Welcome to Jurassic Park: A Comprehensive Study of Security Risks in Deno and its Ecosystem

Abdullah AlHamdan
CISPA Helmholtz Center for Information Security
abdullah.alhamdan@cispa.de

Cristian-Alexandru Staicu
CISPA Helmholtz Center for Information Security
staicu@cispa.de

***Abstract*—Node.js and its ecosystem npm are notoriously insecure, enabling the proliferation of supply chain attacks. Deno is an emerging runtime that promises to offer a safer alternative for running untrusted JavaScript code outside of the browser. Learning from Node.js's mistakes, Deno is written in Rust, a memory-safe programming language, and it includes a strict permission system that checks all accesses to sensitive APIs via static or runtime permissions. Deno also allows the inclusion of third-party code via URLs, which promises a more transparent way of handling dependencies, advocating for a fully *decentralized software supply chain*. In this paper, we study if Deno delivers on its promise of increased security. We find that indeed Deno has a smaller attack surface than Node.js, but there still are known attacks that are not addressed (ReDoS) or only partially mitigated (prototype pollution). Moreover, we find several weaknesses in Deno's permission system, which allow sophisticated supply chain attacks. First, coarse-grained permissions allow attackers to abuse the ambient authority of the operating system to sidestep the permission system. Second, we find that URL imports are exempted from the permission checks, allowing attackers to perform unlawful network requests. We also identify time-of-check to time-of-use issues when handling symbolic links, making fine-grained file system access control ineffective. We then perform an empirical study of Deno's main ecosystem deno.land to understand how developers consume third-party code and how permissions are used and communicated. We identify classical URL-related issues such as expired domains and reliance on insecure transport protocols, but we also find that it is challenging to guarantee uniform immutability and version control when multiple domains are involved in code distribution. We also provide initial evidence that developers poorly document required permissions on deno.land and that they tend to abuse coarse-grained permissions, reducing the benefits of the permission system. Our findings resulted in two security advisories for Deno and a redesign of its import mechanism. We also make concrete recommendations for improving Deno's security model to further prevent supply chain attacks: add import permissions, additional access control at file system level, support for compartmentalization, and a manifest file that persists fine-grained permissions.**

I. Introduction

JavaScript is arguably the most popular programming language in the world, supporting a wide palette of use cases, starting from simple client-side animations in the browser to full-fledged, portable desktop applications. Node.js is a popular, open-source, cross-platform runtime that enables developers to execute JavaScript code outside of a web browser. Its package manager, npm, is the largest package collection in the world, hosting more than two million packages. However, this success also led to significant security issues, including the high prevalence of vulnerabilities [1], [2], [3], [4], [5] and the widespread distribution of malicious packages [6], [7]. As a result, both academic and developer communities have been working to improve security by implementing stricter package distribution guidelines [8], [9] and by restricting the capabilities of consumed packages [10], [11].

Deno is a radically new JavaScript runtime for server-side and standalone applications, designed with security in mind. According to State of JS[1], Deno is the second most used server-side JavaScript runtime after Node.js, with 11.2% of developers using it in 2022, a raise from 5.6% in 2021. Additionally, it is one of the most starred open-source projects (93k GitHub stars) along Node.js (105k stars). Deno provides strong isolation and a restrictive permission system. The authors of Deno argue that it is secure by default, *allowing developers to run untrusted code with confidence*[2], similar to a web browser. The permission system empowers developers to selectively grant either static or runtime permissions to grant access to sensitive operations. One can grant either coarse-grained permissions, e.g., allow all network operations, or fine-grained ones, e.g., only allow reading a specific file from the disk. Nonetheless, all permissions are granted at the application level, so there is currently no built-in support for compartmentalization [10], [11], i.e., reducing the privileges of some parts of the code only. Moreover, certain permissions like `--allow-sys` allow running binary code that is not under the control of the permission system [12], rendering the security mechanism ineffective in such cases.

Let us consider the example in Figure 1 to illustrate how Deno's permission system works. In this example, we use `earthstar`, a package that implements a third-party distributed storage protocol. We initialize a local database (lines 3-7), add a local file into the database (lines 7-12), and synchronize with a remote instance (lines 13-15). To successfully execute this piece of code, we need to grant

[1]https://2022.stateofjs.com/en-US/other-tools/#runtimes
[2]https://docs.deno.com/runtime/manual/basics/permissions

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.230284
www.ndss-symposium.org

```
1 import {Crypto, Replica, ReplicaDriverMemory, Peer
      } from "https://deno.land/x/earthstar/mod.ts";
2
3 const p = await Crypto.generateShareKeypair("ab");
4 const replica = new Replica({
5   driver: new ReplicaDriverMemory(p.shareAddress),
6   shareSecret: p.secret,
7 });
8 await replica.set(p, {
9   path: "/img/leaf.jpg",
10  text: "A sample image",
11  attachment: await Deno.readFile("./leaf.jpeg"),
12 });
13 const peer = new Peer();
14 peer.addReplica(replica);
15 peer.sync("https://my.server");
```

(a) Example usage of the earthstar package to initialize a database with a local image and synchronize it with a remote instance.

```
> deno run --allow-read=./leaf.jpeg --allow-net
    sample.ts
```

(b) Permissions for the code above, passed as arguments to Deno.

