2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)

2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) | 978-1-6654-0296-5/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICSE43902.2021.00121

Containing Malicious Package Updates in *npm* with a Lightweight Permission System

Gabriel Ferreira, Limin Jia, Joshua Sunshine, Christian Kästner
Carnegie Mellon University

***Abstract*—The large amount of third-party packages available in fast-moving software ecosystems, such as Node.js/npm, enables attackers to compromise applications by pushing malicious updates to their package dependencies. Studying the npm repository, we observed that many packages in the npm repository that are used in Node.js applications perform only *simple* computations and do not need access to filesystem or network APIs. This offers the opportunity to enforce least-privilege design per package, protecting applications and package dependencies from malicious updates. We propose a lightweight permission system that protects Node.js applications by enforcing package permissions at runtime. We discuss the design space of solutions and show that our system makes a large number of packages much harder to be exploited, almost for free.**

***Index Terms*—security, malicious package updates, supply-chain security, package management, permission system, sandboxing, design trade-offs**

I. Introduction

Modern software applications are commonly built on top of many reusable packages that are constantly evolving [11, 22, 49], which raises a risk of supply-chain attacks through *malicious packages updates*. Such kind of attacks target applications or its users, but are performed through updates in applications' package dependencies, which are downloaded into an application automatically or manually by unsuspecting developers.

The risk from malicious package updates, beyond transport security [12, 54, 65], has long been ignored or seen as a theoretical possibility only [33, 64]. However, recently, more and more cases of malicious package updates have been discovered in multiple large open-source repositories [5, 6, 14, 23, 27, 69]. Attackers keep finding ways to obtain control of developer accounts (e.g., using leaked credentials, targeting weak passwords, or offering help to maintain a package). When in control of an account, attackers can publish a modified malicious version of the package, which is then downloaded (often automatically) by applications depending on this package. Figure 1 shows an excerpt of a real attack.

While malicious package updates are a potential problem in all software projects with external dependencies, we will argue that common practices and design decisions in the *Node.js/npm* ecosystem make such JavaScript applications a particularly attractive target for malicious package updates. Among others (see Sec. II for details), they tend to depend on many small external libraries, they tend to allow automatic updates of minor updates, and the runtime gives all packages the same application-level privileges. When faced with numerous updates from many direct and indirect dependencies, *Node.js/npm* developers often enable automated updates, despite potential security risks.

```
var https = require("https");
https.get({ hostname: "pastebin.com", path: "/evil" },
  r => { r.on("data", c => { eval(c); }); }
).on("error", () => {});
```

(a) After a malicious update, the package now downloads and executes the script below.

```
var fs = require("fs");
var npmrc = require("path").join(...,".npmrc");
if (fs.existsSync(npmrc)) {
  var content = fs.readFileSync(npmrc, "utf8")
  var https = require("https");
  https.get({ hostname: "evil.com", method: "GET",
    headers: {Referer: "http://1.a/"+content}},()=>{}
  ).on("error",()=>{});
}
```

(b) Downloaded malicious script reads and leaks *npm* package manager credentials.

Fig. 1: Essence of the **eslint-scope@3.7.2** attack.

Many defenses against supply-chain attacks have been developed [12, 18, 25, 31, 51, 54–56, 65, 66, 68], but they tend not to be practical in many *realistic software engineering settings*. Defenses include carefully reviewing all dependencies and dependency updates [25], hardening the package infrastructure (e.g., transport security, two-factor authentication) [12, 54, 56, 65], and various forms of program analysis and anomaly detection [18, 31, 51, 55, 66, 68]. However, as we will discuss, in a practical software engineering perspective current approaches either (a) are too expensive for practical use, (b) require a complete redesign of the Node.js module system or runtime environment that is unlikely to see adoption in practice, or (c) only defend against already known vulnerabilities.

In this paper, we design a lightweight permission system and a corresponding enforcement mechanism that protects applications against malicious updates from a large number of packages in direct and indirect package dependencies. Our solution is partial, in that it only defends against attacks of a subset of packages, but it is explicitly designed to be *easy to adopt* and has *negligible runtime overhead*, making it an important and practical building block in defending against malicious package updates.

We build on the insight that *many Node.js packages perform simple computations and do not need access to security-relevant resources, such as the filesystem or the network APIs or metaprogramming constructs.* Our solution effectively sandboxes the large number of simple third-party packages in the *Node.js/npm* ecosystem that do not require access to

978-1-6654-0296-5/21/$31.00 ©2021 IEEE
DOI 10.1109/ICSE43902.2021.00121

security-critical resources, making malicious updates attacks that attempt to elevate packages' privileges ineffective.

The novelty of our permission system lies in the *design* of a practical and lightweight solution that focuses on providing useful and easy to adopt, albeit partial protections. Where existing sandboxing solutions require invasive changes to infrastructure or package implementations, or impose severe runtime overhead [35, 73, 74], ours integrates with the current *Node.js* infrastructure without changes to the implementation of existing packages and imposes negligible runtime overhead. Even though we cannot protect all packages, taking a software engineer's system perspective in the fast paced world of open-source software ecosystems, we argue that even a 10 percent reduction in attack surface that can actually be enacted broadly would result in significant saving of community resources for security reviews and would make it harder for attackers to find packages that they can exploit.

Our evaluation shows that 31.9% of all *npm* packages can be protected by our design and that 52 percent of one year's package updates in 120 popular npm packages and applications are for those protected packages. In addition, our implementation's average performance overhead is negligible ($\ll 1\%$).

Overall, we make the following contributions: (1) we design a lightweight permission system that protects Node.js applications against malicious package updates for a significant number of packages, (2) we discuss design trade-offs to highlight how the chosen partial but low-cost solution fits into a larger security strategy, (3) we evaluate the solution on a large number of packages and applications, and (4) we make both the implementation and evaluation benchmarks available (`https://github.com/gabrielcsf/malicious-updates-icse2021`).

II. The Problem: Malicious Updates, npm, and Current Defenses

We focus on malicious package updates in the Node.js/npm ecosystem, which is the largest, most popular, and fastest growing open-source ecosystem with over one million reusable packages available to download. Several actual attacks were found recently (discussed below), emphasizing the importance of the problem.

Node.js/npm: To explain the problem and our solution, it is important to understand how packages and updates work in Node.js/npm. Node.js is a runtime system that provides powerful APIs to interact with the host system (files, network, processes), which enable programmers to write applications beyond JavaScript's traditional use in a browser. While early applications were heavily biased toward backend web servers, Node.js is also popular for command-line, desktop, robotics, and IoT applications.

Node.js provides its own module system, where each JavaScript file is loaded as a module. Once loaded, modules are represented as JavaScript objects. Node.js projects are structured into modules, which are grouped into named packages or applications. Core APIs are offered through a small set of *native modules*, but developers routinely import a large number of additional modules from third-party packages. Besides JavaScript files, a package contains a manifest file that lists package dependencies required for it to work properly.

Node.js is tightly integrated with *npm*, a package manager and a repository for Node.js packages. The package manager *npm* provides convenient mechanisms to download, install, and update packages and their recursive dependencies from the *npm* repository. In a typical applications's (or package's) installation process, the package manager interprets the content of the manifest file, resolves packages versions, and downloads the source code of direct and indirect packages listed as dependencies.

The design of the *npm* package manager encourages automatic updates and favors ease of publishing packages [11]. Package dependencies can be pinned down to specific versions or defined as version ranges [20, 26, 62]; the use of ranges to automatically install minor updates is very common [15, 26, 40, 82].

