# Stepping out of Bounds: Security Impact of Allowing Packages on npm to Declare External Dependencies

Dominic Tassio
University of Kansas
Lawrence, KS, USA
dominic.tassio@ku.edu

Elizabeth Wyss
University of Kansas
Lawrence, KS, USA
elizabethwyss@ku.edu

Gael Salazar-Morales
University of Kansas
Lawrence, KS, USA
gmorales034@ku.edu

Lorenzo De Carli
University of Calgary
Calgary, CA
lorenzo.decarli@ucalgary.ca

Drew Davidson
University of Kansas
Lawrence, KS, USA
drewdavidson@ku.edu

## ABSTRACT

In this paper, we explore an understudied feature of the npm package registry, allowing packages to specify a dependency that is served from a source that is external to (and outside the control of) npm. Although URL dependencies or external dependencies have been recognized as code smells in previous literature and current security tooling, there is a lack of analysis of external dependencies across the entirety of npm. Thus, we present our method for conducting a registry-wide npm analysis, describing special considerations that we made to account for the large size of the registry. To address those considerations, we analyze a cache of all npm packages. External dependencies were identified based on the type returned by the `npm-package-arg` package which is used by the npm cli. Our work characterizes the usage of external dependencies among all packages on npm, considering their popularity and potential security impact. We identify several notable attack vectors that do not exist for internal packages and are enabled when the package is served from an external location. In our usage analysis, we find that 0.41% of npm packages make use of an external dependency in their latest version. Of those packages, 372 packages have more than 350 weekly downloads, which we consider to be the lower threshold of usage by developers. Further, there are six packages that have over 100 thousand weekly downloads, which we consider to be significant usage by developers. Finally, we also discuss a number of mitigation steps that software developers and package registry maintainers may use to reduce the negative impact of external dependencies.

**ACM Reference Format:**
Dominic Tassio, Elizabeth Wyss, Gael Salazar-Morales, Lorenzo De Carli, and Drew Davidson. 2025. Stepping out of Bounds: Security Impact of Allowing Packages on npm to Declare External Dependencies. In *Proceedings of the 2025 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3733827.3765525


*SCORED '25, October 13–17, 2025, Taipei, Taiwan*

ACM ISBN 979-8-4007-1915-8/2025/10
https://doi.org/10.1145/3733827.3765525

## 1 INTRODUCTION

Modern software development increasingly relies on the discovery, procurement, and integration of *dependencies* - units of code that provide generic functionality to a codebase [10, 14, 42]. These dependencies form a major part of the *software supply chain*, allowing developers to conveniently avail themselves of a wealth of prebuilt infrastructure. Government, industry and academic entities have all noted that the software supply chain requires serious security examination [4, 8, 39, 41]. A particular area of study within the domain is the security of open-source package registries [15]. These package registries enable developers to easily share open-source dependencies to a large developer audience. Much of this work studies the methods and effects of uploading malicious packages to the registry, focusing primarily on the *integrity* aspect of the CIA triad. Our larger vision is to contribute a complementary line of analysis: a study that also considers the *availability* aspect of the CIA triad for package registries. To this end, we focus this work at the level of the design and policies of package registries. We look beyond the treatment of a single malicious behavior or attack vector. Rather, we consider whether the registry has the appropriate means to (1) provide reliable access to benign packages and (2) remediate malicious package code.

Our work is focused on the npm package registry for managing node.js dependencies. The npm registry provides an important target of study, as it is the largest package registry - both in terms of the number of packages that are listed in the registry and in the number of downloads. An underappreciated aspect of npm is that it is possible to create a package hosted within the registry itself (and thus forming a part of the software supply chain provided by npm), which declares a dependency outside of the registry itself. We refer to the former as an *internal* dependency, and the latter as an *external* dependency.

The unique design of npm, which mixes the availability of both kinds of dependencies, leads us to study the adoption and effect of external dependencies. External dependencies force npm to extend its root of trust beyond the boundaries of its own mediation: the registry becomes reliant upon the availability of 3rd-party infrastructure to serve code reliably and benignly, which may be underappreciated by developers that rely on the registry for availability.

Packages on npm are highly interconnected. If one package becomes unavailable or buggy, all packages that transitively depend upon it are exposed.

The effect of package interconnectivity has been demonstrated in several high-profile security incidents. In March 2016, a widely depended on npm package, `left-pad`, was unpublished from the registry by its author, resulting in widespread issues across npm-based web applications by preventing any packages that depended on `left-pad` from building [38]. As another example, in January 2022, the npm packages `colors` and `faker` were intentionally sabotaged by their author, impacting the many projects that depended on them [33]. These incidents demonstrate the impact of package interdependence, but the mitigations that were applied in these cases are also notable. In all cases, the npm maintainers restored the packages to previous archived versions from within the registry. This option would have been unavailable if these dependencies were external to the npm registry, since there would be no archived version available.

Motivated by these incidents, we set out to exhaustively study how external dependencies are used by npm packages in practice. We note that our focus is on the impact to the npm registry itself. Our interest is studying how external dependencies declared by packages on npm impact the health and security of npm. We do not study packages or applications that are not hosted on npm. These packages or applications do not impact the availability of npm packages themselves.

Our primary contribution is a registry-scale analysis of npm on the actual usage of external dependencies. We find that external dependencies are rarely used, implying that developers may be able to avoid depending upon them. We also note that because they are rare, developers may not be fully aware of their security impact. Our work describes software supply chain attack vectors that are enabled by external dependencies, and we demonstrate precisely how the design of a package registry can impact these threat vectors. Finally, we discuss protections for both package registry maintainers and developers on how to reduce the impact of external dependencies.

## 2 BACKGROUND

In this section, we provide the necessary background on the design and operation of npm and its package system.

