What's in a URL? An Analysis of Hardcoded URLs in npm Packages

Elizabeth Wyss
University of Kansas
Lawrence, United States
ElizabethWyss@ku.edu

Drew Davidson
University of Kansas
Lawrence, United States
DrewDavidson@ku.edu

Lorenzo De Carli
University of Calgary
Calgary, Canada
Lorenzo.DeCarli@ucalgary.ca

Abstract

npm, a package manager for code written for Node.js, is a core component of the software supply chain, and a critical piece of software infrastructure. By selecting and importing code using npm, software developers can quickly build complex software products that would be extremely expensive to develop from scratch. However, "with great power comes great responsibility", and the npm ecosystem is a frequent target for attacks, which aim at injecting malicious code in packages.

In this paper, we take a look at an understudied internet-based attack surface in npm packages: URLs hardcoded within package code. The presence of such URLs—while often necessary—create risks as package behavior may dependend on data - and even code - retrieved from online endpoints. Unless care is taken to ensure such endpoints remain under control of the package authors, package functionality and security may at later point be compromised.

Our analysis of the presence of URLs in the npm ecosystem reveals that problematic URL usage is a present threat, albeit one which is primarily localized within unpopular packages that are infrequently maintained.

CCS Concepts

• **Software and its engineering → Open source model**; • **Security and privacy →** *Web application security*; • **Human-centered computing →** *Open source software.*

Keywords

npm, url, open-source, software-supply-chain

ACM Reference Format:
Elizabeth Wyss, Drew Davidson, and Lorenzo De Carli. 2024. What's in a URL? An Analysis of Hardcoded URLs in npm Packages. In *Proceedings of the 2024 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3689944.3696168

1 Introduction

npm [4] is a package registry for JavaScript, and one of the largest language-based software ecosystems available online. Many developers rely on it to integrate third-party libraries and tools into their projects. Together with similar registries for other languages (e.g., PyPI for Python, RubyGems for Ruby), it constitutes a key component of the open source software supply chain: a set of ecosystems allowing developers to build software quickly by integrating pre-existing functionality. This style of compositional software development has significant economic benefits, but also problematic security implications [24, 29]. These involve supply chain attacks, where threat actors inject malicious code into the dependencies of popular packages [24]; and more generally, the presence of vulnerable code that may be exploitable [12, 26]. Furthermore, npm packages typically define large dependency trees as they themselves use functionality from other packages. Thus, the impact of a compromised or vulnerable package can have far-reaching effects [12].

In this paper, we look at an attack surface within npm packages, which has rarely received scrutiny: that of packages utilizing hardcoded URLs within their code. "Hardcoded URLs" here refers to packages sending and/or receiving data or code from URLs which are statically defined within the code itself. This may entail posting to/receiving data from hardcoded URLs, or even pulling and executing code from online sources. Such practices, while often necessary to interact with internet endpoints (e.g. web service APIs), introduce security risks, as malicious actors can commandeer and/or exploit hardcoded URLs to inject malicious payloads or redirect users to phishing websites. Without proper validation mechanisms, developers may unknowingly introduce vulnerabilities into their applications. The core issue is that hardcoded URLs expose npm code to the age-old internet issue of link rot [30]: online content may become inaccessible, or be compromised. Worse still, an expired domain may become available and be acquired by questionable or malicious actors, who can then control the content accessed by a package [19]. Assessing the presence of URLs within open source npm packages is thus a relevant source of threat intelligence, and can assist in formulating secure coding guidelines.

Performing such an analysis at a meaningful scale requires analyzing the entire npm ecosystem for package URLs. Logistically, this is a non-trivial task, as at the time of analysis (Spring 2023), the npm registry consisted of $> 2M$ packages, whose code must be obtained and scanned. Ideally, the context where URLs appear should also be taken into account and further analyzed, to weed out false positives and gain broader understanding of why static URLs may be used. Indeed, while certain uses pose obvious security risks (e.g., steering the code execution based on content from external URLs), others (e.g., placeholder example URLs) may be acceptable.

We address the scalability challenge by setting up and maintaining, over the course of several months, an *npm observatory*: a mirror of the main npm registry that provides a local clone of *all*

ACM ISBN 979-8-4007-1240-1/24/10
https://doi.org/10.1145/3689944.3696168

packages available of npm, and incorporates updates on the main npm registry so that it remains a faithful copy. This allows us to efficiently analyze code for all 2M+ npm packages, without creating undue burden of the npm infrastructure. We investigate context surrounding URL usage, and functional issues by performing manual analysis of selected representative samples.