Node.js/npm's characteristics facilitate malicious update attacks: Attacks through malicious package updates are possible in most software ecosystems, though certain characteristics make Node.js/npm a particularly attractive target:

- The JavaScript language and Node.js platform provide only a small set of native modules and essentially **no standard library**, leaving it up to the community to develop packages even for standard tasks such as string manipulation and collections. Hence, developers often depend on many third-party packages, even for simple functionality, contributing to a *large attack surface*.[1]
- The Node.js/npm community prefers a model of **many small packages** (inspired by the Unix philosophy) [1]. Thus, it is common to depend on a large number of packages, where each of those packages contributes to a *large attack surface*.
- Developers commonly provide version ranges on dependencies, such that patch-level **updates are automatically installed**, depending on version labels set by the package maintainer [40]. The practice of installing updates automatically in development, test, and sometimes even production systems, contributes to making applications *easy to exploit*.
- The Node.js/npm community values ease of publishing, where updates can be published with a single command-line instruction (typically with locally stored credentials) without further quality checks or reviews [11]. Due to a constant stream of updates, **developers update frequently**, to avoid having to update many packages across many versions at once [11]. This also makes applications *easy to exploit*.
- Most packages also have dependencies of their own, so adding a single package dependency often comes with **many indirect package dependencies** that are **de-facto invisible** to developers. Hence, indirect dependencies are an attractive target for attackers, making applications *easy to exploit*.
- Node.js applications are typically deployed as single-threaded applications, in which **all loaded packages inherit the applications' privileges to use security-relevant resources**

[1] Informally, we consider the number of accounts that can update any of an application's dependencies as the attack surface; the more accounts involved, the higher the chance that any one of them may be compromised.

from accessing local files and the network, to modifying global objects and other packages, to generating code at runtime [63]. As a consequence, loaded malicious packages have a *high potential for damage*.

We exemplify the ease of exploit and the potential damage with three recent attacks detected in the last three years:

- In 2019, the *npm, Inc.* security team identified and reported a malicious version of the *electron-native-notify* package [6]. The attacker published the package with useful functionality and waited until it was added as a dependency of the *Agama* wallet application before publishing a malicious update. The attacker stole about $13 million dollars in bitcoin tokens.
- Also in 2019, the popular *event-stream* package was updated maliciously to steal bitcoins [69]. The malicious update was discovered only after 2.5 month and 8 million downloads. The original maintainer of the *event-stream* package had handed over the account to the attacker when he offered to help maintain the project (i.e., social engineering).
- In 2018, the *eslint-scope* package, part of a widely used JavaScript linter, was also a target of a malicious update. The attack aimed at stealing the *npm* package manager credentials from users of the linter and affected around 4500 accounts (see Figure 1).

State-of-art defenses: In current practice in Node.js/npm, a number of strategies can lower the risk from malicious package updates, though all have severe limitations:

- *Inspection:* Node.js developers are unlikely to carefully audit the large number of direct and indirect dependencies and their updates. Developers typically hope that the community at large will find and report vulnerabilities quickly, but past attacks remained undetected for months or caused significant damage within short periods. Current static analysis and anomaly detection tools detect usually only very specific issues and produce many false alarms [18, 31, 66, 80].
- *Tracking known vulnerabilities:* Many third-party services scan the dependency tree of Node.js applications for known vulnerabilities (e.g., *Snyk.io*, *npm*, *GitHub*). This strategy is reactive, and research has shown that developers are developing notification fatigue and are slow to update [11, 19, 24, 44, 53, 79].
- *Avoiding automatic updates:* Rather than using automatic updates with version ranges, developers may *lock* package versions or use bots to only update dependencies after executing tests [34, 53, 78]. However, it is not clear that automated test executions would detect malicious updates.
- *Infrastructure hardening:* two-factor authentication in the *npm* package manager [56] reduces some attack vectors but does not protect against attacks using social engineering as in past incidents.
- *Application-level sandboxing:* Some Node.js applications are deployed within a sandbox (e.g., containers [58]), reducing potential damage. However, sandboxing is done at application level where all packages have the same capabilities as the application (where the application often rightfully has access to files, databases, or the network).

All these practices help but offer only limited protection. More secure solutions from academic security research on isolating individual packages or tracking information flows (cf. Sec. IV & VI) are not adopted in practice because of their limitations. We complement existing practices with an easy to adopt and low-overhead sandboxing strategy at the package level that can substantially reduce the attack surface.

III. Permission System Design

We propose a permission system that sandboxes packages and enforces *per-package* permissions in Node.js applications, i.e., we enforce a *least-privilege design [75] at the package level.*

Our approach is not the first to sandbox individual *npm* packages [35, 73, 74] (cf. Sec. VI), and there is a large design space for possible solutions, as we will discuss in Sec. IV. However, our approach identifies *a novel design* that provides protections for a large subset of packages without requiring changes to package implementations and with negligible overhead. We align our design with the requirements and values of the Node.js/npm community and propose it as one useful building block in a security strategy.

A. Goals and Assumptions

The design of the permission system focused on three main goals that are important for it to be relevant in practical software engineering settings: First, the permission system should actually *reduce the attack surface* of applications by containing certain types of attacks. Second, the permission system should *not require major infrastructure changes*, be backward compatible, and *not break existing user code* (assuming sufficient permissions). Lastly, the proposed permission enforcement technique should have *low performance overhead*, which is relevant for practical adoption.

In this work, we focus exclusively on malicious package updates, which are attacks following the following pattern: First, an attacker obtains credentials of package developers by using leaked *npm* package manager credentials in Git repositories, gaining access to a package developer's machine, buying packages, or using traditional tactics such as targeting weak passwords, phishing, social engineering, and typo-squatting. Note, it is sufficient to compromise the credentials of a single developer among an application's often hundreds of transitive dependencies. Second, once the attacker uses the credentials to publish malicious code with an update, applications that directly or indirectly depend upon the package and install updates (automatically or manually) are at risk. Once a malicious package is loaded in a running application, it may import native modules, import modules from other packages, and use metaprogramming constructs to perform malicious actions (see Fig. 1).

B. Package Permissions

We follow a familiar permission strategy, as known from mobile apps or web-browsers extensions: (1) developers declare required permissions from a small set of common and easy to understand permissions for their packages, which would be shown in the *npm* repository and by the command-line tools, (2) the system enforces that the package does not use not-required permissions, and (3) developers who add or update a package dependency must accept the package's permissions at installation time and again when permissions change in an update.

On permission systems: These kinds of permissions systems are well understood by users, easy to use for developers, and also well studied, including problems of developers asking for too many permissions, and users ignoring permissions [7, 28, 37, 77]. Our design shares similar challenges, but we expect fewer practical problems due to fewer monetization concerns (e.g., many Android permissions are needed just for targeted advertising) and a different target audience: Package users are developers and can usually clearly understand why a package would or would not need specific permissions (e.g., a string template engine needing network access would raise immediate suspicion). As permission changes on Node.js/npm are rare and suspicious, especially for minor and patch updates (see Sec. V-C), developers and the community at large are much more likely to focus their attention on such updates.

Set of permissions: Our design is not limited to a specific set of permissions (i.e., other specific permissions can be defined and mapped to other security-relevant resources, if desired), but for our discussion, implementation, and evaluation we consider four easy to understand permissions:

- The **network** permission is required to reference APIs to communicate with remote servers (e.g., HTTP, sockets). Specifically, the native modules *http, http2, https*, and *net* require this permission. Without the *network* permission, malicious code cannot leak data over the network.
- The **filesystem** permission is required to reference APIs to access the local filesystem, especially the native module *fs*. Without this permission code cannot perform attacks that read, write, or delete local files.
- The **process** permission is required to reference APIs for interacting with operating-system processes, particularly the native module *child_process*. Without this permission, malicious code cannot open reverse shells or kill processes.
- The **all** permission is required to use metaprogramming constructs (e.g., *eval, with*). Without this permission, malicious code cannot affect applications globally (e.g., modify the prototypes of native objects) and cannot evade the permission system. The **all** permission is a *superset* of the other permissions, since the use of metaprogramming constructs enables packages to obtain references to the security-relevant resources enabled by the other three permissions.

In our specific design, each package may require one or multiple permissions, which then apply to all modules in that package. Intuitively, the code of a package can only import modules from packages that have the same or fewer permissions. Permissions of native modules are hard-coded.

As we will explain, mapping permissions to code, a source object from a module with permissions X may only hold a reference to a value originating from a module with permissions Y if $Y \subseteq X$.

Package permissions are composed transitively: To depend on another package, a package must have at least the same permissions. For packages, this implies that they cannot circumvent the permission system by delegating critical tasks to packages that have the suitable permissions.

For developers, this means that they can easily (i) interpret a package's permissions without also investigating all indirect dependencies and (ii) declare the needed permissions for their own packages based on which permissions imported packages need.