Fig. 1: Sample TypeScript code and its Deno permissions.

Deno permissions to read the image file in line 9 and to make a network request in line 15. We note that while the file read operation is explicitly visible in line 11, the actual network request is hidden in the implementation of the `sync()` method invoked in line 15. If one attempts to run the code in Figure 1a in Deno's default configuration, the execution will pause both at line 11 and line 15 to prompt the user for consent to proceed with the sensitive operation. If the user does not explicitly grant this runtime permission, Deno will throw an exception instead of proceeding with the operation. For convenience, users can also grant permissions statically, as command line arguments to Deno. In Figure 1b we show how one can grant coarse-grained permissions to the network (`--allow-net`) or fine-grained permissions to read a specific file (`--allow-read=./leaf.jpeg`).

Another peculiar feature of Deno is its mechanism for consuming third-party code. While Node.js advocates for a centralized solution, in which all packages are published on npm or on private package repositories, Deno proposes a fully decentralized approach. As seen in line 1 of Figure 1a, Deno allows importing third-party code via unrestricted URLs, akin to browsers. We argue that this results in a *decentralized software supply chain* with multiple benefits, e.g., increased resilience, but also with multiple challenges, e.g., non-uniform security policies for the involved domains. Let us take a closer look at the software supply chain of the `earthstar` package, depicted in Figure 2. When importing this package from deno.land, Deno makes 114 network requests to retrieve all the transitive dependencies, from four different domains. We note an inconsistent publishing policy among these domains: while deno.land prevents published code from being altered after release, the code included directly from GitHub has no such restrictions. We believe that this can lead to serious availability issues in the future, akin to the infamous `left-pad` incident[3], when the deletion of an npm package lead to multiple build failures in dependant projects. This is because the resilience of the entire supply chain depends on its weakest node.

In this paper, we perform a comprehensive analysis of Deno's security risks, insisting on how the newly adopted features like the permission system or the URL-based import can influence the runtime's security. We first perform (i) a comparative study of Deno's and Node.js's attack surface. We then (ii) explore ways to circumvent Deno's permission system, e.g., by exploiting the power of coarse-grained permissions. Finally, (iii) we analyze Deno's decentralised software supply chain by performing an in-depth study of deno.land.

By analyzing the security model of Deno and known vulnerabilities of Node.js, we find that indeed Deno's attack surface is reduced by the deployed mitigations. For example, by using a memory-safe programming language for implementing Deno, entire classes of low-level vulnerabilities are eliminated, e.g., buffer overflows in binding code [13]. Nonetheless, we find that Deno users can only benefit from the reduction in the attack surface if they correctly configure the security mechanisms, i.e., avoid giving too many (coarse-grained) permissions to their applications and minimize careless code reuse. We also note that there are important differences between Deno's and other widely-studied permission systems: (i) as opposed to Android, Deno does not deploy additional file system access control [14] to prevent access to sensitive OS resources, (ii) it does not offer a way to persist the permissions in a manifest file, and (iii) it allows context-based calls to sensitive APIs, unchecked by the permission system, e.g., perform network requests to import third-party code.

Next, inspired by prior work on Android [14], [15], [16], [17], [18], we perform an in-depth security analysis of Deno and **discover unexpected ways to sidestep the permission system**. Concretely, we find that coarse-grained read permissions allow unprivileged access to environmental variables via internal operating system files, and write permissions allow arbitrary code execution by modifying OS start-up scripts. These attacks would not be possible if Deno would supplement the permission checks with additional access control [14]. We also find that Deno's fine-grained file system permissions do not work correctly in the presence of symbolic links, allowing attackers to escape directories for which permissions were granted. Finally, leveraging the exemption of static code `imports` from network permission checks, we show how attackers can exfiltrate arbitrary data from the victim machine, when only file system permissions are given. These attacks **show for the first time the feasibility of supply chain attacks in Deno**, in particular, when users carelessly grant coarse-grained permissions. We disclosed all our findings to Deno's security team and to a third-party specialized in vulnerability disclosure. At the time of writing, two CVEs