Results suggest that most URLs appearing in package code present limited security risks; however, the presence of problematic URLs increases in less popular packages, including use of URLs pointing to broken links and/or expired domains. We also find evidence of use of basic HTTP authentications, even including credentials embedded within plaintext URLs.

By undertaking this measurement study, we aim to provide insights into the prevalence and implications of hardcoded URLs in npm packages and identify problematic practices affecting the reliability, security, and maintainability of JavaScript applications. Our findings contribute to a more comprehensive understanding of software supply chain dependencies on the internet ecosystem.

The rest of this paper is structured as follows. Section 2 provides background on npm and relevant security issues. Section 3 discusses related work. Section 4 presents methodology and dataset, Section 5 presents our measurement results, and Section 6 discuss their implications. Finally, Section 7 concludes the paper.

2 Background

2.1 npm ecosystem

The npm archive is one of the largest software ecosystems in the world, counting as of 2023 over 2M packages. These packages implement a wide variety of functionality, and span a wide range of popularity and quality.

Many npm packages do not work in isolation. Instead, they require functionality from other packages in the ecosystem, and thus depend on them. In npm, such reuse is very common [18]. Thus, packages tend to transitively define large dependency trees that span hundreds of packages across many layers of dependencies [29]. While this approach leads to faster implementation of complex software, it also causes security issues and vulnerabilities to propagate across the ecosystem. Dependency code, once imported into a package, becomes a part of its attack surface [12].

Such issues are part of the more general problem of open source *software supply chain security*: as a software package depends on many others, it becomes easier for attackers to target a high-value software project, by injecting malicious code into vulnerable dependencies. The type of attacks that can be conducted this way have been discussed elsewhere in literature [24, 29] but, briefly, they may entail exfiltration of sensitive data [20], execution of unwanted operations such as cryptomining [6], and others. Even without explicit attacks, code written following poor programming practices may generate vulnerabilities exploitable at ecosystem scale [12].

2.2 Issue under study and threat model

The use of hardcoded URLs in package source code generates potential security issues. First, if a package's flow of execution depends on external content, it becomes impossible to statically determine whether executing the package may result in an undesirable outcome. Second, there is no guarantee that a URL will remain under ownership or control of a benign author indefinitely. If the content from a URL influences—directly or indirectly—package behavior, the URL owner may gain considerable power on that package's use. Indeed, we found a few example packages (such as `goindexmod666647@1.0.1`, discussed in Section 6) which use URLs to download executable code. Even if URLs are hardcoded to benign functionality (e.g., interface to a remote database, transmit installation counts, etc.), a threat actor may replace or alter that functionality by taking over the URL. Such a takeover may be as simple as waiting for the URL domain registration to expire, and purchasing it [9]. We emphasize that domain takeover is not just a theoretical issue; as Lauinger et al. point out, it is a somewhat common practice, with negative security implications [19].

Threat model: we consider an adversary who intends to broadly cause a package to perform an unintended action. To do so, the adversary may try to take control of the content served by URL(s) referenced within the package. By doing so, the adversary may cause the package to fetch incorrect content, altering the package execution and/or storing attacker-controlled content on the victim machine. The attacker may also capture data transmitted during package executions. Our goal is to measure the extent to which the above threat exists in the npm ecosystem at large. To do so, we measure the presence of hardcoded URLs, and we perform an in-depth analysis of select identified samples.

One threat we *do not* consider is that of an attacker uploading a malicious packages with URLs pointing to know malware or exploit download links. These are non relevant to our goal, which is to identify possible risks from benign use of URLs. Typically, malicious URLs will be obfuscated or composed dynamically at run-time to avoid detection; if unobfuscated, they may be detected by matching against URL blocklists (e.g., [10, 25]).

3 Related Work

Software supply chain security. Problems in supply chain security stem both from vulnerabilities, or malicious code. Malicious code may have several different purposes, and may be injected in software projects via different methods. Several works have analyzed and categorized common attack strategies in this domain [13, 24, 29]. Other works focus instead on detecting and identifying malicious packages. Sejfyia and Scäfer introduced Amalfi, a classifier which flags possible malicious code based on a number of package features [22]. Similar approaches were investigated by Vu et al. in the context of the PyPI ecosystem for Python [14]. Froh et al. [15] propose statistical analysis to flag suspicious package updates.

Beyond detection, previous research has also focused on containment of attacks. Wyss et al. proposed to contain install-time attacks via enforcement of user-defined policies [28]; Christou et al. propose a similar permission system for native JavaScript libraries [8]. At higher level, Project Sigstore [1] enable code-signing at scale for provenance verification, while Hassanshahi et al. [16] propose a logic framework for running provenance queries using rich package metadata.