Permission enforcement: The challenge when designing a permission system is in enforcing permissions, to prevent that attackers gain access to resources for which a package does not have permission. Intuitively, our enforcement mechanism needs to ensure that *source* objects from a module cannot import *target* objects from another module, including native modules, for which they do not have permission to import.

We considered different enforcement design options, but settled on a lightweight sandboxing strategy that combines dynamic checks with static analysis. We arrived at this design after exploring alternative designs in the design space for a practical and lightweight, yet effective solution; we discuss alternative designs and their trade-offs in Sec. IV.

C. Specification: Protecting Security-Relevant Resources

Given the dynamic nature of JavaScript and the design of Node.js, there are many ways code can gain references to objects from other modules. In Figure 2, we define a concrete policy that our permission system aims to enforce: If a module has the **all** permission, we do not enforce any restrictions (case 1), otherwise we only allow references to objects from modules with the same or fewer permissions, typically received via import (case 2), with the *global* object being a special case that may always be referenced (case 3). In addition, we allow three recursive mechanisms from which modules can derive new references from legally held references: received as arguments from a function call where the caller was allowed to hold the reference (case 4), received as return value from a call to a legally referenced function (case 5), or received by accessing the properties of a legally referenced object with the exception of a few restricted properties (case 6).

More intuitively, the specification prohibits *actively* importing objects from modules without suitable permissions, but it allows code to receive and hold references without corresponding permissions when those references are explicitly *provided by other modules* through function parameters, return values of function calls, or global variables. This design allows modules to pass objects (including callback functions) to modules with fewer permissions. It is the caller's responsibility not to provide security-critical references to untrusted code and it is unlikely that a malicious package update for a package without permissions can expect being passed the right security-critical resource (e.g., the *http* module) as an argument. This restriction puts some burden on developers, but is standard in the design of permission and effect systems [50].

In case 6, we restrict the following properties from the *require*, *module*, and *global* objects: `{require.main, module.paths, module._load, module.globalPaths,` *A **source** object (representing module A) may hold a reference to a value **v** originating from a module B if and only if:*

*(1) package A has the **all** permission, or*
(2) A has at least the same permissions as B, or
*(3) **v** is the **global** object, or*
*(4) **v** was received from a third object as a parameter in a function call to the **source** object, where the third object may hold a reference to **v**, or*
*(5) **v** is the resulting value of calling a function **f** and the **source** object may hold a reference to **f**, or*
*(6) **v** is the value held by property **p** of an object **o** where the **source** object may hold a reference to **o**, unless property **p** is restricted for object **o***

Fig. 2: Protecting Access To Security-Relevant Resources.

`module.constructor`, `module.parent`, `module.children`, `global.eval`, `global.Function`, `global.process`}. In addition, we restrict {`prototype`, `__proto__`, `create`, `setPrototypeOf`} for *native objects* (e.g., *Object*). All of these may lead to unprotected import mechanisms or enable non-local changes via metaprogramming such as prototype pollution attacks [60]; modules rarely use these access paths in practice, and they can still continue to do so if needed, requesting the **all** permission.

D. Enforcement: Protecting Security-Relevant Resources

To prevent active access to objects from other modules without suitable permissions we need to effectively perform checks only for two actions: First, we need to control module imports with Node.js' *require* function and, second, we need to control property access for certain restricted properties (which could be used for accessing other import mechanisms and evading the sandbox). By working with the existing implementation structures of Node.js and making small modifications to the runtime, both restrictions can be enforced without requiring developers to modify their packages and with negligible runtime overhead, as we will explain.

(1) Mediating the main import mechanism (**require*****):*** Performing additional runtime checks during imports is straightforward with only small modifications of the Node.js runtime (without any modifications of the modules).

Node.js provides a function *require* to every module that can be called to import packages (technically using the *Module Pattern* [59]). This design is beneficial for us, since each module already receives its own *require* function. To intercept all imports, we simply wrap the provided *require* with one that conducts permission checks for this module with a one-line modification of the Node.js runtime shown in Figure 3 – thus comparing the permissions of the importing package with those of the imported package. Beyond the one-line modification to insert the wrapper, our implementation for loading and comparing permissions is less than 100 lines of code.

Note that a module can potentially gain access to a *require* function of a different module. To prevent *active* access to other module's require functions, the access path *module.parent* is restricted.

```
1  Module._compile = function(code, file) {
2    var rcode = propAccessRewrite(code);
3    var wcode = Module.wrap(rcode);
4    var cwpr = vm.runInThisContext(wcode);
5    var dir = path.dirname(file);
6    var wreq = wrapRequire(require, dir);
7    var args = [this.exports, wreq, this, dir];
8    return cwpr.apply(this.exports, args);
9  }
10 function wrapRequire(require, currentModule) {
11   var permA = lazy(loadPermissions(currentModule));
12   return function(targetModule) {
13     var permB = loadPermissions(targetModule);
14     if (!subset(permA(), permB))
15       throw new Error('...');
16     return require(targetModule);
17 }}
```

Fig. 3: Package loading mechanism in the Node.js runtime system. Our modifications are highlighted in blue: the re-writing of property accesses (Line 2) and the wrapping of the *require* function with permission checks (Lines 6).

(2) Mediating property accesses in special objects: The second part of our enforcement, preventing access to certain restricted properties for certain objects, requires slightly more extensive changes, but is also fairly straightforward. Since we cannot fully statically reason about property access in JavaScript, we combine static analysis with selected dynamic checks. At load-time, we automatically *rewrite* the code of each module without the **all** permission to insert dynamic checks for every property access for which we cannot statically exclude that it may access a restricted property.

Our rewrite rules, which we apply to all modules without the **all** permission at load time work in two steps: normalizing references to global variables and introducing dynamic checks. First, using scope analysis, we rewrite references to global variables to make the property access visible, for example, rewriting *console.log* to *global.console.log* (the *with* statement which may prevent accurate scope analysis requires the **all** permission). Second, for every property access in the form `x.y` or `x[y]` (outside the right-hand side of an assignment), we introduce a dynamic check `$$prop(x, y)`, unless `y` can be resolved statically (name or string literal) to a name that is not in the list of restricted properties. Function `$$prop` (fresh random name generated for every module) checks whether the object-property combination is restricted, as shown in Figure 5.

Our rewrite technique *conservatively* combines static and dynamic analysis. For many property access locations, we can statically identify the property name and avoid dynamic checks if the name is never restricted. If the name is restricted for some objects, we insert a runtime check to determine whether the target object is the restricted one; if we cannot statically resolve the property name, we introduce a dynamic check, as illustrated in Figure 4. In practice few dynamic checks are needed in most modules.

Implementation: We implement our permission system with the described extensions to the Node.js platform. We store declared permissions in a dedicated file in the package. We use *esprima* to rewrite code and *escope* to analyze scope.

```
z.cache;
var x = b
x.foo()
eval("...")
o[f()]
o["ma"+"in"]
```

⇒

```
global['z']['cache']
var x = global['b']
x.foo()
$$prop(global, 'eval')('...')
$$prop(global['o'], global['f']())
$$prop(global['o'], 'ma' + 'in')
```

Fig. 4: Examples of member expressions re-writing (including normalization of global properties).

```
restrictedMap["parent"] = module;
restrictedMap["eval"] = global;
restrictedMap["prototype"] = Object;
...
function $$prop(obj, p) {
  if (obj == undefined) return undefined;
  if (restrictedMap[p] && obj == restrictedMap[p])
    throw new Error('...');
  return obj[p];
}
```

Fig. 5: Simplified code of dynamic checks with `$$prop`.

IV. Design Space

Notice how our solution is lightweight, intercepting calls to *require* with small changes to the Node.js runtime and dynamically ensuring that certain properties are not referenced. No further expensive information flow or origin tracking is needed to enforce the policy at runtime.

When designing our permission system, both policy and corresponding enforcement system, we explored many design alternatives and settled on the presented design after multiple experiments and iterations. While our design may appear simple, limited, or even obvious in retrospect, it is actually the result of careful consideration of multiple trade-offs, involving provided security guarantees, flexibility and understandability, backward compatibility and ease of adoption, and runtime overhead. In the following, we describe the design space, alternative designs, and justify our design decisions. Let us start by discussing general concerns before exploring two obvious alternative designs.