The npm system consists of distinct, but related, components that enable the consumption and distribution of software packages [24]. These components are the npm website, the command line interface (CLI), and the package registry. Each component has a purpose in the user workflow of using npm. The primary points of contact for a user when installing or publishing a package are the website and CLI. When exploring packages, the website provides discovery through search functionality and presents basic information about the package. In addition, this information includes displaying the package readme file, versions, dependencies, dependents, and download information. The CLI presents a large surface area, consisting of many sub-commands and providing user access to much of the npm system. By default, the npm system uses the npm registry to resolve packages [28]. As of our analysis from April 2025, the npm registry reports more than 3.5 million packages.

When installing a package using the npm CLI, the package's dependencies are also installed with it. This process is recursive, leading to the installation of all transitive dependencies of the original package.

As points of comparison, we look at package systems in other programming language ecosystems. Unique among them is the Go programming language, in that it lacks a central registry and provides a standard way to specify dependencies using URLs. Most package registry systems prevent the ability for a package on the registry to depend on a package external to the registry. For example, the Rust ecosystem's package registry `crates.io` prevents packages from being published with external dependencies [3]. Similarly PyPI, the package registry for Python, also prevents packages from depending on external packages [1]. Both Go's and npm's usage of external dependencies present technical and security challenges that are not found in other package registries. After describing how external dependencies are supported, we will look at the mechanisms Go uses to mitigate these challenges.

### 2.1 Support for External Dependencies

Below, we describe the mechanism through which npm allows for packages to specify external dependencies. See figure 1 for an architecture of a package on npm specifying an external dependency.

**External Registry**: The npm registry maintains the list of its internally hosted packages, dubbed the package registry [28]. Users are able to specify external registries in a user-level `.npmrc` configuration file, thus causing packages to be fetched from the external registry [9]. However, this association only impacts the local user's development environment.

**Direct URL**: Another way that an npm package can specify a dependency is through an explicit URL [27]. The URL must resolve to a tarball containing an npm package or be a URL to a Git registry of a package. This deviates significantly to the standard method of specifying a version range.

**Implicit Domain**: A specific feature of npm dependency specifiers is the ability to set a GitHub registry as the source using a short form specifier. The short form is `<user>/<repository>` and will be treated as a Git URL to that repository on GitHub. Another short form allows specifying Git repositories on other hosts in addition to GitHub. By prefixing with "`bitbucket:`", "`github:`", or "`gitlab:`", a repository can be targeted that is hosted on BitBucket, GitHub, or GitLab respectively. We note that the form `<user>/<repository>` is very similar to how scoped packages are named on npm: `@<scope>/<package>`, where the scope refers to a user or organization on npm. This allows for multiple packages to have the same name if they are under different scopes. The npm cli tool allows installing packages using the `<user>/<repository>` short form instead of the name of the package on npm. This will install the package from the repository on GitHub. The entry in the resulting dependency field of the `package.json` will be `"<name>": "github:<user>/<repository>"` where the name will be what is given in the dependency's `package.json`. See listing 1 for an example of the GitHub short form specifier.

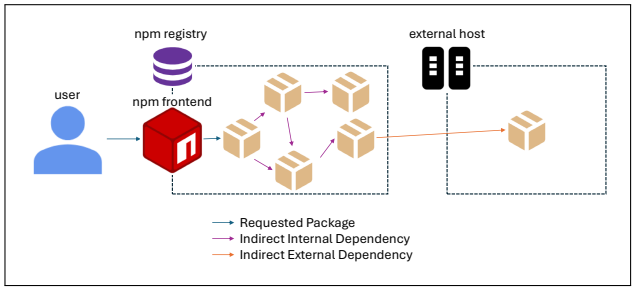


**Figure 1: Architecture of a user consuming a package from npm that depends on a package that is hosted externally by a third-party.**

```
"dependencies": {
    "ansi-escape-sequences": "^6.2.2",
    "debug": "^4.3.4",
    "eastasianwidth": "^0.2.0",
    "hash-arg": "^1.0.6",
    "node-getopt": "github:takamin/node-getopt"
}
```

**Listing 1: Example snippet of the package.json file from the `list-it` package on npm with an external dependency on the `node-getopt` package.**

## 2.2 Protections

As a point of comparison, we look towards Go. Since Go and npm both support direct URL dependencies, it is interesting to compare the protections in place against the threats posed by allowing the package registry to depend upon third-party web domains. Go has implemented two protections for the unique issues its package ecosystem faces by not having a central package registry [30].

**Package Checksum**: A checksum database stores a SHA-256 hash for a Go package which is used when downloading to verify the contents of a package. The checksum database stores for each package the package path, the package's version, and the computed hash of the package [2]. When downloading a package, the hash of the package is compared to the hash stored in the checksum database to ensure the package has not been altered [30].

**Package Mirror**: To guarantee the availability of packages, Go also provides a mirror of cached packages. When installing packages, the mirror is used instead of the direct source if the package is already present on the mirror. When a package is requested for the first time and has an open-source license, the package is cached to the mirror. All future requests of a package cached on the mirror are served from there, even in the case that the package is removed from its original source [30].

# 3 SECURITY IMPACT

In this section, we contribute descriptions of the security impacts that arise when external dependencies are supported by registry ecosystems.

We also reiterate that the scope of our study, and thus the description of security impact, is limited to the effects of external dependencies on packages that are present within the package registry itself. In particular, we note that the study of dependencies declared by *applications* is beyond the scope of this work. We believe that application developers may have good reason to declare the need for a package that is not part of the registry.

We note that external dependencies have been identified as a cause for concern in other systems [6, 18]. Both the external security scanning tools Sandworm and Socket.dev, two popular node package security scanners, mark any external dependency as a security issue [31, 35, 36]. Socket.dev ranks external dependencies as high severity security issues [35, 36].

We identify developer confusion, user targeting, package availability, and obscuring remediation. These correspond to the categories of "Create Name Confusion with Legitimate Package", "Distribute Malicious Version of Legitimate Package", "Compromise Build System", and "Prevent Update to Non-Vulnerable Version", respectively, from the taxonomy of attacks from Ladisa et al. [25] We will refer to this taxonomy as the attack taxonomy going forward. Packages on npm depending on external packages present unique security threats and new avenues for known security threats that may otherwise not be possible without external dependencies.