[3]https://arstechnica.com/information-technology/2016/03/rage-quit-coder-unpublished-17-lines-of-javascript-and-broke-the-internet/

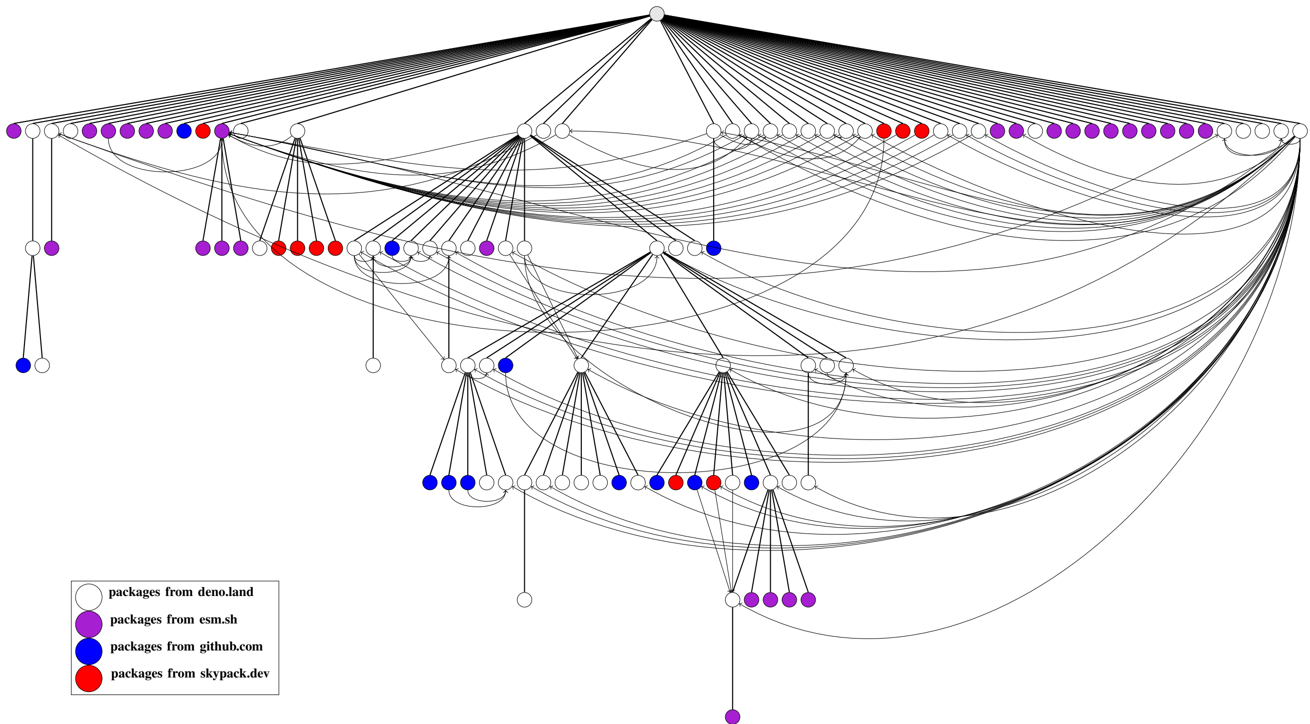

Fig. 2: Dependency graph for `earthstar` package on deno.land with its transitive dependencies, illustrating Deno's decentralized software supply chain. Each node is a different package and colored nodes are packages hosted outside deno.land.

were issued for our findings (CVE-2024-21486, CVE-2024-21487) and the Deno team promised to deploy a permission check for importing code from untrusted domains in Deno 2.0.

We also perform a measurement study of deno.land, the main software registry of Deno to study code reuse practices and to understand how developers configure the permission system. We analyze all the 5,400 packages hosted on this registry, together with all their transitive dependencies, including the ones stored outside deno.land. In total, during dependency resolution we visited 10,544 URLs from 21 different domains. We show that **decentralized software supply chains introduce unique challenges**: 39 packages include code over insecure protocols (HTTP), more than 400 packages transitively include unavailable URLs, and one of the 21 domains involved in distributing the code was available for sale, allowing attackers to take control of legitimate packages. We also find that it is difficult to guarantee transitive immutability and to perform version control of dependencies, in a decentralized setup. Our measurements also **highlight the limited usability of Deno's permission system**. First, developers tend to recommend granting coarse-grained permissions over fine-grained ones, potentially amplifying the weaknesses of the permission system. We believe that this is due to the cumbersome way of granting permissions as command line arguments. Similar to Android, we also note that Deno packages are overprivileged [15] and that unexpected permission prompts may surface at runtime due to third-party code [17].