Overall Number of Packages	2,154,700
Overall no. of Packages w/ URLs	486,558
Overall no. of URLs (w/ duplicates	6,898,948
Overall number of unique URLs	1,834,110

Table 1: Package and URL datasets

Our work complements and extends the existing body of knowledge, as to the best of our knowledge, prevalence and risks related to embedded URLs have not been investigated.

URL and link decay. The fact that links may cease to function, or point to content different than what was originally intended, is a well-know, long-running issue affecting the web. This issue is oftentimes referred to "linkrot". Zittrain et al. [30] investigated links appearing in articles from the New York Times from 1996 to 2019, discovering significant linkrot. Lauinger et al. [19] measured domain re-registration after expiration, a practice often performed by questionable parties that may result in misleading users into accessing unwanted content. Part of our work investigates the prevalence of this type of issue in package-embedded URLs. There is also anecdotal evidence that domain expiration issues affect the domains used by package maintainers to run and manage their projects (e.g. email servers) [9]; however, this issue is orthogonal to the one investigated in this paper.

4 Methodology and Data

In this section, we present the overall design of our measurement study on URL use in npm packages.

4.1 Methodology

Figure 1 presents our analysis pipeline. We deploy an *npm observatory* - a mirror of the main npm package registry (based on CouchDB [2]), which stores all npm packages and mirrors any update to the main registry[1]. The observatory maintains consistency with npm via an update observer, based on npm's Public Registry API [5]. For the purpose of this work, we generated a snapshot of the package database in April 2023, resulting in the dataset described in Section 4.2 (step **1**). We also collect package popularity data, which we used for our analysis, via npm's public-facing API which gives weekly download counts.

In order to identify package URLs, we scan package code files (e.g. *.js, *.ts, *.ejs, *.mjs, *.cjs) using a regular expression-based approximation of URL format. To limit false positives, we filtered out comments prior to parsing, and performed data cleanup using basic sanity checking (e.g. ensuring that URLs declare a valid URI scheme and contain a valid domain or IP address) (step **2**). Finally, we sent all detected URLs to our analysis pipeline, discussed in Section 5 (step **3**). While most of the analyses were accomplished via custom parsing scripts, some required investigating characteristics and health of URL domains. For those, we used `whois` queries [11] (limitations of this approach are discussed in Section 5.2.2).

URL	Count (%)
http://www.w3.org/2000/svg	93,277 (1.35%)
http://www.w3.org/1999/xlink	45,980 (0.67%)
http://www.w3.org/1999/xhtml	44,228 (0.64%)
http://fb.me/use-check-prop-types	30,298 (0.44%)
http://www.w3.org/1998/Math/MathML	24,544 (0.36%)
http://www.w3.org/XML/1998/namespace	23,964 (0.35%)
http://localhost	19,447 (0.28%)
http://a	17,616 (0.25%)
http://a/c%20d	14,364 (0.21%)
https://fb.me/react-warning-dont-call-proptypes	12,516 (0.18%)
Total	326,234 **4.7%**

Table 2: Top-10 most common URL in dataset

4.2 Dataset

Table 1 summarizes our package/URL dataset. Overall, 22.6% of packages contain complete or incomplete URLs within their code, resulting in $6,898,948$ URLs overall, and $1,834,110$ distinct URLs. Note, we include incomplete URLs in our analysis, as many such fragments contain sufficient information (e.g., complete domain) to enable us to perform our analyses. These fragments constitute a small fraction of our dataset.

In terms of URI schemes, the majority of identified URLs (72.1%) used HTTPS. However, a significant portion of identified URLs (27.2%) used unencrypted HTTP. A small potion of identified URLs (0.6%) used the FILE scheme, most of which were either incomplete fragments or placeholder examples, and a tiny fraction of which pointed to valid JavaScript or JSON files within the package. The remainder of our dataset (0.1%) used other schemes including FTP, GOPHER, TELNET, and WAIS. Due to the negligible fraction of these outlier schemes, we do not analyze them further.

For reference, we also display the top-10 encountered URL expressions in Table 2. Overall, they account for 4.7% of our dataset. Five of them represent XML namespaces. Their use is necessary to ensure XML-based formats such as SVG and MathML are standard-compliant. While these namespaces are valid URLs, they are not intended to represent actual web content. Others may provide context to specific error messages, and some appear to be used for host-local communication.

5 Analysis

Our measurements aim at answering these questions:

- What are the characteristics of URLs within packages?
- What are the characteristics of the domains within URLs?
- Which type of content is accessed via URLs? And how much of this content is still accessible?