Permission granularity: We assign permissions at the granularity of packages, rather than entire applications or modules within those packages. Application-level permissions can already be assured with traditional sandboxing techniques (e.g. containers), though this would give malicious packages all resources the application may rightfully have. Package-level permissions, as opposed to module-level permissions, is suitable because *packages* are the building blocks that developers install and refer to in their applications, whereas *modules* within a package are rarely referred directly (information hiding).

We settled on the set of 4 simple permissions after multiple iterations reflecting that they correspond to the most important resources and to allow for flexible and advanced metaprogramming behavior in trusted packages if needed. The approach could be extended with a more fine-grained permission model where access to individual access paths or permissions for specific parameters could be restricted (e.g., file access in a specific directory only), but such approach would raise complexity and runtime overhead. Given that either design would be limited to a subset of modules, we opted for the simpler, more runtime-efficient, and easier to adopt design.

Integration with user-facing tools: The integration of the permission system with current tools and developers workflow are an important part for its adoption in practice. Even though we do not evaluate the integration in this paper, we outline the changes in user-facing tools and in developers' workflow that would be required to integrate the permission system in the Node.js/npm ecosystem.

Package owners are expected to manually declare required permissions in the package's manifest file before publishing a package. Package users can see required permissions in the npm repository before installation and will be notified about permission changes and potential permission mismatches at installation/update time. For example, one might be notified that their own package does not have permissions to import another package, so one has to update one's own permissions. Tools like *npm* or *yarn* package managers are expected to *not* automatically update packages when additional permissions are requested in the update, but instead require extra confirmation, encouraging a closer look at the update.

Alternative: Taint Tracking: The most obvious alternative solution would be a policy that restricts the flow of sensitive information to security-sensitive sinks using information-flow analysis [4, 8, 38], such that packages require permissions to initiate certain flows. Such policy is much more flexible, because rather than completely restricting access to packages with requiring additional permissions, we can allow packages to import other packages as long as they do not use them (directly or indirectly) to leak sensitive information; it would allow more flexible differentiation of which information may flow into which sinks independent of which packages are involved; it might also depend less on the fact that many packages are simple in their implementation. In addition, such a policy could be enforced uniformly independently of how modules obtain references to other modules (actively or passively) and, given a suitable information-flow tracking system, we would not need to restrict *eval* and metaprogramming.

However, we ruled out information-flow policies because corresponding static information-flow analyses are notoriously difficult to implement precisely in a dynamic language like JavaScript [36, 70] and dynamic information-flow analysis tend to have unacceptable performance overhead to consider adoption in practice (often 40x-100x performance overhead) [4, 9, 13, 45]. Neither the performance overhead nor a requirement for developers to rewrite or annotate their code would have made ecosystem-wide adoption plausible.

Alternative: Restricting Inter-Module Communication: While our policy restricts the *access* to other packages (and their resources), an alternative design could assign permissions to modules (as in our design) and then restrict the *communication* across package boundaries: A corresponding policy could simply state that no code originating from a module with permissions X shall call code from a module with permissions Y unless $Y \subseteq X$, restricting packages from calling code from other packages that require more permissions.

Checking function calls to packages is more flexible than checking imports of packages, as it provides further opportunities for inspecting arguments and return values, and observing how packages communicate. More importantly, such design could ensure that a package can only call other packages with suitable permissions even when a reference to a security-relevant resource is received *passively*, e.g., passed in by the user of the package as argument. Such design could also avoid special handling of restricted properties that our design requires to avoid unchecked access to certain references.

Such designs have previously been explored to sandbox individual code fragments, usually a single third-party code fragment included on a web page [2, 61] or a single module in a Node.js application [35]. The key is to track *origins* of all functions (e.g., when a function is passed from module A to B it still needs to be associated with A) and to intercept all calls that cross module boundaries to check permissions corresponding to origin modules of the calling and called functions. In JavaScript, a typical solution is to install monitors between package boundaries using proxy objects, following the *Membrane Pattern* [17, 35, 73], but also replacing calls by inter-process communication has been explored [74].

Despite its elegance, this policy and the corresponding implementations impose at least three severe challenges that limit its practical use:

- The performance overhead of the Membrane Pattern is nontrivial. Prior work, isolating only a single package, reported about a 20 percent slowdown [17, 35, 73]; we observed similar and slightly higher slowdowns in our own experiments installing membranes among all packages. This overhead may discourage adoption in many settings for server-side applications.
- Node.js packages and many native modules heavily rely on callback functions, for example, to asynchronously return the result of an I/O operation. Since callback calls also cross package boundaries, it would be subject to the same permission checks, essentially requiring the same permissions for caller and callee, making this policy way too restrictive in the context of typical Node.js programming patterns. Instead, we would have to design a policy that specifically allows certain call patterns, which is nontrivial and would eventually permit most *passive* means of accessing security-critical resources also allowed in our design.
- Implementing enforcement with the Membrane Pattern is technically challenging. First, while it is easy to install a membrane for modules, using the membrane or other mechanisms to handle the *global* object and its properties requires invasive changes into Node.js and the underlying JavaScript engine. Second, installing proxy objects for objects breaks many existing implementations since proxies are not entirely transparent (e.g., object identity is no longer reliable). We implemented a prototype to run experiments with the Membrane Pattern and had to adapt many existing implementations. While technical challenges can potentially be addressed in an automated way (invasive changes to Node.js runtime, automated code rewriting to account for proxy behavior), such implementation is complex and difficult to maintain with frequent updates to the Node.js engine.

While the policy to restrict inter-module communication seems more elegant and requires fewer exceptions, performance and implementation challenges are too severe to justify its marginal benefits. To account for callbacks and global, actual policies would likely have similar limitations as our own policy.

Design space summary: Overall, while we cannot support the flexibility of a policy based on taint tracking, our approach provides similar protections to prior approaches checking inter-module communication, but with a much simpler implementation and with negligible performance overhead.

While our design cannot provide protections for modules that legitimately need access to security-critical resources or metaprogramming as part of their implementation (neither can approaches based on restricting inter-module communication), it protects those small and simple packages that are common in Node.js/npm and do not need any permissions almost for free; Source modules may still need to be inspected, to ensure that they do not provide security-critical resources to target modules with fewer permissions.

V. Evaluation

To evaluate our proposed permission system, we consider the goals of our design (see Sec. III-A) and show that our permission system (1) can significantly reduce the attack surface of applications and npm in general, (2) is useful in containing real attacks, (3) incurs negligible performance overhead, and (4) can reduce review effort for security purposes on package updates. Specifically, we answer the following research questions:

- **RQ1:** How many packages in the *npm* repository could we protect with our permission system?
- **RQ2:** How effective could the permission system be to contain attacks like the *eslint-scope*, *event-stream*, and *electron-native-notify* attacks?
- **RQ3:** What performance overhead would the permission system cause in Node.js applications and is the permission system transparent?
- **RQ4:** How much review effort could be saved on package updates with our permission system in a realistic setting?

The evaluation of our proposed permission system maps directly to our design goals (see Sec. III-A). Research questions RQ1 and RQ2 map to our first design goal described in Sec. III-A, which is to propose a solution that *reduces the attack surface* of applications by *containing certain types of attacks*. RQ3 maps to our second and third design goals, which aim at demonstrating transparency and low performance overhead of our proposed permission system design, both important aspects for practical adoption. Finally, RQ4 aims at illustrating another benefit of our (partial) solution: its saves review effort on package updates. It also shows how permission changes are rare and suspicious, especially for minor and patch updates, and that the developer community is much more likely to focus their attention on such updates.

A. Protected Packages (RQ1)

To show how our permission system can help protect packages and applications, significantly reducing the attack surface of applications and npm in general, we approximate the required permission for all packages in the *npm* repository and report how many packages can no longer gain access to security-relevant resources, making malicious updates or other exploits targeting those packages ineffective. To answer RQ1, we report the relative amount of all packages in the *npm* repository that would need each permission (either directly or indirectly through its dependencies).

***Data Collection and Study Design**:* To answer RQ1, we use a snapshot of the entire *npm* repository with the latest versions of all packages. We gathered a total of 703,457 packages (February 2018), from which 604,159 contain valid JavaScript code. For each package, we analyze both the manifest file and all JavaScript source files (modules) to identify declared dependencies, imports to other modules, and expressions with property accesses. We infer permissions by searching for *require* calls to native modules. We then assign permissions to each package, recursively considering the permissions of its direct and indirect dependencies, as defined by our compositional permission model (see Sec. III-B).