## 3.1 Taxonomy of External Packages

We detail some of the security implications of allowing packages within the registry to declare dependencies external to the registry. These dependencies provide a vector for a malicious actor to harm users of the registry package.

**Developer Confusion**: There are many previous works that detail the threat of *package confusion*, whereby a developer includes an unintended package due to a typo or other confusion that an attacker has taken advantage of to cause the developer to include malicious code [29, 37, 39]. This is classified under the attack taxonomy as "Create Name Confusion with Legitimate Package" [25].

Package confusion is a problem that is not exclusive to packages on a central registry. For example, a malicious Go package was recently removed, `boltdb-go/bolt`, which was intended to induce package confusion with the benign `boltdb/bolt` package. The confusion attack caused the wrongly-included code to grant persistent remote access to the malware author [11].

In particular, npm allows for notational shorthand to indicate that a particular external package source is being requested. As an example, consider the distinction between the external dependency shorthand `<user>/<repository>` versus the scoped dependency naming scheme `@<scope>/<package>`. The first indicates a package on GitHub and the second is a scoped package on npm. When using the npm cli tool to install a package, forgetting the '`@`' in front will cause npm to install an external dependency hosted on GitHub.

We consider the possibility of such confusion to be a plausible and accidental behavior. The omission of a single character falls within previous definitions of package confusion [29, 37]. Our analysis found 5,862 dependencies potentially installed where the '`@`' character was omitted. Of those dependencies, 449 correspond to a scoped package that currently exists on npm. It is possible that some of these packages were installed accidentally instead of the intended scoped package on npm. The majority of dependencies (3,691) have a corresponding scope that exists on npm without

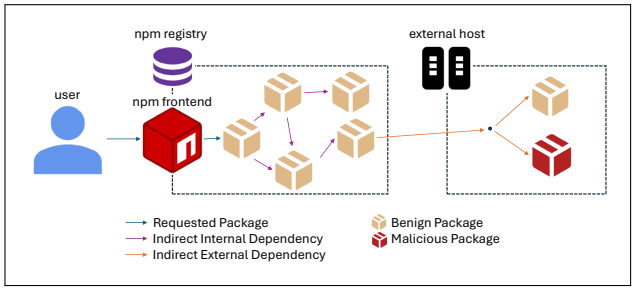


**Figure 2: Architecture for a user targeting attack using *differential serving*. The third-party external host can choose to serve a malicious version of a package instead of the benign version based on the user requesting the package.**

the package. Most concerningly, 1,722 dependencies do not have a corresponding scope or package on npm. A malicious actor would be able to claim the scope and package to enact a dependency confusion attack against a user attempting to install an external dependency. Thus, we find that the different methods of specifying packages may serve as a vector for attack.

**User Targeting**: External dependencies necessarily involve a 3rd-party host delivering a package. Most naturally, the host would act as a static package registry. However, the host may instead take dynamic action when a client attempts to fetch a package, even as it masquerades as a static service. We refer to this behavior as *differential serving*, as it allows for differential treatment of clients, according to the goals of the adversary. See Figure 2 for an attack architecture. Under the attack taxonomy this would be classified as "Distribute Malicious Version of Legitimate Package" [25]. An adversary may generally distribute a legitimate package, only to then use differential serving to target specific users and serve them a malicious version of the package.

In practice, such a host may choose to use a differential policy to target particular users with malware. For example, the host might serve a malicious codebase (perhaps a trojaned version of the legitimate package) to a targeted block of IP addresses. The desire to target a particular set of clients in this way has been observed in software supply chain attacks in the past–the `node-ipc` package was updated to corrupt the developer's filesystem if it was run from a Russian or Belarusian IP address [13]. This same malicious behavior could be achieved by serving benign code to non-Russian or Belarusian IP addresses.

Alternatively, the host might choose to use its differential behavior to conceal malicious behavior and delay detection. An adversary could use differential serving to serve a benign package to clients that are known to be security scanners or auditing tools in order to escape detection, while serving malware to all other clients.

**Package Availability**: An additional security impact of allowing external dependencies is that it increases the attack surface of the package ecosystem.When a package is hosted on a private server, it provides a target for a denial of service attack. We place this under "Compromise Build System" from the attack taxonomy due to potential impacts on build availability[25]. Should an adversary be able to take down a third-party external dependency host, it would break the installation process of any dependent package. As demonstrated by the `left-pad` incident, even a seemingly inconsequential package can lead to significant service outages [12].

An author may intentionally *unpublish* a package from npm. Unpublishing is subject to the npm's unpublishing policy. This policy disallows package unpublication if it has any dependent packages (amongst other criteria). An external dependency is effectively excused from such rules, as the host can simply refuse to serve the package at its own prerogative.

**Obscuring Remediation**: A key security feature of npm is the ability to remove code when it has been deemed malicious. This capability is employed in order to ensure a mandatory stoppage of the distribution of malicious code. In npm, the content of a package is replaced with a "security holding package", a code-free package whose metadata links to the security advisory that compelled the takedown. This approach has the advantage that any package that incidentally uses the affected package will at least still complete installation, though usage of any code from the removed package will fail.

When an external package is subject to a security advisory, there is significantly less control that can be exerted over the package contents. In the most basic sense, the package exists outside of npm and therefore its content cannot be replaced with a holding package. We place this under "Prevent Update to Non-Vulnerable Version" from the attack taxonomy.

The recent example of the `boltdb-go/bolt` malicious package, which induces package confusion against the benign `boltdb/bolt` package, is a representative example of this issue. The malicious version of `boltdb-go/bolt`, 1.3.1, was published to GitHub in November of 2021, and cached by the package mirror. Once cached, the malware authors rewrote the v1.3.1 Git release tag to point to a benign commit. This change allowed the malicious code to be served from the mirror, but an audit that simply followed the version number and analyzed the Git code would observe no malicious behavior.