Based on our findings, we propose concrete improvements to Deno's ecosystem. First, with respect to the permission system, we recommend to (i) add permission checks for static imports, (ii) fix the faulty support of symbolic links, (iii) block read/write access to sensitive operating system paths, (iv) use compartmentalization to further reduce the privileges of untrusted code, and (v) add support for permission policy files. Regarding code reuse practices on deno.land, we recommend more redundancy in the distributed software supply chain via immutable mirrors of packages, the discouraging of coarse-grained permissions, and vetting and uniformizing security practices of widely-used content delivery networks.

In summary, our contributions are the following:

- We are the first to perform a thorough analysis of Deno's security features, in particular of its permission system. We find that coarse-grained permissions, unchecked static imports, and faulty support for symbolic links enable powerful supply chain attacks in Deno. Our findings lead to a redesign of the import mechanism and to the publication of two security advisories.
- We present the first measurement study of code reuse practices in deno.land. We analyze 5,400 packages and their dependencies, delivered by 21 distinct domains. We note the fragility of this decentralized software supply chain by uncovering packages that import code from insecure domains, broken import URLs, or expired domains. We also warn about the over-reliance on coarse-grained permissions on deno.land, which might amplify the weaknesses of Deno's permission system.
- We discuss concrete improvements to Deno's security model that can mitigate most of the discovered issues.

II. Deno's features and attack surface

In this section, we outline Deno's threat model and other security-relevant features. We then study whether known security concerns of Node.js also apply to Deno, or how the deployed security controls reduce the attack surface.

A. Deno's Security Model and Features

One of the main promises of Deno is to learn from Node.js's mistakes and significantly reduce the privileges of the executed code[4]. Deno makes grandiose security promises like:

- *Deno's secure sandboxed environment means that you can run untrusted code and party dependencies with confidence.*
- *You no longer need to conduct audits on your dependencies — simply run your program and see what access your app and its dependencies require.*

To do so, Deno implements a Rust-based wrapper around the JavaScript engine, which promises full interposition for all runtime I/O operations, e.g., file system accesses. That is, whenever the code requires access to a privileged API, it only proceeds after obtaining explicit consent from the user. However, the permissions are granted for the entire application, thus, there is no built-in support to compartmentalization, i.e., only granting certain permissions for a subpart of the application.

Deno supports eight types of permissions[5]: file system (i) reads or (ii) writes, (iii) running subprocesses, (iv) network requests, access to (v) environmental variables or (vi) system information, (vii) perform high resolution timing measurements, and (viii) loading dynamic libraries. Each permission can be specified in two ways: as static or runtime permission. *Static permissions* are predefined before the program's execution and are provided as command line arguments to Deno. When no static permissions are given or they are insufficient, the runtime pauses the execution and asks for explicit consent to proceed with each offending API calls. Once the consent is given for a particular API, its associated *runtime permission* becomes part of the policy and subsequent calls to the same API will be granted access automatically. By default, all the code executed on Deno is assumed to be untrusted and thus, no permissions are automatically granted.

Both static and dynamic permissions can take two forms: coarse-grained or fine-grained. Coarse-grained permissions grant access to multiple resources at once, e.g., allow all network requests, while fine-grained ones only grant access for a specific resource, e.g., allow reading a single file from the disk. Users can also grant all permissions at once, but the documentation advises against this practice, since it drastically reduces the benefits of using Deno in the first place.

While useful, Deno's permission system is not very user-friendly, in its current form. First, we are not aware of any widely-used specification mechanism, similar to manifests in Android, where static permissions can be declared in a user-friendly way, thus, users need to declare them as command-line arguments, for each execution. Second, there is no mechanism to persist the runtime permissions and use them as static permissions in subsequent runs. Thus, when running the same program multiple times, the user has to choose between the following subpar options: (1) redundantly give consent via runtime permissions, for each run, (2) provide intricate fine-grained policies that can result in tens or even hundreds of command line arguments, (3) giving excessive privileges to their applications via coarse-grained permissions.

We note that traditional server-side runtimes, i.e., Node.js or Python, consume packages from a single domain, i.e., npm, through a package manager. This mechanism downloads the entire source code of each package declared as a dependency in a manifest file. Deno uses a radically-different approach to manage third-party dependencies. Similar to client-side code, it allows importing code from any domain via URLs, as seen in Figure 1a. Any valid URL is considered a legitimate source for fetching third-party code, e.g., CDNs, personal websites, and local IP addresses. The only restriction is that the request should either return a JavaScript, TypeScript, or a JSON file, which will be directly loaded into the runtime, i.e., Deno supports TypeScript natively, using on-the-fly compilation. Moreover, Deno does not require the upfront declaration of dependencies in a manifest file, and it does not force the user to download and use entire packages. Instead, developers can import only the functionality they need via URLs pointing to the desired files. A peculiar aspect of Deno's threat model is that the fetching and parsing of third-party code is not mediated by the permission system, only its execution is.