5.1 URL Demographics

5.1.1 HTTP URLs. We discuss first the characteristics of URLs using plain HTTP, which make up 27.2% of our total dataset. Note that in general, HTTP is a deprecated protocol due to lack of secrecy and authentication; thus, the fact that more than 1/5 of our

[1]Note that we ignore package deletions, as those are oftentimes related to takedowns of malicious packages, which we specifically wish to archive for research.

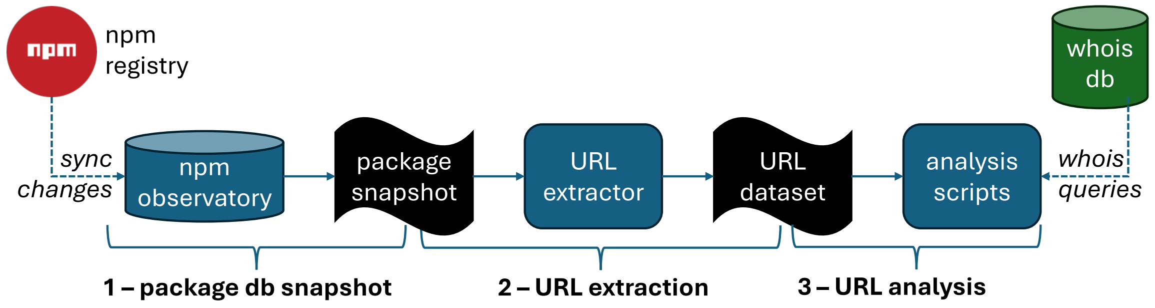

Figure 1: Overview of URL analysis pipeline

URL dataset consists of HTTP URLs is somewhat surprising. As discussed above, a fraction of these HTTP URLs consist of XML namespaces, which are valid URLs but not expected to represent actual online locations; however these only constitute a minority of the overall number of HTTP URLs. Of the identified HTTP URLs, 55.7% were URL string constants pointing to static web pages[2], and an additional 33.5% were URL string constants pointing to statically hosted files. The rest consisted of various types of dynamically constructed URLs and partial URL strings, the latter intended to be dynamically completed at runtime. Of those, incomplete URL fragments which lacked a complete domain name or IP (e.g. `http://` or `http://abc`) accounted for 6.6% of the sample, and incomplete URLs which declare but do not supply a value for a query variable made up 1.5% of the sample[3]. Across the entire set, 7.6% of identified HTTP URLs accessed localhost, and 3.9% were static IPs (suggesting their possible use for testing or examples).

Concerningly, a total of 1, 301, or 0.3% of examined HTTP URLs supplied username and password credentials directly in the plaintext of the URL. However, many of these credential-carrying URLs appeared to supply only default credentials or were intended to serve as examples for the purpose of URL parsing. Furthermore, in modern server setups, requests for HTTP URLs are frequently redirected to equivalent HTTPS endpoints, which somewhat mitigates their negative security impact. However, we posit that even the implicit endorsement of sending plaintext credentials over unencrypted HTTP communications is cause for reasonable security concerns.

5.1.2 HTTPS URLs. As for the identified HTTPS URLs, 64.8% were URL string constants pointing to static web pages, and an additional 33.8% were statically hosted files. Indications of dynamic URL construction were significantly less common. Incomplete URL fragments accounted for only 0.5% of the sample, and URLs which declare but do not supply a value for a query variable made up 1.0% of the sample. Additionally, just 0.2% of identified HTTP URLs accessed the localhost, and only 0.1% were static IPs.

With respect to passing user credentials, a total of 1, 095, or 0.1% of examined HTTPS URLs supplied username and password credentials directly in the plaintext of the URL. As this percentage is significantly smaller than in the HTTP case, it is possible that HTTPS usage correlates to greater security awareness as compared to HTTP usage within npm packages.

[2]We define here static web pages as web pages without indication of server-side dynamic construction, such as URL parameters. We acknowledge that this definition is incomplete, as any web page may be dynamically constructed w/o external indicators.
[3]Note that the two detection rules for "incomplete" and "missing parameters" are not mutually exclusive; the former detects URL which does not includes all sections required by specification; while the latter detects URLs ending with "=".