## 4 REGISTRY SCALE ANALYSIS

Due to the unique feature of npm that allows packages on the central registry to specify external dependencies, and the lack of mitigating features employed by npm, we have chosen to conduct a registry-scale analysis of external dependencies across every package that has ever been uploaded to npm. Performing such an analysis is necessary in order to fully assess the state of the registry, but it requires overcoming a significant challenge of scalability. Not only are there 3,527,282 total packages on npm at the time of our work, packages may also have more than one version. For the sake of completeness, we choose to analyze every version of every package available on npm.

An additional challenge of our analysis is that we aim to minimize the impact on `npmjs.com`. We note that in prototyping the analysis, checking for various aspects of each package, and running a crawler over the entire registry is not insignificant.

In order to meet the scalability needs of our analysis while simultaneously reducing the load on `npmjs.com`, we construct a special private mirror of the entire npm registry. In recognition of the speed at which the registry grows, we design our system to maintain a

"living snapshot" of the registry that stores a copy of each new package and package version that is added to npm. A benefit of this is that if the package version is removed from npm, we retain that copy for future analysis.

Registry-scale analysis must contend with an important characteristic of real code. Namely, that much of the code on the registry is not used in practice. For example, it is both trivial and common to quickly create a free account and upload a test package to the registry. Even code with nontrivial functionality may become abandoned or be uploaded without any appreciable audience. A recent npm analysis deemed that only about 12.1% of packages on npm see actual use [37].

We note that any registry-scale analysis that does not account for this unused package phenomenon may be confounded by skewed aggregate results. For this reason, we modulate our analysis using a popularity threshold. We decide upon this threshold using npm's reported weekly download counts as a proxy for usage. As such, we regard any package with less than 350 weekly downloads as effectively unused, as has been done by several recent works in this space [29, 39]. Throughout our evaluation, we refer to any npm package above that threshold as *utilized*, and any (effectively unused) package below the threshold as *unutilized*.

We identified external dependencies based on their specifier listed in the `dependencies` object of a package's `package.json`. Taking advantage of the package used internally by the npm cli, we used `npm-package-arg` to determine the type of the specifier. In the cases where `npm-package-arg` returned the type as git or remote, were the cases in which we classified a dependency as external.

Our analysis used information on a package from three sources. First, a cached version of the metadata provided by the npm api at `https://registry.npmjs.org/<package_name>` to collect the latest version. Second, the `package.json` of each package version to obtain the list of declared dependencies. Third, the npm api for download counts.

## 5 EVALUATION

Our analysis leverages our private mirror of npm detailed in Section 4. We contribute an analysis on the prevalence of external dependencies on npm.

We find the usage of external dependencies by packages on npm is a rare phenomenon. Of the more than 4.7 million packages that we have cached from npm, 0.41% make use of external dependencies in their latest version. There are 0.99% of package versions that use an external dependency out of more than 44.5 million versions.

Despite the low percentage of packages that use external dependencies, we look to the six latest versions identified in Figure 4 that have over 100 thousand weekly downloads to highlight the risk posed to npm by external dependencies.

### 5.1 Prevalence Within the npm Registry

As described in Section 4, characterizing any aspect of a package registry requires careful analysis of which packages are being considered. Figure 3 breaks down all versions of all packages that have an external dependency. We analyze 44,507,170 package versions

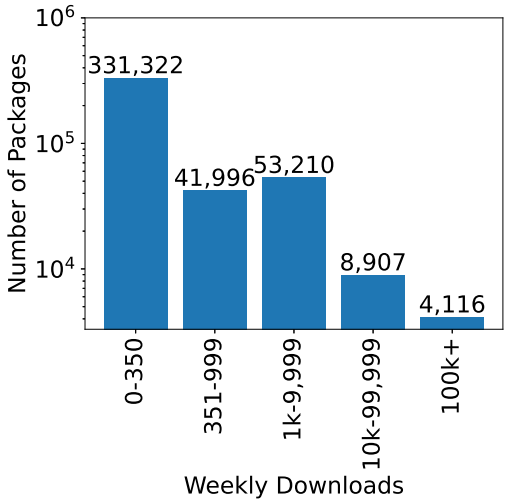


**Figure 3: Weekly downloads of all package versions with an external dependency**

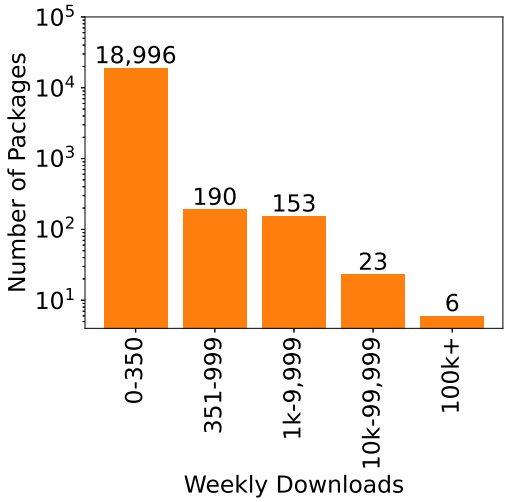


**Figure 4: Weekly downloads of the latest package versions with an external dependency**

and find that 439,551 (0.99%) specify an external dependency. Taking into account only the latest version of a package, Figure 4 shows a similar pattern of external dependency usage. We analyze the latest versions of 4,706,114 package versions and find 19,368 (0.41%) use at least one external dependency. We see that the vast majority of packages are downloaded 350 times or less a week. This remains the same whether looking at all versions of packages that have an external dependency or only the latest version of a package. We believe that these results provide evidence that external dependency use is rare.