Since not all packages in the *npm* repository are equally used and updated [22], we also analyze packages samples of size 100 according to four criteria: most downloaded, most depended on within the *npm* repository, most stars on GitHub (a common popularity measure), and most updated in the year prior to our sample date. We gathered *downloads*, *dependencies*, and *updates* statistics from the *npm* repository and *stars* from GitHub. Note that these criteria look at the dataset from different facets, for example, *downloads* may be biased more toward utility packages used by many other packages, *dependencies* indicate popularity among library developers, *updates* highlight packages that are creating a particularly high review load if one was to review all updates, and *stars* indicate popularity or attention by users more broadly.

Note that our permission inference is an approximation, whereas in practice we would expect developers to manually declare required permissions. Our inference may miss some required permission (that our runtime enforcement would catch), for example, when imports use dynamically computed names to import native modules (dynamic imports are found in 8% of all packages, though they rarely seem to import native modules) and may sometimes infer the **all** permission for non-problematic access to restricted paths. We validate the accuracy of our permission inference by manually identifying permissions of 30 randomly chosen packages and found that those all matched the automatically inferred ones, except 7 cases where we unnecessarily inferred the **all** permission. Furthermore, we did not encounter issues from missing permissions in our experiments (Table II), further indicating accurate inference.

***Results**:* Based on our permission inference, we found that 192,585 packages in the *npm* repository (31.9%) do not need any permission and another 9.4% need only a subset of all permissions. Among the most downloaded, depended, starred, and updated packages, a higher percentage of packages needs some or all permissions, but still a significant number of those packages need no (14–33%) or only some permissions (1–11%) as shown in Table I. The differences in different populations are to be expected, as popular (starred) packages tend to be larger end-user packages, whereas downloads favor smaller, often indirectly used utilities.

Permissions	all	downl.	dep.	stars	upd.
no perm.	31.9%	27%	15%	14%	33%
only **network** perm.	1.2%	0%	1%	0%	3%
only **filesys.** perm.	4.8%	8%	11%	1%	6%
only **process** perm.	0.7%	0%	1%	0%	0%
multiple perm.	2.7%	5%	2%	0%	2%
all perm.	58.7%	60%	70%	85%	56%

TABLE I: Distribution of permissions of all packages and the 100 most downloaded/depended/starred/updated packages.

Note, many packages require permissions primarily due to dependencies: We infer that 21.9% of all packages directly need the **all** permission, while another 36.8% inherit the **all** permission from depended on packages. Among packages that directly need the **all** permission, around 5% directly use *eval* while the remaining of packages change the prototype of native objects such as *Object*, *String*, *Array*, and others. Similarly, we infer that only 4.7, 15.6, and 2.1% of all packages need the **network**, **filesystem**, and **process** permissions. Multiple permissions refer to combinations of permissions (e.g., **network+process**) and packages that need them are rarely found in the *npm* repository.

Overall, while there are a large number of packages that genuinely need access to security-critical APIs and a significant number of packages that use language features that cannot be contained by our mechanism (hence the **all** permission), these results also confirm that many packages published on the *npm* repository are indeed fairly simple and can be protected with our lightweight permission and enforcement system. Since permissions are enforced for all packages (including their clients), any attempt of a package to import a security-relevant resource that does not correspond to its permissions would fail, even if a package uses a transitive dependency relationship with another package.

In summary, our permission system can protect the 14–33% packages that need no permission and partially protect another 1–11% percent that need only a subset of all permissions. These packages can no longer gain access to security-relevant resources, making malicious updates that attempt to elevate packages' privileges ineffective. Thus, reducing the attack surface for the npm repository and for typical Node.js applications that use these packages.

B. Containing Past Attacks (RQ2)

Our permission system is a general defense mechanism, but is not designed to detect new malicious attacks in the wild. It is designed to contain attacks. We illustrate the usefulness of the proposed permission system by containing some past attacks. To that end. we replay the *eslint-scope*, *event-stream*, and *electron-notify-native* attacks using our modified Node.js engine and show that the attacks would be ineffective.

Data Collection and Study Design: To evaluate the containment, we considered two versions of each attacked package: (i) the version that preceded the attack and (ii) the attacked version. First, we inferred the permissions of the version that preceded the attack and assigned the same permissions to the attacked version. Then, we installed the attacked version with the *npm install* command, which installs package dependencies and executes the installation scripts of the package. To replay the attacks on the *eslint-scope*, *event-stream*, and *electron-notify-native* packages, we used versions 3.7.2; 0.1.1 of the *flatmap-stream* package; and 1.1.6, respectively.

Results: None of the packages with malicious updates required any permissions in their latest release before the attack. When replaying the attacks, we observed in each case that the permission system contains all three attacks by preventing unauthorized imports of security-relevant resources and by denying the use of metaprogramming constructs at runtime. For example, when the malicious version 3.7.2 of the *eslint-scope* package is installed, a *post-install* script is executed, but the permission system prevents the imports of the *http* and *fs* modules. This occurs because the malicious version 3.7.2 did not have the corresponding permissions. An attacker would have had to explicitly request additional permissions, which makes the detection of the attack much more likely.

In summary, our permission system can successfully contain past malicious updates.

C. Performance and Transparency (RQ3)

To evaluate practicality, which is important for adoption (see Sec. III-A), we measure the performance overhead caused by the enforcement mechanism for our policy (see Sec. III-D) and whether it transparently supports the execution of packages without modifications (assuming sufficient permissions).

To be realistic, we measure the runtime overhead of *applications*, which represent actual usage scenarios relevant for end users, rather than conducting microbenchmarks on individual packages. To answer RQ3, we evaluate and report the performance overhead caused for 20 Node.js applications when executing them with and without the permission system.

Data Collection and Study Design: As subject systems, we collected 20 command-line applications and corresponding realistic workloads; which are common modern use cases for Node.js and released as open source (e.g., whereas for web services most public examples are demos or tutorials, most production servers are closed source). We selected the 20 most popular *npm* public applications that depend on the commonly used *commander* package, which provides command-line interfaces for Node.js applications. For each application, listed in Table II, we read the documentation and identified common inputs. Then, we selected a realistic large workload (usually a large JavaScript/JSON file) and measured performance when executing each application with an input.

To mitigate systematic errors in our performance measurements, we followed standard guidelines [32], running each application 10 times under each experimental condition, interleaving the runs across applications, and reporting the median execution time. To avoid the noise problems of microbenchmarks, we selected large workloads taking at least five seconds. All measurements were performed on a MacBook Pro equipped with a 2.8GHz Intel Core i7 processor and 16GB 1600MHz DDR3 of memory. The benchmarks are also shared with the project's repository.

Application	Baseline (in ms)	Overhead (in %)	Import Checks	Property Checks
d3-dsv	5268.0	0.3	25	408
docco	5148.5	0.0	71	65774
dot-object	5339.5	0.2	40	2
dox	5408.5	0.0	3478	21
findup	5227.5	0.1	12	0
html-minifier	5700.5	0.0	44	3
js-cfb	5300.0	0.0	10	0
js-xss	5261.5	0.3	11	0
js-yaml-front-matter	5200.0	0.0	7	7
json-refs	5894.5	0.0	88	750
json2csv	5317.0	0.1	15	3
juice	6143.0	-0.3	301	1104
metalsmith	5828.5	0.3	148	127
mocha	5141.5	0.0	116	5
mock	4947.0	1.3	7	0
node	6627.5	0.2	934	106
sails	6792.0	0.2	1364	36455
svgicons2svgfon	5427.5	0.1	57	2
traceur-compiler	13784.0	0.5	52	2
uglify-js	12468.0	0.9	10	1

TABLE II: Applications and performance results; runtime of the unmodified Node.js (Baseline) and the overhead of the modified Node.js with the permission system (Overhead); reporting also the number of executed import and property access checks.

Results: We report the observed performance differences in Table II, which across all applications, is negligible. The permission system causes a small performance overhead—barely distinguishable from measurement noise—typically under 1 percent (average 0.2%). The difference in overhead between the executions with the unmodified and the modified Node.js is not significant (Welch's $t(0.0218) = 37.9964$, $p < 0.9827$).