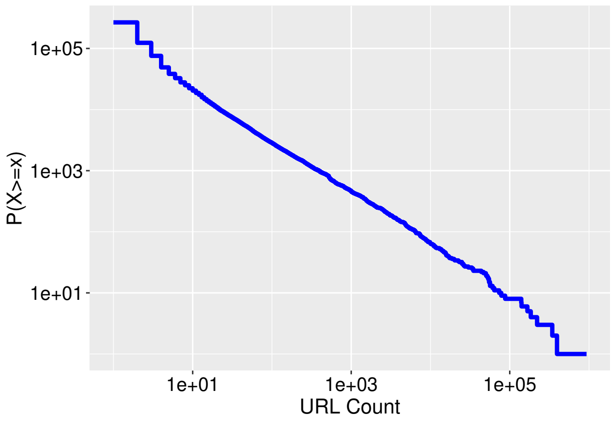

Figure 2: Empirical CDF for URL count per domain

5.1.3 Take-aways. A significant percentage of URLs in our dataset use the plaintext HTTP protocol rather than HTTPs. Across our datasets, we also found over 2,000 URLs which incorporate user credentials, despite the fact that HTTP basic authentication is generally deprecated due to its security implications. While these findings do not directly imply the existence of vulnerabilities, they suggest gaps in the security awareness of npm developers and poor security posture amongst many packages.

5.2 Domain Demographics

A URL domain is critical from a security point of view, because it determines—directly or indirectly—who controls the content pointed by the URL. Out of 1, 834, 110 examined unique URLs, we identified a total of 267, 115 unique domains to which they belonged. A very small portion of these domains appeared frequently, such as `github.com`, which accounted for 13.5% of all unique URLs. Most domains appeared somewhat infrequently, and 10.9% of identified domains appeared only in a single URL. This suggests that the distribution of URLs across domains may follow a power law; this indeed confirmed by the ECDF of #URL per domain, displayed in Figure 2 (note the use of log scales for X and Y axis). Power-law

Domain	**Count (%)**
github.com	931,298 (13.5%)
www.w3.org	394,602 (5.7%)
ecma-international.org	344,045 (5.0%)
developer.mozilla.org	221,199 (3.2%)
262.ecma-international.org	184,130 (2.7%)
s3.amazonaws.com	166,912(2.4%)
fb.me	140,879 (2.0%)
lichess.org	139,787 (2.0%)
fonts.gstatic.com	87,492 (1.3%)
docs.aws.amazon.com	78,232 (1.1%)
Total	**268,8576 (39.0%)**

Table 3: Top-10 most common domains in URL dataset

distributions tend to arise in npm in relation to item popularity (e.g., package download counts [24]).

For details, Table 3 lists the top-10 most popular domains, which together account for nearly 40% of the URLs. Many of them are likely used to download additional features (e.g., `fonts.gstatic.com`) or interface to online storage resources (e.g., `s3.amazonaws.com`). Interestingly, a significant (2%) fraction of the URLs points to *lichess.org*, an online chess platform. We discovered this is due to the single `l1-chessboard` package (a web component implementing a chessboard), which includes 139, 668 URLs pointing to the site.

5.2.1 Domain registration status. For each unique domain, we performed a `whois` domain lookup query [11] to establish domain registration and expiration information. We note that although `whois` may provide incomplete information, it is the most general and readily available approach to gather domain data, and its use is common in network security research (e.g., [7, 21]). Thus, we believe `whois` data is still valuable for lower-bound estimates of *domain expiration*. This analysis is important, as expired domains are up for grab for anyone willing to pay the registration fee [19]. As such, if a package relies on a URL pointing to an expired domain, a malicious actor may be able to take over control of that URL and exploit it to influence package behavior.

We identified a total of 784 unique domains across 648 packages, whose registration was past the expiration date provided by `whois`. To verify this result, we manually analyzed a subsample of 86 of these domains (11% of all those flagged by `whois`) to determine whether they were truly expired. Of this subsample, 20 domains (23.3%) had been reclaimed and re-registered, while 66 (76.7%) remained expired. Assuming that the distribution from this subsample generalizes, we estimate that up to 601 identified domains may remain expired and may be potentially vulnerable to malicious takeover.

5.2.2 Missing data. Of the 267, 115 identified domains, 31.3% of their respective `whois` queries were unable to identify a domain registrar. However, this does not necessarily indicate that a domain is truly unregistered. It is likely that many of these instances are due to invalid or incomplete domain fragments, or they are the result of imperfect `whois` information (e.g. whois does not supply domain registration information for European domains due to the establishment of GDPR in 2018 [17], and as such, we were unable to automatically identify a domain registrar for those domains). We investigate the existence of truly valid but unregistered domains by manually vetting a random sample of 100 domains in which the `whois` query was unable to identify a domain registrar. Within this sample, 47 domains were invalid or incomplete, 43 were valid and registered, and 10 were valid, unregistered, and publicly available for purchase (verified using Google Domains [3]). Assuming the distribution from this subsample generalizes, we estimate that up to 8, 360 of identified domains may be unregistered and publicly available for purchase.

5.2.3 Take-aways. While `whois` data is noisy, our results - complemented by manual analysis - found a non-negligible number of packages incorporating URLs with expired domains. These findings indicate that domain takeover attacks constitute a potential threat to the health of the npm package ecosystem.