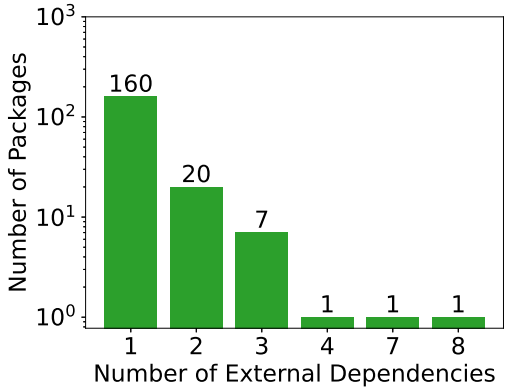


**Figure 5: Number of external dependencies used by the latest versions of packages on npm with greater than 350 weekly downloads. Of the 190 utilized packages with external dependencies, 84% have only 1 such dependency.**

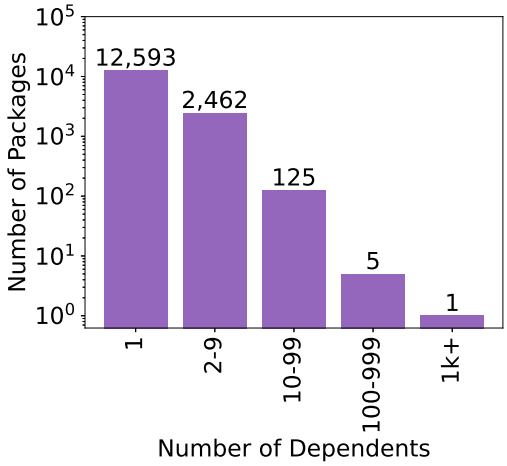


**Figure 6: Dependents of External Dependencies. Most external dependencies have only one internal npm user.**

Despite usage being generally rare, there are six packages with over 100,000 weekly downloads that pose a risk to the npm ecosystem. These packages are: @electron/rebuild, node-xlsx, volar-service-emmet, @axelar-network/axelar-cgp-sui, @whiskeysockets/baileys, and domotz-remote-pawn. Between all six of these packages, there are 105 direct dependents and 201 indirect dependents. Due to the transitive nature of package dependencies on npm, the risks identified in Section 3 now impact 306 other packages on npm outside of the six packages that directly use external dependencies.

## 5.2 Prevalence within Individual Packages on npm

While the overall number of packages that use external dependencies is low, we additionally consider the degree to which *utilized* packages (>350 weekly downloads) employ external dependencies. Our analysis identifies only 190 *utilized* packages from the latest versions that use external dependencies, and Figure 5 provides a breakdown of the 190 packages. We find that 160 (84%) of the *utilized* packages only specify one external dependency, and the maximum number of declared external dependencies by a single package is eight (of which there is only one such package).

Since most of the external dependencies serve only a single internal package, they are functionally much closer to a particular module of that internal package. As such, these particular external dependencies could instead be bundled to avoid the security issues associated with external dependencies.

## 5.3 Characteristics of External Dependencies

In addition to exploring how widespread the use of external dependencies is from within npm, we are also interested in capturing information about those external packages themselves. The non-standard presentation of external packages makes them difficult to profile, especially in terms of download count. Given the lack of a consistent statistic, such as the weekly download count provided by npm, we instead use the number of npm packages that use a particular external dependency as a metric for its use.

| Package | Dependents |
|---|---|
| github:libphamton/client-fb | 1796 |
| github:uNetworking/uWebSockets.js | 170 |
| github:daemonsec637/dotgov-list-node | 163 |
| github:daemonsec637/eks-auto-create-idp | 125 |
| github:gulpjs/gulp | 121 |
| github:bitpay/node-buffers | 100 |

**Table 1: Number of Dependents for the Most-Used External Packages. All packages with greater than 100 dependents were selected.**

Figure 6 characterizes the degree of use that an external package receives. As described above, we find that of the latest versions, 19,368 packages use at least one external dependency. Furthermore, we find that there are a total of 15,186 distinct external dependencies that are referenced from packages on npm. Of these distinct external dependencies, 12,593 or 83% are used by only one npm package. This result seems to indicate that for the cases in which a package exists outside of npm, its adoption is quite low.

It is of note that there are six external packages that are each listed as dependencies for over 100 packages. These packages are shown in Table 1. The top package seems to be a clone of the npm package `@libphamton/chatfanpage` and looks to take advantage of the *tea* crypto rewards program [34]. The second package, `uNetworking/uWebSockets.js`, takes an ideological stance against being hosted on npm [7, 19, 20]. The third and fourth packages from the user `daemonsec637`, are no longer available on GitHub. The fifth package is available on npm and does not need to be installed as an external dependency. The last package is a clone of the `buffers` package on npm.

In a random sample of 11 packages that depend on the top two external dependencies, none were published more recently than 2021, indicating that their dependencies are likely abandoned.

Of the 15,186 total unique dependencies, we find that 13,552 (89%) use the git protocol (see Section 2 for more information). The remaining 1,634 dependencies fetch a file over http. The majority of these point to a package on GitHub (13,253), either through a git url, the GitHub short form, or a direct url to GitHub.

Not only is the adoption of external dependencies rare, but the packages that do declare external dependencies overwhelmingly target `github.com`. Since GitHub is the owner of npm, this represents little actual decentralization in practice.

## 6 DISCUSSION

Above, we explored security problems with external dependencies and detailed our experiments characterizing the actual usage of the feature. Based on our findings, we contribute recommendations to package registry maintainers and package developers for mitigation strategies to address the issues raised by external dependencies.

### 6.1 Registry Recommendations

Based on our findings regarding the security implications of external dependencies, we have arrived at three recommendations that registry maintainers could adopt. We note that these recommendations serve to blunt the security impact of misuse of external dependencies. However, we take a "do no harm" stance towards security here - proposed changes must not break the build of any legitimate package, nor should they forbid the use of external packages. Rather, they should seek to align the transitive use of external packages with the functionality of packages hosted within the registry. Our first recommendation, for npm to implement an external dependency checksum database, is intended to prevent the problems that we described in Section 3. Our second recommendation, to deprecate unused packages that currently use external dependencies, is designed to caution and dissuade future use of these packages. Our third recommendation is to add a confirmation step when installing an external dependency, is intended to prevent the confusion problem discussed in Section 2.1 and Section 3.1.