Notice how the overhead caused by the permission system is negligible, compared to alternative isolation and compartmentalization solutions [35, 73, 74] with $> 20\%$ overhead or information flow solutions with $> 20\times$ overhead; cf. Sec. IV). The number of dynamic import checks and property checks needed in those applications (including all direct and indirect dependencies) differ substantially from application to application depending on how many and what kind of dependencies they use, but they are usually low, explaining the low runtime overhead. For instance, *docco* and *sails* are both HTML generators. While the amount of *actually executed property checks* is much larger for these two applications (as we show in Table II), the amount of *rewritten property checks* is not. For these two applications, the rewritten property checks get executed several times when generating HTML. By inspecting their code, we observed they change the prototype of several native objects (e.g., *String*, *Object*), which cause the rewrites.

Even in systems with higher numbers of property access checks, the overhead of these checks is dwarfed by the main computations and I/O operations of these applications.

Furthermore, our experiments confirm that our solution is entirely transparent for the executed applications, if permissions are correctly set. All applications work as before without any source code modification of the application or its dependencies.

In summary, our permission system is transparent and causes negligible performance overhead ($\ll 1\%$).

D. Review Effort Reduction on Package Updates (RQ4)

Our final research question hypothesizes a scenario in which developers would review all package updates (which may be recommended but is rarely done systematically in practice, cf. Sec. II). We argue that even though, in practice, developers do not review all updates, our permission system would allow developers to save even more effort if they skip reviews for those packages that do not need any permissions and pay particular attention to updates that request additional permissions. Thus, we evaluate how our permission system can reduce the review load in this scenario, indirectly demonstrating the potential usefulness of our (partial) solution.

Data Collection and Study Design: We analyze package updates and observe how permissions evolve by replaying the evolution and updates of dependencies for a sample of packages and applications over a one-year period. We observe updates both from the perspective of a package maintainer of highly depended packages and from the perspective of an application developer by analyzing two datasets: (i) the 100-most-depended-upon packages in the *npm* repository that we previously described for RQ1 and (ii) the 20 Node.js applications from RQ3. For each of the 120 analyzed packages and applications, we computed their entire dependency tree (including resolving the latest version of a dependency when version constraints are declared as a range) for each day in 2018 and identified all package updates that happened in that time period. This is equivalent to installing or updating these applications every day, approximating common practice among Node.js developers [11].

We downloaded all distinct packages and their versions to infer approximate permissions (as in RQ1). For each update, we collect whether and how the corresponding permissions have changed. In total, we analyzed 4,962 distinct dependency versions for the 100 most-depended-upon packages and 1,310 distinct dependency versions for the 20 applications.

Results: During the one-year period, a total of 5,042 package updates occurred for all direct and indirect dependencies of the 100-most-depended-upon packages; an average of 66 updates per year per subject. Among these, 2,644 updates (52 percent) were to packages that did not need any permissions before or after and could be installed without any review – this represents a substantial reduction in review effort in our scenario. The percentage of no-permission updates naturally differed among the datasets, but represented a substantial number of updates in most cases. Applications, on average, had a lower proportion of no-permission updates (6 percent), explained by the larger number of non-trivial packages they depended on. Updates from packages with all permissions make up 33 and 86 percent of all updates in the top libraries and applications respectively.

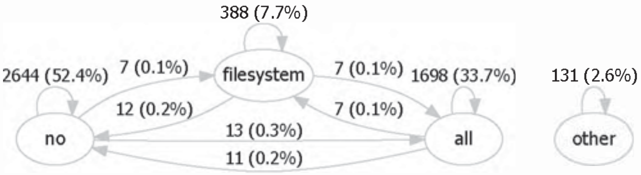

Fig. 6: Permission changes for the 100-most-depended-upon packages and all its direct and indirect dependencies. *Nodes* represent permissions, *edges* represent permission changes in updates, and labels indicate the absolute and relative number of changes over a year.

Changes among the permissions in updates in the top libraries were generally rare, as shown in Figure 6. There are only 27 updates (0.5 percent) that needed more permissions than the previous one; most maintainers would not face such an update more than once a year, making it realistic that they would analyze such packages more closely. Nodes that represent other permissions (e.g., the *network* node) and and edges that represent other permission changes (e.g., the edge from *filesystem* to the *network*) are grouped in the *other* node and account for only 2 percent of the updates (see Figure 6).

In summary, our permission system can reduce review effort for updates substantially (6–52%) in our scenario.

E. Threats to Validity

First, different subpopulations of packages in the *npm* repository may have different characteristics and may need more or fewer permissions on average; our results must be interpreted as averages over all of packages and averages over popular packages in the *npm* repository. Second, it is challenging to assemble a representative set of benchmarks of *Node.js* applications: while libraries are published on the *npm* repository, applications are often proprietary and not public on the *npm* repository or *GitHub*. We use publicly released command-line utils to represent applications, but generalizations must be made with care. Third, as discussed, our permission inference is only an approximation of the permissions a developer would manually declare; despite our careful validation we may miss permissions or developers may choose to over-permission their packages affecting the results. Finally, the evaluation of review effort makes strong assumptions on developer behavior and uniform review effort per package that may not be realistic in practice, especially with often observed complacency or a false sense of security from a security mechanism like ours; readers may extrapolate other behavior from the reported numbers.

VI. Related Work

Security challenges with the JavaScript programming language and the limits of JavaScript analysis techniques have been widely explored in the web browser context, with less focus on the Node.js/npm ecosystem.

Node.js/npm Ecosystem: Ojamaa and Düüna [57] discussed security challenges of the Node.js platform, pointing out typical pitfalls of JavaScript programming, such as the single-threaded event-loop-based architecture, *eval*, lack of isolation, but also highlighting the possibility of attacks through malicious package updates. Some of the pitfalls (e.g., *eval*) could result in insecure applications and are partially addressed by our permission system. Wittern et al. [79] found that the number of direct dependencies of Node.js packages grows over time and that over 40% of all packages allow automatic updates of minor revisions. These results reinforce the common but problematic practice of accepting automatic package updates (see Sec. II). Decan et al. [19] reported that the *npm* repository has an abundant number of packages whose failure can impact the ecosystem, some affecting more than 30 percent of the packages available on the *npm* repository. Abdalkareem et al. [1] studied the pervasive use of small, single purpose packages in the *npm* repository, which motivated our permission system focused on the many simple packages.

Hejderup [40] analyzed dependencies among packages published on the *npm* repository and found that known vulnerabilities in packages often affect many other dependent packages in the ecosystem, with many packages depending on vulnerable versions for a significant time after a patch has been released. Similarly, Decan et al. [21] analyzed packages in the *npm* repository over six years and found that package vulnerabilities affect many dependent packages, but it still takes a long time for them to be discovered and fixed. Zimmermann et al. [83] further reported many unmaintained packages in the *npm* repository, which no longer receive patches, indirectly threatening the security of the Node.js/npm ecosystem. Staicu et al. [70] further reported how injection vulnerabilities are prevalent in the Node.js/npm ecosystem. These results highlight an orthogonal but relevant issue: not updating vulnerable package dependencies can cause as much damage as accepting malicious package updates. In practice, many developers rely on tools to track known vulnerabilities (Sec. II).

JavaScript Analysis: It is well understood that analyzing and sandboxing JavaScript code is difficult. Prior work, usually focused on untrusted scripts embedded in web pages, restricts how third-party code can interact with application code, a browser, or the web page. Due to the dynamic nature of JavaScript programs, many existing analysis approaches combine static analysis with runtime mechanisms, like our approach. Strategies typically either define and enforce a safe subset of JavaScript [10, 16, 30, 43, 48, 61], isolate scripts/packages in different execution environments (e.g., *iframes*) [42, 46, 52, 71], modify the browser to enforce restrictions at runtime [51], or attempt to sandbox a script/package without JavaScript engine modifications by rewriting it [2, 35, 72–74].

We combine several of these ideas (language subset, rewriting, runtime modification) in a lightweight design. More precise dynamic information flow techniques have been widely investigated in academic literature, but usually require very invasive changes to execution engines and are very slow [4, 13, 38, 39]; static information flow analysis is only feasible with requiring developers to write fine-grained annotations [3].