All in all, we argue that the unusual import mechanism proposed by Deno enables one of the first decentralized software supply chain for server-side code and the only case in which this paradigm is deployed in conjunction with a permission system. The smart contracts language Solidity and Golang possess a similar URL-based import mechanisms, where code can be imported from arbitrary third-party domains. However, we argue that in the context of JavaScript, where software supply chain problems are prevalent [19], [7], this is a radical departure from the Node.js model. Moreover, we are not aware of any other programming language or runtime where URL-based imports are deployed together with a permission system, as is the case in Deno. This combination can lead to unexpected interferences, as discussed later in this work.

Decentralized software supply chains promise increased flexibility and greater resilience, i.e., when central repositories like npm are down, the entire ecosystem is down, while that is not the case in Deno. While packages can be stored on any domain, Deno also provides a centralized repository where most of its packages are stored, i.e., deno.land. Similar to npm, deno.land adopts a version immutability policy in the sense that specific versions of packages cannot be altered, once published in the repository. Additionally, Deno also

[4]https://deno.com/learn/nodes-security-problem

[5]https://docs.deno.com/runtime/manual/basics/permissions

Security Concern	Affected	Mitigated
Code injection [2]	Yes	No
Command injection [2]	Yes	Yes
ReDoS [5], [3], [4]	Yes	No
Prototype pollution [20], [21], [22]	Partial	No
Install-time hooks [23]	No	n/a
Native extensions [24]	Yes	Yes
Path traversal [22]	Yes	Yes
Malicious dependencies [1]	Yes	No
Low-level/binding bugs in the runtime [13]	Partial	No
WebAssembly bugs [25]	Yes	No

TABLE I: A detailed overview of Deno's attack surface, considering known security concerns of Node.js.

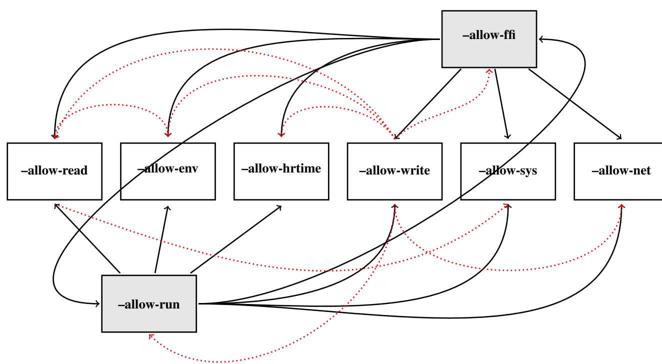

Fig. 3: Relations between coarse-grained permissions in Deno. An arrow between two permissions means that granting one implicitly grants the other as well. Solid arrows show previously-documented relations, while dashed ones show implications discovered by us.

provides an integrity checking mechanism[6], which guarantees predictable application composition by ensuring that each loaded dependency is locked at a specific version.

While creating a lock file, Deno collects all the imported files and calculates a hash value for each one. On further attempts to run the same application, for each imported file, Deno will compute a hash and verify that it matches with the one in the lock file. In this way, it ensures that no inadvertent modifications occur in the depended-upon third-party code. This feature exposes users to a well-known tension between version locking and continuous security updates, i.e., users need to explicitly update their dependency and recompute their hashes, upon each release/commit. As we discuss in Section IV-C, version management and the update of dependencies is a very challenging task in Deno applications, due to the decentralized software supply chain.

B. Deno's Attack Surface

To study the attack surface of Deno, we surveyed related work on Node.js security and attempted to implement proof-of-concept payloads, for each known issue, to verify if Deno is also affected by it. Concretely, we searched the proceedings of the last 10 years (2014-2024) of top-tier conferences in security and software engineering to identify papers that study Node.js-specific security problems. In Table I, we show the results of our investigation, highlighting for each problem whether it affects Deno, and if so, to what degree it is mitigated by the proposed permission system.

As seen in the table, most security issues are still affecting Deno, with few notable exceptions. Due to the proposed import mechanism, there is no installation phase for packages, and hence, no install-time hooks that can run during this phase. This is a worth-celebrating achievement, considering that such hooks are infamous vehicles for supply chain attacks [6] in Node.js. Also, due to the fact that Deno is written in Rust, many memory-related issues like the ones reported in Node.js's binding code are rather improbable in Deno. Nonetheless, the underlying JavaScript engine might still contain such security problems, but this is a rather small risk considering that the entire web threat model relies on this assumption as well. Finally, while there are efforts to make prototype pollution vulnerabilities impossible in Deno, we believe that these efforts are not complete yet. While the runtime disallows certain pollution paths (`Object.prototype.__proto__`), it still allows others (`Object.prototype.constructor.prototype`). Nonetheless, exploiting the latter is significantly harder, since it requires attackers to pollute three nested property accesses simultaneously.