**Implementing a Checksum Database**: Our primary recommendation is the addition of a checksum database, similar to the one provided by Go, for all existing and any future external dependencies to be cataloged in this database. This recommendation is based on our finding that several new security threat vectors are enabled by external dependencies. When downloading an external dependency, a checksum will be created for the package to verify against the checksum database for external packages. If the checksum is not found in the database, it will be added. Any subsequent lookup of a checksum will compare to the originally calculated checksum. By comparing these checksums, it would be possible to defeat differential serving and user targeting attacks, as any alteration of the package would be detected when its checksum does not match the one stored in the database.

**Deprecating Unused Packages with External Dependencies**: Our analysis shows that the use of external dependencies tends to cause instability in a package. Several of the most-depended-upon external dependencies across all npm packages are entirely unavailable, causing any package that transitively depends upon them to fail its installation procedure because the external dependency cannot be fetched. This situation implies that many packages using external dependencies are not maintained, and that as external dependencies fall victim to bitrot, their dependents do not take appropriate ameliorative action.

Overall, the existence of unmaintained packages is not surprising. Indeed, our registry-scale analysis estimates that nearly 95% of all packages are effectively unused. As such, flagging those packages that are particularly likely to break a transitive build is a worthwhile endeavor. We recommend that registry maintainers flag such packages to provide indications to potential users that the package they are considering including may be particularly subject to bitrot. A conservative metric to identify such packages is to flag any package with no dependents, less than 350 weekly downloads, and an external dependency. We do *not* recommend that such packages be taken down, as they do not necessarily exhibit malicious behavior, nor violations of any terms of service. Indeed, such packages may be dependencies of application code, for which the registry cannot account (since they are not part of the registry itself). Instead, we suggest that registry maintainers mark such packages as deprecated, as it is of increased likelihood to break any build that uses it as a transitive dependency.

We suggest the metrics and thresholds above because they are inspired by npm's own policy for unpublishing a package from the registry [23]. When a package meets those criteria, it is considered ancillary to the operation of the npm infrastructure. We further note that, like many other deprecation decisions, a package developer can chose to update their package to avoid the notice or simply ignore the warning with no penalty in functionality.

**Require Confirmation**: A user is most susceptible to the potential confusion presented in Section 2.1 and Section 3.1 when installing dependencies using `npm install`. In the common case of installing a scoped package, if the user forgets to type the leading '`@`', then it is possible that they would unknowingly install an external dependency. To prevent this, we recommend introducing a confirmation step when installing an external dependency using the `npm install` command. This step should warn the user that they are about to install an external dependency and in the case that the scoped package does exist on npm, should indicate that the package name should be entered with a leading '`@`'. This should greatly reduce the risk of a user mistakenly installing an external dependency when they intended to install a scoped package.

### 6.2 Developer Mitigations

Our view is that external dependencies incur unique threats to developers that rely on them. However, we note that many of the worst outcomes from this feature can be mitigated through careful package development. Below, we describe several features that developers should consider as they target dependencies.

**Package Bundling**: The primary mitigation step that we encourage any package developer who must rely on an external dependency to deploy is a feature provided by the npm protocol. This feature, called *package bundling*, allows a package to copy all of its dependency code into its tarball distribution. In this way, the package is robust to package availability issues if the external dependency is taken down, trojaned, or altered to serve malware post-facto from an external url. Bundling comes with its own set of additional burdens, as any update to the external package will

not be propagated to the internal consumer. Nevertheless, bundling is highly targeted: the developer may choose to specify that some subset of their dependencies (such as the external dependencies) should be bundled, while the rest of the dependencies should be fetched at install time.

**Internal Clone Packages**: For the sake of completeness, we note that a developer could choose to fork an external dependency and upload their fork to the registry. We do not advocate for cloning a package without providing a functional improvement, as doing so may take credit from the original package author. However, we note that many external dependencies seem to be non-functional code bases that presumably can be improved upon by those developers that rely upon them. Furthermore, many of the packages we found outside of npm use permissive licensing that allow for forking, and additionally, recent evidence suggests that such cloning is a common practice in npm [39].

## 7 RELATED WORK

While a great deal of work has focused on the security of the software supply chain in general, and on the security of npm in particular, we believe that the security impact and adoption of external dependencies as they relate to supply chain security at a registry-scale is under-studied. We describe some of the work most closely related to our own below.

**Registry-Scale Analysis**: As described in Section 4, characterizing an entire package ecosystem is challenging due to the scale of the largest package registries. Several previous works have provided registry-scale analyses, though they do so with different goals. Taylor et. al. present a novel method of typosquatting detection across the entire npm package set [37]. Whereas their work compared multiple packages by metadata within the registry, our work analyzes packages within the registry for their relationships to dependencies outside the registry. Wyss et. al. present a tool for detecting cloned packages within all packages available on npm [39]. Like our work, they operate at registry scale using a mirror-based analysis but use it for a comparative analysis between packages.

**Package Confusion**: Previous works have identified the threat of package confusion, i.e. the threat of an application developer or package developer including the wrong package. Package confusion may be encouraged by a malicious actor or may cause accidental harms if the incorrect package is confused incidentally for the correct one [22, 25, 29, 37, 40–42]. Our work extends this line of work by noting that external dependencies can be an exacerbating factor in package confusion, since the registry maintainer has less control over takedowns of malicious packages. Furthermore, the various means by which external packages can be specified present another way that package confusion may occur.

**Dependency Smells**: Jafari et. al. characterizes several dependency smells related to npm packages, including the use of external dependencies [21]. Where a dependency smell is an issue pertaining to the management and development of software. We expand the security impact of external dependencies Jafari et. al describe, identifying more attack vectors uniquely enabled by external dependencies. We also provide analysis of the entirety of packages on npm, covering a vastly larger dataset of packages.