Permission Systems and Sandboxing: Permission systems are common to provide controlled access to security-relevant resources, allowing users to monitor, review, and revoke applications' permissions, if applications' behaviors (or intentions) are misaligned with application users' expectations. Our permission system is inspired by the one available in Android OS, which has been broadly studied [7, 28, 29, 41, 77], which isolates apps from each other rather than packages within a single application. While the systems are not directly comparable and have different user populations, we can learn from their experience, including well studied problems such as developers asking for too many permissions and users ignoring permissions [7, 28, 37, 77].

Sandboxes are important building materials to harden the security of diverse software systems and are typically used simultaneously as (i) encapsulation and as (ii) policy enforcement mechanisms [47]. In our case, we effectively implement a sandbox in the Node.js engine, to enforce our permission-dependent policy to limit the behaviors of packages at runtime. More commonly, sandboxing isolates programs in the operating system or in containers [67, 76, 81], or, for JavaScript, isolates untrusted code in the browser [2, 16, 61].

VII. Conclusion

In this paper, we discuss the emerging security challenges of malicious package updates which recently surfaced in multiple software ecosystems [5, 6, 14, 23, 27, 69]. We design a permission system with a policy and corresponding enforcement mechanism that sandboxes individual packages, rather than entire applications, ensuring that malicious updates cannot use security-critical resources for which they do not have permissions. We design our system to be simple, easy to adopt, with marginal runtime overhead and show that 31.9% of all packages can be protected at almost no cost.

Acknowledgment

This work has been supported by NSF award 1717022. G. Ferreira is supported by a doctoral research grant from CAPES, Brazil.

References

[1] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. 2017. Why Do Developers Use Trivial Packages? An Empirical Case Study on NPM. In *Proc. Foundations of Software Engineering (ESEC/FSE)*. 385–395.

[2] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H Phung, Lieven Desmet, and Frank Piessens. 2012. JSand: Complete Client-side Sandboxing of Third-party JavaScript Without Browser Modifications. In *Proc. Annual Computer Sec. Appl. Conf. (ACSAC)*. 1–10.

[3] Owen Arden, Michael. D. George, Jed Liu, K. Vikram, Aslan Askarov, and Andrew C. Myers. 2012. Sharing Mobile Code Securely with Information Flow Control. In *Proc. IEEE Symposium on Security and Privacy (IEEE S&P)*. 191–205.

[4] Thomas H. Austin and Cormac Flanagan. 2012. Multiple Facets for Dynamic Information Flow. In *Proc. Symp. Principles of Programming Languages (POPL)*. 165–178.

[5] Adam Baldwin. 2018. Reported malicious module: getcookies. https://blog.npmjs.org/post/173526807575/reported-malicious-module-getcookies

[6] Adam Baldwin. 2019. Plot to Steal Cryptocurrency Foiled by the npm Security Team. https://blog.npmjs.org/post/185397814280/plot-to-steal-cryptocurrency-foiled-by-the-npm

[7] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. 2012. Automatically Securing Permission-based Software by Reducing the Attack Surface: An Application to Android. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*. 274–277.

[8] Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, Michael Stroucken, and Yuan Tian. 2015. Run-time Monitoring and Formal Analysis of Information Flows in Chromium. In *Proc. Network and Distributed System Security Symposium (NDSS)*.

[9] Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, Michael Stroucken, and Yuan Tian. 2015. Run-time Monitoring and Formal Analysis of Information Flows in Chromium. In *Proc. Network and Distributed System Security Symposium (NDSS)*.

[10] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. 2014. Defensive JavaScript. In *Foundations of Security Analysis and Design (FOSAD)*. 88–123.

[11] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2016. How to Break an API: Cost Negotiation and Community Values in Three Software Ecosystems. In *Proc. Foundations of Software Engineering (FSE)*. 109–120.

[12] Justin Cappos, Justin Samuel, Scott Baker, and John H. Hartman. 2008. A Look in the Mirror: Attacks on Package Managers. In *Proc. Conf. on Computer and Communications Security (CCS)*. 565–574.

[13] Andrey Chudnov and David A. Naumann. 2015. Inlined Information Flow Monitoring for JavaScript. In *Proc. Conf. on Computer and Communications Security (CCS)*. 629–643.

[14] Catalin Cimpanu. [n.d.]. Two Malicious Python Libraries Caught Stealing SSH and GPG Keys (blog post). https://www.zdnet.com/article/two-malicious-python-libraries-removed-from-pypi/.

[15] Filipe Roseiro Cogo, Gustavo Ansaldi Oliva, and Ahmed E. Hassan. 2019. An Empirical Study of Dependency Downgrades in the npm Ecosystem. *IEEE Trans. Softw. Eng. (TSE)* (2019).

[16] Douglas Crockford. 2008. ADsafe. http://adsafe.org

[17] Tom Van Cutsem and Mark S. Miller. 2013. Trustworthy Proxies: Virtualizing Objects with Invariants. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*.

[18] Rickard Dahlstrom. [n.d.]. Using JSLint For Faster, Safer Coding With Less Javascript Errors (blog post). https://raygun.com/blog/jslint-safer-coding/.

[19] Alexandre Decan, Tom Mens, and Maeelick Claes. 2017. An Empirical Comparison of Dependency Issues in OSS Packaging Ecosystems. In *Proc. Int'l Conf. Software Analysis, Evolution and Reengineering (SANER)*. 2–12.

[20] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the Evolution of Technical Lag in the npm Package Dependency Network. In *Proc. Int'l Conf. Software Maintenance and Evolution (ICSME)*. 404–414.

[21] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the Impact of Security Vulnerabilities in the Npm Package Dependency Network. In *Proc. Mining Software Repositories (MSR)*. 181–191.

[22] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An Empirical Comparison of Dependency Network Evolution in Seven Software Packaging Ecosystems. *Empirical Software Engineering* 24, 1 (2019), 381–416.

[23] Hayley Denbraver. [n.d.]. Code Execution Back Door Found in Ruby's *rest-client* Library (blog post). https://snyk.io/blog/code-execution-back-door-found-in-rubys-rest-client-library/.

[24] Erik Derr et al. 2017. Keep Me Updated: An Empirical Study of Third-Party Library Updatability on Android. In *Proc. Conf. on Computer and Communications Security (CCS)*. 2187–2200.

[25] Ben Dickson. [n.d.]. The Complete Package: Everything You Need To Know About NPM Security (blog post). https://rb.gy/abrkpr.

[26] Jens Dietrich, David Pearce, Jacob Stringer, Amjed Tahir, and Kelly Blincoe. 2019. Dependency Versioning in the Wild. In *Proc. Mining Software Repositories (MSR)*. 349–359.

[27] ESLint. 2018. Postmortem for Malicious Packages Published on July 12th. https://eslint.org/blog/2018/07/postmortem-for-malicious-package-publishes

[28] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android Permissions Demystified. In *Proc. Conf. on Computer and Communications Security (CCS)*. 627–638.

[29] Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. The Effectiveness of Application Permissions. In *Proc. USENIX Conference on Web Application Development (WebApps)*. 7–7.

[30] Matthew Finifter, Joel Weinberger, and Adam Barth. 2010. Preventing Capability Leaks in Secure JavaScript Subsets. In *Proc. Network and Distributed System Security Symposium (NDSS)*.

[31] Kalil Garrett, Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. 2019. Detecting Suspicious Package Updates. In *Proc. Int'l Conf. Software Engineering, New Ideas and Emerging Results Track (ICSE-NIER)*. 13–16.

[32] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. 57–76.

[33] David Gilbertson. 2018. I'm harvesting credit card numbers and passwords from your site. Here's how. https://medium.com/p/9a8cb347c5b5

[34] greenkeeper.io. [n.d.]. Greenkeeper | Automate your npm dependency management. https://greenkeeper.io/.

[35] Willem De Groef, Fabio Massacci, and Frank Piessens. 2014. NodeSentry: Least-privilege Library Integration for Server-side JavaScript. In *Proc. Annual Computer Sec. Appl. Conf. (ACSAC)*. 446–455.