For the remaining security problems affecting Deno, we observe that the permission system directly targets many of them. For example, to exploit a path traversal vulnerability, attackers need to trick the users into granting too broad file system permissions. The same applies to running native extensions/libraries or executing operating system commands. While the permission system reduces the attacker's space of manoeuvre for the other security issues, it does not aim to prevent them directly. For example, there is no security control available to restrict the provenance of the code executed via `eval`, e.g., similar to `script-src` in CSP. Similarly, resource exhaustion attacks like ReDoS are still possible on Deno and there is no built-in mechanism to restrict the effect of untrusted dependencies on the trusted code. Similarly, bugs in WebAssembly modules or the low-level code could spill into the Deno runtime unrestricted by the existing controls.

All in all, we believe that Deno is a step in the right direction, exposing a smaller attack surface to its applications. Nonetheless, the security of the runtime is highly dependent on the specified permissions and their runtime enforcement, which we study in detail in the next sections.

III. Security of Deno's permission system

As mentioned in Section II, out of the box, Deno is supposedly as secure as a browser, since no permission is granted to use the powerful Deno-specific APIs, e.g., file system access or spawning of new processes. However, the

[6]https://docs.deno.com/runtime/manual/basics/modules/integrity_checking

```
1 let _fname = new URL('', import.meta.url).pathname;
2 let oldContent = await Deno.readTextFile(_fname);
3 let passFl = await Deno.readTextFile('/etc/passwd');
4 let pre = 'import {foo} from "https://attacker.com?
     val=' + encodeURIComponent(passFl) + '";\n'
5 await Deno.writeTextFile(_fname, pre + oldContent);
```

Fig. 4: A two-steps attack that overwrites the current file to exfiltrate the content of /etc/passwd, without being granted the network permission.

```
1 // equivalent with Deno.env.toObject();
2 ev = await Deno.readTextFile("/proc/self/environ");
3
4 // equivalent with Deno.systemMemoryInfo();
5 mi = await Deno.readTextFile("/proc/meminfo");
```

Fig. 5: An example showing how coarse-grained file system read (--allow-read) permission allows access to sensitive information, which is otherwise guarded by other permissions (--allow-env,--allow-sys).

security of a Deno application is still highly dependent on the correctness of the underlying permission system. A sound permission system must ensure that (i) all critical functionality is guarded by a permission check and that (ii) the functionality guarded by a given permission should only be accessible when users explicitly grant that permission. We found evidence that Deno fails to enforce both these properties, enabling powerful supply chain attacks in this ecosystem. Below, we discuss our findings, grouped into three categories: unchecked functionality that we believe should trigger a permission check, capability leaks due to coarse-grained permissions, and problems with enforcing fine-grained permissions.

A. *Missing permission checks*

Import statements allow Deno developers to use third-party packages after loading them directly into the application space, either via a given package URL or a path to the package directory. Unfortunately, static imports, with all their ability to make a network request and access local directories, are exempted from the permission checks. This might open the door to performing sophisticated attacks via abusing the import mechanism. Deno's security model[7] states that this is intended part of the design: *the initial static module graph that is constructed when doing* deno run, *does not have any permission restrictions*. We argue that this can be abused in multiple ways by adversaries, as part of a supply chain attack. The simplest payload is a tracking pixel-like import that attackers place in their code to find out when developers use the attacker-controlled code. Since URLs in imports can contain GET parameters, attackers can generate different instances of their code for the target users, each instance containing a unique ID embedded in the import statement. Whenever developers run deno run on an application containing the malicious code, it will inform the attacker about this action, enabling precise tracking. This tracking payload reveals the time of the action and the unique application ID. It can be used to infer important information about the user's behavior, e.g., when do they open their IDE or their crypto wallet.

When write permissions (--allow-write) are also granted for the current folder, attackers can perform a sophisticated metamorphic attack in which they overwrite files that are typically loaded by the application with a static import that includes a GET parameter with privacy-sensitive information available in the runtime. When the application is run a second time, the static import will send the data to the attacker's website. We show such an attack in Figure 4, where we also assume a coarse-grained read permission (--allow-read). The attacker first retrieves the path of the current file (line 1). They then read the old content of the file (line 2) and of the password file (line 3). Subsequently, the password file is encoded as an HTTP parameter and appended to the URL pointing to the attacker's website (line 4), which is then written on the disk together with the old content of the file (line 5). If this payload is executed twice in Deno with read and write permissions, it exfiltrates the password file to the network, even though the user never granted the network permission.