**Security of npm**: The security of package managers, npm specifically, has been the subject of a large collection of works. Here, we discuss some examples. Duan et. al. present a comparative framework for analyzing package managers, including npm, and provide an analysis pipeline to identify malicious packages [15]. The health and security of the npm system relies on developers to follow best practices when configuring their packages and applications, Kabir et. al. investigates how these practices are followed [22]. How these vulnerabilities in packages on npm manifest in applications has been studied by Alfadel et. al. [8].

**Other Ecosystem Solutions:** Google's Go programming language features a unique method of package management. All dependencies are initially decentralized and not found in a central registry [6]. The Go team has added a mirroring and indexing service to their package management system [5]. This approach was analyzed by Hu et. al. in their paper covering how vulnerabilities are addresses in Go's ecosystem [18]. Ultimately, the security mechanisms of Go could serve as mitigation against some of the threats we describe in npm.

**Threat Vectors**: A great deal of work has described some of the threats that may be launched via the software supply chain, including npm. These threats include research studies which detail the importance of supply chain security and various forms of attack against them [12, 15–17, 42]. Package-based supply chain security has also been the subject of a number of popular news stories [32, 33, 38]. These works show the reality that packages can be subverted (and are) attacked in practice, motivating our stance that extending the root of trust outside of the registry's domain has negative security implications. Finally, we note that the ability for a webserver to target particular segments of users, as pointed out in Section 3, has been well-studied. A representative work is Mansoori et al. which describes several means by which a site may distinguish various traffic populations [26].

## 8 CONCLUSION

In this paper, we study the impact of allowing packages on npm to depend on external dependencies. This feature allows a package to specify a dependency that is served from a url outside the control of npm. Our study characterized both the security impact and the actual incidence of external dependencies.

On the security front, we observe that several attack vectors that do not exist for internal package dependencies are enabled when a package is served from another location. These threats include taking advantage of developer confusion, changing the code that is served to different clients (allowing targeted attacks on particular developer populations), and evading auditing. Even if the maintainer of an external dependency is benign, they may become the target of denial-of-service attacks in which they are unable to avail themselves of the resources of the registry's web infrastructure.

On the usage front, we find that the prevalence of external dependencies is rare in practice. Overall, the proportion of packages using external dependencies at all is quite scarce. Furthermore, more popular packages tend to avoid the use of external dependencies. Those packages that do use external dependencies use them sparingly, and

that many of the most popular external dependencies are entirely non-functional.

In summary, we conclude that there are potential security impacts of allowing for external dependency usage on a package registry. Given the infrequency of external dependencies, we note that the security implications of their use may be underappreciated. We also discuss a number of mitigation steps that can reduce the security impact of external dependencies, particularly when used by open-source package registries. These mitigation steps can help to make the registry less vulnerable to the misuse of external dependencies. We also discuss steps that a developer transitioning away from external dependencies may use to reduce their negative impact.

## REFERENCES


[1] [n. d.]. Cannot upload with external dependency due to "Invalid value for requires_dist" · Issue #9404 · pypi/warehouse. https://github.com/pypi/warehouse/issues/9404

[2] [n. d.]. Go Modules Reference - The Go Programming Language. https://go.dev/ref/mod

[3] [n. d.]. Specifying Dependencies - The Cargo Book. https://doc.rust-lang.org/cargo/reference/specifying-dependencies.html

[4] 2021. Exec. Order No. 14028. https://www.federalregister.gov/documents/2021/05/17/2021-10460/improving-the-nations-cybersecurity

[5] 2024. Go Module Mirror, Index, and Checksum Database. https://proxy.golang.org/

[6] 2024. Managing Depdendencies - The Go Programming Language. https://go.dev/doc/modules/managing-dependencies

[7] 2024. uNetworking/uWebSocket.js: uWebSockets for Node.js back-ends. https://github.com/uNetworking/uWebSockets.js

[8] Mahmoud Alfadel, Diego Elias Costa, Mouafak Mokhallalati, Emad Shihab, and Bram Adams. 2020. On the Threat of npm Vulnerable Dependencies in Node.js Applications. doi:10.48550/arXiv.2009.09019 arXiv:2009.09019.

[9] Roberto Basile, Andrew, and Luke Karrys. 2024. scope | npm Docs. https://docs.npmjs.com/cli/v11/using-npm/scope

[10] Ethan Bommarito and Michael Bommarito. 2019. An Empirical Analysis of the Python Package Index (PyPI). arXiv:1907.11073 [cs.SE] https://arxiv.org/abs/1907.11073

[11] Kirill Boychenko. 205. Go Supply Chain Attack: Malicious Package Exploits Go Module... https://socket.dev/blog/malicious-package-exploits-go-module-proxy-caching-for-persistence

[12] Md Atique Reza Chowdhury, Rabe Abdalkareem, Emad Shihab, and Bram Adams. 2021. On the untriviality of trivial packages: An empirical study of npm javascript packages. *IEEE Transactions on Software Engineering* 48, 8 (2021), 2695–2708.

[13] Joseph Cox. 2022. Open Source Maintainer Sabotages Code to Wipe Russian, Belarusian Computers. https://www.vice.com/en/article/open-source-sabotage-node-ipc-wipe-russia-belraus-computers/

[14] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2017. An Empirical Comparison of Dependency Network Evolution in Seven Software Packaging Ecosystems. arXiv:1710.04936 [cs.SE] https://arxiv.org/abs/1710.04936

[15] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2020. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. doi:10.48550/arXiv.2002.01139 arXiv:2002.01139.

