested, turned out to be insufficient in some cases, leading to a crash.

**Response to RQ3:** Our approach would have stopped 9 out of 10 known attacks.

### 5.4 Comparison to Related Work

Our approach stands in direct comparison with Wyss et al. [Wy22] and Ferreira et al. [Fe21]. While Wyss et al. focus on install-time attacks, we focus on runtime protection. In contrast to Ferreira et al. who leverage a manually defined and coarse capability set consisting
of network access, file system access, process creation, and all, we take a more granular approach based on automatically inferred policies.

The policy inference for 90 % of packages takes less than one minute for Wyss et al. while our approach takes only 7.30 seconds. Ferreira et al. measure an overhead of less than 1 % during runtime, Wyss et al. can enforce their policy in less than a second for 99 % of the packages, and our approach adds 198 ms to the program’s start. Wyss et al. block 1.5 % of all packages, Ferreira et al. did not analyze the performance on benign packages, while our approach revealed that only 14.7 % of patch update would require an update of the policy. Ferreira et al. replayed three attacks and were able to avert all of them, Wyss et al. analyze 102 samples and were able to detect all of them while we executed attacks from 10 known malicious packages and were able to avert 9 of them.

6 Conclusion & Future Work

Software supply chain attacks that utilize maliciously manipulated software modules are on the rise. A good portion of these inject malicious code directly into a software or more obfuscated in its dependencies. This code is eventually executed during runtime at the end user’s machine.

In this paper we present and evaluate a system that automatically infers and enforces capabilities for software based on the principle of least privilege. By doing so, we prevent the execution of untrusted code.

For our approach, a capability is understood as access to a certain module of a software. Through static analysis, we determine the minimal set of capabilities that are required for a software to run. This information is persisted to a policy file in form of an allowlist. That policy is used by our approach to restrict the access to capabilities.

We conduct several experiments on software packages available from the npm package repository for Node.js/JavaScript to validate our approach. It leverages Abstract Syntax Trees and detects 92 % of a program’s capabilities. However, it still causes too many false positives and requires further engineering. Still, our research indicates the general viability of the approach. Moreover, we would have been able to prevent at least 9 out of 10 historic attacks. Once a policy is generated it can be used for any future version of a software. Benign updates tend to be unaffected by this rigorous allowlist, but malicious updates often required an extended set of capabilities and hence failed to execute.

While we successfully demonstrated our concept and a corresponding proof of concept, there is still room for improvements. For example, we found that not all modules are equally useful to perform an attack. Some modules allow the execution of new processes, which in turn would be able to run without any restrictions by our approach. Furthermore, we have only implemented restrictions for the standard approaches of importing code. However, in
its dynamic nature, JavaScript possesses a number of other ways to import code, or even import code from compiled C++ binaries.

For future work we would like to further improve our approach and transfer it to other languages such as Python. Furthermore, the weighting of capabilities, for instance eval and child_process may more easily be misused, or some fuzzy logic for “very unusual imports” may yield better results.

Bibliography


[HnLHS] Heine né Lang, Adrian; Haverbeke, Marijn; Stepamily, Ingvar; acorn. [Last accessed on 2022-10-18].


[Za21] Zahan, Nusrat; Williams, Laurie Ann; Zimmermann, Thomas; Godefroid, Patrice; Murphy, Brendan; Maddila, Chandra Shekhar: What are Weak Links in the npm Supply Chain? ArXiv, abs/2112.10165, 2021.