5.3 URL analysis

We additionally test whether each of the 1, 834, 110 unique identified URLs successfully resolve statically via a web request. Out of all tested URLs, 70.4% successfully resolve, and 29.6% do not. We note that failed resolutions are a significant over-approximation because not all URLs are intended to be accessed statically, without any dynamic construction, outside of a package's intended environment.

5.3.1 URL content by popularity. We analyze the content of URLs in our dataset, grouping the URLs in three broad categories based on package popularity. We use weekly download counts as a proxy for popularity. We define three categories, based on previous literature [23]: highly popular packages (more than 100, 000 downloads/week), somewhat popular packages (between 350 and 100, 000 downloads/week), and never-downloaded packages (less than 350 downloads/week). The choice of 350 as a threshold for never-downloaded packages is due to the fact that npm packages experience a routine baseline of 50 downloads/day due to mirrors and bots. Stratifying analysis by popularity is done due to the fact that packages in different popularity groups tend to exhibit distinct traits [27].

5.3.2 Manual case study. We perform manual analysis on the subset of packages that were at least somewhat popular and utilized domains which were confirmed to be expired. This amounted to 5 packages in total, none of which were highly popular; rather, each garnered weekly download counts in the thousands. Of these 5 packages, 2 have since released a newer package version which no longer contains the expired domain, while 2 still contained test cases which accessed expired domains, and one still exported a data object containing an expired domain. This manual analysis demonstrates that even npm packages which are somewhat popular can be potentially compromised by the malicious takeover of expired domains.

Results. Table 4 breaks down results for both URLs that resolved successfully, and URLs that failed to resolve. In the former case, we distinguished between content types: text, web applications, static files across image, audio, binary, font, and video formats, and replies without content type. In the latter case (URL did not resolve), we distinguished between error types: failing HTTP status codes, connection refused, timeout, invalid URL, and Unicode-related.

		Successfully resolves [content type]				Does not resolve [error type]				
Dl count	**#URLs**	**Text**	**Web app**	**File**	**No type**	**Err. code**	**Conn refused**	**T/O**	**Invalid URL**	**UC err**
$> 100,000$	18,288	59.4%	3.1%	1.5%	10.8%	16.4%	7.3%	1.5%	0.1%	0.03%
$(350 - 100.000]$	330,763	41.9%	2.9%	20.1%	8.2%	13.0%	12.6%	1.2%	0.05%	0.03%
$[0, 350]$	1,448,168	43.7%	4.1%	13.2%	8.7%	17.3%	10.0%	2.9%	0.1%	0%

Table 4: URL analysis results by download count

For the 18,288 unique URLs accessed by highly popular packages, we find that 74.8% successfully resolve statically, while 25.2% do not. Most URLs return some type of text reply, suggesting that the use of Web APIs (e.g., REST) may be the main reason for URL embedding.

For the 330,763 unique URLs accessed only by somewhat popular packages, we find that 73.1% resolve statically, and 26.9% do not. For the remaining 1,448,168 unique URLs accessed only by never-downloaded packages, we find that 69.7% successfully resolve statically, and 30.3% do not. In both of these latter groups, a non-negligible percentage of URLs return file-based content.

5.3.3 Manual URL testing. We further verify the status of URLs that fail to resolve via static access by performing manual analysis on random subsamples of such URLs. We manually test each URL in a given subsample within the package context in which it is intended to be used, and we additionally look for any more recent updates to package code.

Highly popular packages. In a random subsample of 100 URLs which are accessed by highly popular npm packages and fail to resolve via static, programmatic access, we find that 35 resolve successfully within the intended context (including 8 which require additional configuration performed by the package code) 15 that fail within the context of the most up-to-date version of the package, 33 that fail but were removed from a newly released version of the package, 14 that were not intended to be accessed via a web request (e.g. test cases for URL parsing), and 3 that were intended to fail (e.g. test cases for failed web requests).

Somewhat popular packages. We contrast this with a random subsample of 100 URLs which are accessed only by somewhat popular npm packages and fail to resolve via static, programmatic access, wherein we find that 35 resolve successfully within the intended context (including 23 which require additional configuration performed by the package code), 25 that fail within the context of the most up-to-date version of the package, 30 that fail but were removed from a newly released version of the package, and 10 that were not intended to be accessed via a web request.

Never-downloaded packages. Last, we examine a random subsample of 100 URLS which are accessed only by never-downloaded npm packages and fail to resolve via static, programmatic access, wherein we find that 23 resolve successfully within the intended context (including 10 which require additional configuration performed by the package code), 66 that fail within the context of the most up-to-date version of the package, 4 that fail but were removed from a newly released version of the package, and 7 that were not intended to be accessed via a web request.