B. *Risks of coarse-grained permissions*

Deno's documentation[8] warns that certain permissions allow running code outside the sandbox and hence, give attackers the ability to directly access functionality that would otherwise be controlled by a permission check. The documentation warns about two such permissions (--allow-run and --allow-ffi), which allow escaping the otherwise constrained execution environment of Deno. Abbadini et al. [12] study in detail this shortcoming of the security model and propose a way to harden it.

While mindful users might be aware of such weak parts of the permission system, they do not suspect that granting other coarse-grained permissions like read or write access to the file system, implies granting additional permissions, which we term *shadow permissions*. In Figure 3, we show all the shadow permissions we uncovered in our work. Most of them are caused by over-permissioned code in conjunction with quirks in the runtime or features of the underlying operating system. We argue that shadow permissions might both surprise users and enable future supply chain attacks in Deno. Below, we discuss concrete cases of shadow permissions and their potential impact.

First, we argue that in most cases, granting a coarse-grained write permission (--allow-write) is equivalent with a full bypass of the permission system. That is because when this permission is granted, Deno allows unrestricted write access to all files for which the current user has operating system privileges. On UNIX systems, for example, attackers can

[7] https://deno.land/x/deno@v1.18.2/SECURITY.md

[8] https://docs.deno.com/runtime/manual/basics/permissions#permissions-list

```
1 // The symlink allows escaping the cache folder
2 let pwd = await Deno.readTextFile("./cache/root-
      folder/etc/passwd");
3 console.log(pwd);
```

(a) Example code leveraging the symbolic link.

```
> deno run --allow-read=./cache --allow-write=./cache
    poc.js
```

(b) Permissions granted to the malicious application. The cache folder contains a symbolic link `root-folder` to the root of the operating system.

Fig. 6: A proof of concept showing how symbolic links can be used to escape a folder for which users granted fine-grained file system permissions.

modify important scripts on the disk, e.g., `.bashrc`, to inject arbitrary commands, which upon system restart, are further executed outside Deno's sandbox. Malicious code might also try to overwrite important executables like the browser, e.g., `firefox.exe`, to be triggered the next time the users try to use this functionality. We note that in other systems with similar permission systems, e.g., Android, there are additional access control mechanisms [14] to prevent altering sensitive files of the operating system. Thus, coarse-grained writes should be discouraged at all costs or Deno should limit the power of its file system API by blocking certain file paths.

Similarly, coarse-grained file system reads (`--allow-read`) allow reading sensitive files from the disk, which reveal environmental variables or system information, bypassing the corresponding Deno permissions. In Figure 5, we show such an example, in which we read the path environmental variable and the memory capacity of the underlying system. This information is normally guarded by `--allow-env` and `--allow-sys`, respectively.

C. Failures in enforcing fine-grained permissions

Fine-grained permissions allow to specify a single target entity for which the permission is granted. For example, file system permissions can be granted for a single file or folder, process spawning for a particular command, and network permissions for a specific domain and port. We note that there is an indirection layer between the names developers use in this fine-grained permissions and the actual resource they aim to control. For example, attackers can abuse DNS resolution to perform scans of local network[9], i.e., they trick the user into granting permission for an attacker-controlled domain and redirect the traffic from that domain to the user's local network/IPs, for which no permission was granted. Similarly, when permission is granted for a particular command like `cat` on UNIX, attackers can abuse a write permission to add a malicious binary on the user's `PATH` and hence hijack the execution of the builtin command.

[9]https://github.com/denoland/deno/issues/21227

We also found a more subtle indirection caused by symbolic links. Assuming a user grants permission for a folder containing a symbolic link, Deno incorrectly considers all the files reached through that symbolic link as part of the folder's substructure. In Figure 6 we show how a malicious actor could abuse this flaw in Deno's enforcement. Let us assume a user checks out an untrusted GitHub repository containing a Deno application. The documentation specifies that the code should be run with read and access permissions for the `cache` folder, which is also part of the repository, thus, attacker-controlled. Even the most security-conscious people would probably consider this operation safe, considering Deno's claim that it can safely run untrusted code[10]. However, the attackers can create inside the cache folder a symbolic link to the root folder of the operating system. This link is automatically created when the users clone the attacker-controlled-repository. At runtime, the attacker-controlled code can navigate the symbolic link and read or write any file on the disk, i.e., it obtained coarse-grained read and write permissions, even though the user only granted fine-grained ones for the cache folder. In Section III-B we discuss how these coarse-grained permission enable more powerful attacks such as unpriviledged network requests, read of sensitive system data, or even arbitrary command execution.