[36] Salvatore Guarnieri and Benjamin Livshits. 2009. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for Javascript Code. In *Proc. USENIX Conference on Security (SEC)*. 151–168.

[37] Arjun Guha, Matthew Fredrikson, Benjamin Livshits, and Nikhil Swamy. 2011. Verified Security for Browser Extensions. In *Proc. IEEE Symposium on Security and Privacy (IEEE S&P)*. 115–130.

[38] Daniel Hedin, Luciano Bello, and Andrei Sabelfeld. 2016. Information-flow Security for JavaScript and its APIs. *Journal of Computer Security* 24, 2 (2016), 181–234.

[39] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: Tracking Information Flow in JavaScript and Its APIs. In *Proc. Symp. Applied Computing (SAC)*. 1663–1671.

[40] Joseph Hejderup. 2015. *In Dependencies We Trust: How Vulnerable Are Dependencies in Software Modules?* Master's thesis. TU Delft, Delft University of Technology.

[41] Shashank Holavanalli, Don Manuel, Vishwas Nanjundaswamy, Brian Rosenberg, Feng Shen, Steven Y. Ko, and Lukasz Ziarek. 2013. Flow Permissions for Android. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*. 652–657.

[42] Lon Ingram and Michael Walfish. 2012. Treehouse: Javascript Sandboxes to Help Web Developers Help Themselves. In *Proc. USENIX Annual Technical Conference (USENIX ATC)*. 153–164.

[43] Vineeth Kashyap et al. 2014. JSAI: A Static Analysis Platform

for JavaScript. In *Proc. Foundations of Software Engineering (FSE)*. 121–132.

[44] Raula Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do Developers Update Their Library Dependencies? *Empirical Software Engineering* 23, 1 (2018), 384–417.

[45] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 Million Flows Later - Large-scale Detection of DOM-based XSS. In *Proc. Conf. on Computer and Communications Security (CCS)*.

[46] Mike Ter Louw, Karthik Thotta Ganesh, and V. N. Venkatakrishnan. 2010. AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In *Proc. USENIX Conference on Security (SEC)*. 24–24.

[47] Michael Maass, Adam Sales, Benjamin Chung, and Joshua Sunshine. 2016. A Systematic Analysis of the Science of Sandboxing. *PeerJ Computer Science* 2 (2016), e43.

[48] Sergio Maffeis, John C. Mitchell, and Ankur Taly. 2009. Language-based Isolation of Untrusted Javascript. In *IEEE Computer Security Foundations Symposium (CSF)*.

[49] Konstantinos Manikas. 2016. Revisiting Software Ecosystems Research: A Longitudinal Literature Study. *Journal of Systems and Software* 117 (2016), 84–103.

[50] Darya Melicher, Yangqingwei Shi, Alex Potanin, and Jonathan Aldrich. 2017. A Capability-based Module System for Authority Control. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*.

[51] Leo A. Meyerovich and Benjamin Livshits. 2010. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *Proc. IEEE Symposium on Security and Privacy (IEEE S&P)*. 481–496.

[52] James Mickens. 2014. Pivot: Fast, Synchronous Mashup Isolation Using Generator Chains. In *Proc. IEEE Symposium on Security and Privacy (IEEE S&P)*. 261–275.

[53] Samim Mirhosseini and Chris Parnin. 2017. Can Automated Pull Requests Encourage Software Developers to Upgrade Out-of-date Dependencies?. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*. 84–94.

[54] Kirill Nikitin et al. 2017. CHAINIAC: Proactive Software-update Transparency via Collectively Signed Skipchains and Verified Builds. In *Proc. USENIX Conference on Security (SEC)*. 1271–1287.

[55] npmjs.com. [n.d.]. Auditing Package Dependencies for Security Vulnerabilities. https://docs.npmjs.com/auditing-package-dependencies-for-security-vulnerabilities.

[56] npmjs.com. [n.d.]. Requiring 2FA for Package Publishing and Settings Modification. https://docs.npmjs.com/requiring-2fa-for-package-publishing-and-settings-modification.

[57] Andres Ojamaa and Karl Düüna. 2012. Assessing the Security of Node.js Platform. In *Proc. Int'l Conf. Internet Technology and Secured Transactions*. 348–355.

[58] Eric O'Rear. [n.d.]. Containerizing Node.js Applications with Docker (blog post). https://nodesource.com/blog/containerizing-node-js-applications-with-docker.

[59] Addy Osmani. 2012. *Learning Javascript Design Patterns*. O'Reilly Media, Inc.

[60] Phu H. Phung, David Sands, and Andrey Chudnov. 2009. Lightweight Self-protecting JavaScript. In *Proc. ASIA Conf. on Computer and Communications Security (ASIACCS)*. 47–60.

[61] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. 2011. ADsafety: Type-based Verification of JavaScript Sandboxing. In *Proc. USENIX Conference on Security (SEC)*. 12–12.

[62] Tom Preston-Werner. 2018. Semantic Versioning 2.0.0. https://semver.org/

[63] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011. The Eval That Men Do: A Large-scale Study of the Use of Eval in Javascript Applications. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*. 52–78.

[64] Sam Saccone. 2016. *npm hydra worm disclosure*. Technical Report. Google.

[65] Justin Samuel, Nick Mathewson, Justin Cappos, and Roger Dingledine. 2010. Survivable Key Compromise in Software Update Systems. In *Proc. Conf. on Computer and Communications Security (CCS)*. 61–72.

[66] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. 2018. Rescue: Crafting Regular Expression DOS Attacks. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*. 225–235.

[67] Takahiro Shinagawa et al. 2009. Bitvisor: A Thin Hypervisor for Enforcing I/O Device Security. In *Proc. Int'l Conf. Virtual Execution Environments*. 121–130.

[68] snyk.io. [n.d.]. Snyk | Develop Fast. Stay Secure. https://snyk.io/.

[69] snyk.io. 2018. Malicious code found in npm package event-stream. https://snyk.io/blog/malicious-code-found-in-npm-package-event-stream

[70] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. 2018. Synode: Understanding and Automatically Preventing Injection Attacks on Node.js. In *Proc. Network and Distributed System Security Symposium (NDSS)*. 57–76.

[71] Deian Stefan et al. 2014. Protecting Users by Confining JavaScript with COWL. In *Proc. USENIX Symp. Operating Systems Design and Impl. (OSDI)*. 131–146.

[72] Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. 2018. Efficient Dynamic Analysis for Node.Js. In *Proc. Int'l Conf. Compiler Construction (CC)*. 196–206.

[73] Tung Tran, Riccardo Pelizzi, and R. Sekar. 2015. JaTE: Transparent and Efficient JavaScript Confinement. In *Proc. Annual Computer Sec. Appl. Conf. (ACSAC)*. 151–160.

[74] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. 2018. BreakApp: Automated, Flexible Application Compartmentalization. In *Proc. Network and Distributed System Security Symposium (NDSS)*.

[75] John Viega and Gary McGraw. 2011. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley.

[76] Zhiyuan Wan, David Lo, Xin Xia, and Liang Cai. 2019. Practical and effective sandboxing for Linux containers. *Empirical Software Engineering* 24, 6 (2019), 4034–4070.

[77] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. 2012. Permission Evolution in the Android Ecosystem. In *Proc. Annual Computer Sec. Appl. Conf. (ACSAC)*. 31–40.

[78] whitesourcesoftware.com. [n.d.]. Renovate - Automated Dependency Updates. https://renovate.whitesourcesoftware.com/.

[79] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. 2016. A Look at the Dynamics of the JavaScript Package Ecosystem. In *Proc. Mining Software Repositories (MSR)*. 351–361.

[80] Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald Gall, and Massimiliano Di Penta. 2020. An Empirical Characterization of Bad Practices in Continuous Integration. *Empirical Software Engineering* 25, 2 (2020), 1095–1135.

[81] Qiang Zeng, Zhi Xin, Dinghao Wu, Peng Liu, and Bing Mao. 2014. *Tailored Application-specific System Call Tables*. Technical Report. Pennsylvania State University.

[82] Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesús González-Barahona. 2018. An Empirical Analysis of Technical Lag in npm Package Dependencies. In *Proc. Int'l Conf. Software Reuse (ICSR)*. 95–110.

[83] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *USENIX Security Symposium*. 995–1010.