5.3.4 Take-aways. Our findings highlight notable differences in URL usage across the popularity of packages which utilize them. First, the more popular a package is, the more likely its URLs are to resolve statically. Additionally, more popular packages are rather unlikely to access static files via web requests, while less popular packages do so much more frequently. Furthermore, we found that as the popularity of npm packages increase, they are significantly more likely to maintain up-to-date URLs and to utilize more extensive testing as compared to less popular packages.

6 Discussion

6.0.1 Presence of Problematic URL Usage. Our findings demonstrate that problematic URL usage is a present threat within the npm package registry, albeit one which is primarily localized within unpopular packages that are infrequently maintained or outright abandoned. Despite this, we have identified several instances of somewhat popular packages utilizing expired domains, as well as significant use of unencrypted HTTP communication, even within highly popular packages. Packages which access web content, and especially those which download files over HTTP, are prone to adversarial manipulation and potential compromise. We hope that our findings motivate package authors to improve the hygiene and maintenance of URL usage within their packages.

6.0.2 Other Potentially Undesired URL Usage. In addition to dead links and expired domains, we observe some additional potentially undesired URL usage across the npm registry. We identify some packages which appear to promote affiliate marketing links intended to generate profits for the package authors. One such package was `@bevry/links@2.9.0`, which appears to exist for the sole purpose of storing links with affiliate URL redirections. On a similar note, URLs which link to advertising servers are also somewhat common, in packages implementing ad-related functionality such as bidding (e.g. `mk9-prebid@5.12.0`). Moreover, we find numerous packages intended to generate, trade, and interact with cryptocurrency, including many dead or broken links to cryptocurrency exchanges, which may or may not be legitimate (e.g. `solana-token-list@0.0.24`, `moralis@2.14.2`). These packages clutter the ecosystem and may confuse newcomers looking for reliable cryptocurrency-related functionality. Finally, in some extreme cases we found packages downloading executable JavaScript code from URLs; this is a highly sensitive operation which should be avoided unless absolutely necessary. One such package is `goindexmod666647@1.0.1`, which downloads what appears to be a minified JS script to modify a web CSS configuration. We hope that these findings can raise awareness of some potentially undesired forms of URL usage we observed across npm packages.

6.0.3 General implications of our findings. Our goal is to perform a measurement study to understand developer practices concerning use of URLs in npm packages. As such, and also given the ethical implications of doing so, we did not attempt to exploit identified issues or to perform domain takeover. Rather, we reported our findings to the npm security team, including a list of the expired domains identified and the specific package versions utilizing them. As of the time of writing, we have yet to receive a response from the npm security team.

Overall, we believe that our findings point to both security and functional issues deriving from referring to network resources using URL string constants. While there are legitimate reasons to do so (e.g., pointing to API endpoints for database backends), our investigation also reveals a large number of insecure URLs, expired domains, and broken URLs, with negative implications for security and functionality. In general, when hardcoding references to online resources within a package, care should be taken to assess whether (i) this is absolutely necessary; and (ii) which measures will be taken to ensure that the URL—and thus the package using it—continues to function. We put forward that developers, especially inexperienced ones, may benefit from guidelines concerning the use of hardcoded URLs in their code.

7 Conclusion

In this work, we presented a brief review of the use of hardcoded URLs within packages in the npm ecosystem. At a high-level, our measurement study shows prevalence of risky practices such as use of unencrypted HTTP, use of basic HTTP authentication, broken/nonfunctioning URLs, and presence of expired domains. We hope that these findings can foster better awareness and more robust development practices.

Data access

We publicly release our dataset at https://osf.io/5bqnp/?view_only=da6dde3e665c41a6b1cb5943a702a363 to promote broader access and replicability.