[16] Robert J. Ellison, John Goodenough, Charles B. Weinstock, and Carol C. Woody. 2010. Evaluating and Mitigating Software Supply Chain Security Risks. (2010), 1158599 Bytes. doi:10.1184/R1/6573497.V1

[17] Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kastner. 2021. Containing Malicious Package Updates in npm with a Lightweight Permission System. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, Madrid, ES, 1334–1346. doi:10.1109/ICSE43902.2021.00121

[18] Jinchang Hu, Lyuye Zhang, Chengwei Liu, Sen Yang, Song Huang, and Yang Liu. 2024. Empirical Analysis of Vulnerabilities Life Cycle in Golang Ecosystem. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 212, 13 pages. doi:10.1145/3597503.3639230

[19] Alex Hultman. 2024. Addressing Claims of Conspiracy to Commit Crimes under the Computer Fraud and Abuse Act (CFAA). https://github.com/uNetworking/uWebSockets.js/blob/master/misc/npm.md

[20] Alex Hultman. 2024. uWebSockets.js — beware of slander, envious hypocrisy & Manuel Astudillo. https://unetworkingab.medium.com/beware-of-tin-foil-hattery-f738b620468c

[21] Abbas Javan Jafari, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, and Nikolaos Tsantalis. 2022. Dependency Smells in JavaScript Projects. *IEEE Transactions on Software Engineering* 48, 10 (Oct. 2022), 3790–3807. doi:10.1109/tse.2021.3106247

[22] Md Mahir Asef Kabir, Ying Wang, Danfeng Yao, and Na Meng. 2022. How Do Developers Follow Security-Relevant Best Practices When Using NPM Packages?. In *2022 IEEE Secure Development Conference (SecDev)*. IEEE, Atlanta, GA, USA, 77–83. doi:10.1109/SecDev53368.2022.00027

[23] Luke Karrys, Mayank Pathela, Michael Rienstra, Edward Thomson, Seryozha Khachatryan, Talgat Sarybaev, and Demira. 2024. Npm | Unpublishing packages from the registry. https://docs.npmjs.com/unpublishing-packages-from-the-registry

[24] Luke Karrys, Michael Rienstra, Myles Borins, and Edward Thomson. 2024. About npm | npm Docs. https://docs.npmjs.com/about-npm

[25] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. 2023. SoK: Taxonomy of Attacks on Open-Source Software Supply Chains. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 1509–1526. doi:10.1109/SP46215.2023.10179304

[26] Masood Mansoori and Ian Welch. 2019. Geolocation Tracking and Cloaking of Malicious Web Sites. In *2019 IEEE 44th Conference on Local Computer Networks (LCN)*. IEEE, Osnabrueck, Germany, 274–281. doi:10.1109/LCN44214.2019.8990794

[27] Mike McCready, Kyle Mitchell, Santoshraj2, Mottle, Hong Xu, s100, Uiolee, Christian Oliff, , Daniel Kaplan, Jan Sott, Gar, Francesco Sardone, P-Chan, Davide, Darryl Tec, Rohan Mukherjee, and Luke Karrys. 2024. package.json | npm Docs. https://docs.npmjs.com/cli/v10/configuring-npm/package-json

[28] Eric Mutta, Nathan Fritz, and Luke Karrys. 2024. registry | npm Docs. https://docs.npmjs.com/cli/v10/using-npm/registry

[29] Shradha Neupane, Grant Holmes, Elizabeth Wyss, Drew Davidson, and Lorenzo De Carli. 2023. Beyond Typosquatting: An In-depth Look at Package Confusion. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 3439–3456. https://www.usenix.org/conference/usenixsecurity23/presentation/neupane

[30] Julie Qiu and Roger Ng. [n. d.]. Supply chain security for Go, Part 2: Compromised dependencies. https://security.googleblog.com/2023/06/supply-chain-security-for-go-part-2.html

[31] sandworm_alerts 2024. Issue Types | Sandworm. https://docs.sandworm.dev/audit/issue-types

[32] Ax Sharma. 2022. BIG sabotage: Famous npm package deletes files to protest Ukraine war. https://www.bleepingcomputer.com/news/security/big-sabotage-famous-npm-package-deletes-files-to-protest-ukraine-war/ https://www.bleepingcomputer.com/news/security/big-sabotage-famous-npm-package-deletes-files-to-protest-ukraine-war/.

[33] Ax Sharma. 2022. npm libraries 'colors' and 'faker' sabotaged in protest by their maintainer — What to do now? https://www.sonatype.com/blog/npm-libraries-colors-and-faker-sabotaged-in-protest-by-their-maintainer-what-to-do-now

[34] Ax Sharma. 2024. Devs flood npm with 15,000 packages to reward themselves with Tea 'tokens'. https://www.sonatype.com/blog/devs-flood-npm-with-10000-packages-to-reward-themselves-with-tea-tokens

[35] socket_git_dep 2024. Git dependency - Alert - Socket. https://socket.dev/alerts/gitDependency

[36] socket_http_dep 2024. HTTP dependency - Alert - Socket. https://socket.dev/alerts/httpDependency

[37] Matthew Taylor, Ruturaj Vaidya, Drew Davidson, Lorenzo De Carli, and Vaibhav Rastogi. 2020. Defending Against Package Typosquatting. In *Network and System Security: 14th International Conference, NSS 2020, Melbourne, VIC, Australia, November 25–27, 2020, Proceedings* (Melbourne, VIC, Australia). Springer-Verlag, Berlin, Heidelberg, 112–131. doi:10.1007/978-3-030-65745-1_7

[38] Chris Williams. 2016. How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript. https://www.theregister.com/2016/03/23/npm_left_pad_chaos/ https://www.theregister.com/2016/03/23/npm_left_pad_chaos/.

[39] Elizabeth Wyss, Lorenzo De Carli, and Drew Davidson. 2022. What the fork?: finding hidden code clones in npm. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh Pennsylvania, 2415–2426. doi:10.1145/3510003.3510168

[40] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, and Laurie Williams. 2022. What are weak links in the npm supply chain?. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice* (Pittsburgh, Pennsylvania) *(ICSE-SEIP '22)*. Association for Computing Machinery, New York, NY, USA, 331–340. doi:10.1145/3510457.3513044

[41] Ahmed Zerouali, Tom Mens, Alexandre Decan, and Coen De Roover. 2022. On the impact of security vulnerabilities in the npm and RubyGems dependency networks. *Empirical Software Engineering* 27, 5 (2022), 107.

[42] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Smallworld with high risks: a study of security threats in the npm ecosystem. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(SEC'19)*. USENIX Association, USA, 995–1010.