References

[1] 2023. SigStore - Open Source Security Foundation. https://openssf.org/community/sigstore/
[2] 2024. Apache CouchDB. https://couchdb.apache.org/
[3] 2024. Google Domains - My domains. https://domains.google.com/registrar/?pli=1
[4] 2024. Npm | Home. https://www.npmjs.com/
[5] 2024. Public Registry API. https://github.com/npm/registry/blob/main/docs/REGISTRY-API.md
[6] Ax Sharma. 2022. 241 Npm and PyPI Packages Caught Dropping Linux Cryptominers. https://www.bleepingcomputer.com/news/security/241-npm-and-pypi-packages-caught-dropping-linux-cryptominers/
[7] Leyla Bilge, Engin Kirda, Christopher Kruegel, and Marco Balduzzi. 2011. EXPOSURE: Finding Malicious Domains Using Passive DNS Analysis. In *NDSS*.
[8] George Christou, Grigoris Ntousakis, Eric Lahtinen, Sotiris Ioannidis, Vasileios P Kemerlis, and Nikos Vasilakis. 2023. BinWrap: Hybrid Protection against Native Node.Js Add-ons. In *ACM AsiaCCS*.
[9] Thomas Claburn. 2022. Expert Grabs Expired Domain for NPM Package to Make a Point. https://www.theregister.com/2022/05/10/security_npm_email/
[10] CloudFlare. 2024. URL Scanner. https://radar.cloudflare.com/scan
[11] Leslie Daigle. 2004. *WHOIS Protocol Specification.* Request for Comments RFC 3912. Internet Engineering Task Force. https://datatracker.ietf.org/doc/rfc3912
[12] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories.*
[13] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2021. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. In *Proceedings 2021 Network and Distributed System Security Symposium.* Internet Society.
[14] Duc Ly Vu, Zachary Newman, and John Speed Meyers. 2023. Bad Snakes: Understanding and Improving Python Package Index Malware Scanning. In *ICSE*.
[15] Fabian Niklas Froh, Matías Federico Gobbi, and Johannes Kinder. 2023. Differential Static Analysis for Detecting Malicious Updates to Open Source Packages. In *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED '23).*
[16] Behnaz Hassanshahi, Trong Nhan Mai, Alistair Michael, Benjamin Selwyn-Smith, Sophie Bates, and Padmanabhan Krishnan. 2023. Macaron: A Logic-based Framework for Software Supply Chain Security Assurance. In *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED '23).*
[17] INTA News. 2022. The European Union Continues to Tackle the WHOIS Issue. https://www.inta.org/news-and-press/inta-news/the-european-union-tackles-the-whois-issue/
[18] Keith Collins. 2016. How One Programmer Broke the Internet by Deleting a Tiny Piece of Code. https://qz.com/646467/how-one-programmer-broke-the-internet-by-deleting-a-tiny-piece-of-code
[19] Tobias Lauinger, Abdelberi Chaabane, Ahmet Salih Buyukkayhan, Kaan Onarlioglu, and William Robertson. 2017. Game of Registrars: An Empirical Analysis of Post-Expiration Domain Name Takeovers. In *USENIX Security Symposium.*
[20] Nikunj Patel. 2023. Malicious Npm Packages Strike Again: Exfiltrating Kubernetes Configurations and SSH Keys. https://www.cyber-oracle.com/p/malicious-npm-packages-strike-again
[21] Roberto Perdisci, Igino Corona, David Dagon, and Wenke Lee. 2009. Detecting Malicious Flux Service Networks through Passive Analysis of Recursive DNS Traces. In *ACSAC*.
[22] Adriana Sejfia and Max Schäfer. 2022. Practical automated detection of malicious npm packages. In *ICSE*.
[23] Matthew Taylor, Ruturaj Vaidya, Drew Davidson, Lorenzo De Carli, and Vaibhav Rastogi. 2020. Defending Against Package Typosquatting. In *NSS*. Springer-Verlag, Berlin, Heidelberg.
[24] Ruturaj K. Vaidya, Lorenzo De Carli, Drew Davidson, and Vaibhav Rastogi. 2019. Security Issues in Language-based Sofware Ecosystems. *CoRR* abs/1903.02613 (2019). arXiv:1903.02613 http://arxiv.org/abs/1903.02613
[25] VirusTotal. 2024. VirusTotal - Home. https://www.virustotal.com/gui/home/upload
[26] Elizabeth Wyss, Lorenzo De Carli, and Drew Davidson. 2022. What the Fork?: Finding Hidden Code Clones in Npm. In *ICSE*. https://dl.acm.org/doi/10.1145/3510003.3510168
[27] Elizabeth Wyss, Lorenzo De Carli, and Drew Davidson. 2023. (Nothing But) Many Eyes Make All Bugs Shallow. In *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED '23).*
[28] Elizabeth Wyss, Alexander Wittman, Drew Davidson, and Lorenzo De Carli. 2022. Wolf at the Door: Preventing Install-Time Attacks in Npm with Latch. In *ACM AsiaCCS.*
[29] Markus Zimmermann, Cristian-Alexandru Staicu, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the Npm Ecosystem. In *USENIX*. 17.
[30] Jonathan Zittrain, John Bowers, and Clare Stanton. 2021. The Paper of Record Meets an Ephemeral Web: An Examination of Linkrot and Content Drift within The New York Times. *SSRN Electronic Journal* (2021). https://www.ssrn.com/abstract=3833133