We believe that the presented attacks show that Deno's runtime enforcement is far from perfect and that coarse-grained permissions should be discouraged in Deno's ecosystem. We reported our findings to Deno's security team and after a reluctant initial answer, they decided to restrict the import mechanism to a list of untrusted domains. Additionally, a third-party company specialized in vulnerability disclosure issued two security advisories for our findings. We provide more information about the disclosure process in Section V. We now proceed to study Deno's ecosystem in detail, in particular, the declared and used permissions in open-source packages.

IV. An empirical study of deno.land

In this section, we present a measurement study of code reuse in deno.land, emphasizing the potential security implications of the current practices. The results of our study are divided into three main parts. First, we present a study of dependencies in deno.land, where we emphasize the impact of URL-based importing in the proposed distributed software supply chain. This helps us understand if security risks associated with URLs on the client-side translate to deno.land, e.g., expired or unavailable domains. Second, we perform a study of package versioning, including both quantitative and qualitative analysis of Deno packages' immutability, the role of import maps, and the practice of locking packages to a specific version. This helps us assess the risk of security incidents like `left-pad` or `eslint-scope` in deno.land. Finally, we perform a measurement study of the documented and potentially required permissions in deno.land and the accumulation of permissions due to

[10]https://docs.deno.com/runtime/manual/basics/permissions#run-untrusted-code-with-confidence

Domain	#URLs	Percentage of URLs	Direct dependents	Transitive dependents
deno.land	10892	%85.1536	2779	4088
esm.sh	904	%7.0675	221	837
cdn.skypack.dev	214	%1.6731	189	682
cdn.jsdelivr.net	148	%1.1571	15	126
raw.githubusercontent.com	125	%0.9772	159	608
jspm.dev	81	%0.6333	47	233
x.nest.land	51	%0.3987	19	83
unpkg.com	49	%0.3831	36	207
dev.jspm.io	46	%0.3596	30	188
denopkg.com	35	%0.2736	56	214
ghuc.cc	32	%0.2502	12	54
cdn.esm.sh	22	%0.1720	10	116
gist.githubusercontent.com	15	%0.1173	17	56
cdn.pika.dev	14	%0.1095	15	67
lib.deno.dev	10	%0.0782	12	60
cdn.pagic.org	10	%0.0782	2	27
crux.land	9	%0.0704	17	125
cdn.shopstic.com	8	%0.0625	4	28
x.lcas.dev	2	%0.0156	5	39
ghc.deno.dev	2	%0.0156	3	31
cdn.dreg.dev	1	%0.0078	3	15

TABLE II: The list of domains involved in delivering the code in our study, collected while resolving all the transitive dependencies. For each domain, we show the number of URLs pointing to it and the percentage of URLs belonging to that domain, out of all observed URLs. We also show how many packages directly or transitively import code from each domain.

transitive dependencies. This helps us quantify the number of potentially-overprivileged packages, and the developers' preference for coarse-grained or fine-grained permissions.

A. Setup

Our experimental setup comprises three main steps: collect all Deno packages from deno.land, build the dependency graph, and monitor the observed URLs in the dependency graph for several months. To obtain the list of all packages on deno.land, we used deno.land's official API[11]. We downloaded the latest version of each package and its transitive dependencies, on 25th of November 2022. More specifically, we used deno.land stateless API to collect and download deno.land packages on our local server. Subsequently, we employ a web crawler to visit the deno.land entry of each package, based on the URL obtained from deno.land API. The crawler attempts to visit the associated GitHub repository and download the source code of each package. We used `Puppeteer` and the official JavaScript package of GitHub to implement our crawler. As a result, we collected 5,400 packages with associated repositories, which represent the subject of our study.

To build the dependency graph, we harvest all URLs in the import statements of each package. We resolve each URL to the corresponding package, hosted either on deno.land or on other domains. Concretely, we run `deno info URL` command to resolve all external dependencies. If this command finds a package/URL not in our list, we first download that package, then add it to the data set, and continue the process until all transitive dependencies for each package are resolved.

To monitor the collected URLs, we built a script that runs twice a day, once every twelve hours. This script visits each URL and reports the HTTP status code for each visit. We monitored the target URLs starting from 14th of December 2022 to 15th of December 2023. We used the `request` package to visit each URL and log the response and the time for each visit. We consider any status code other than `status_code=200` as a sign of unavailability.

To carry out our investigation of declared vs. used permissions, we collect the permissions from both code and documentation. We also integrate this information with the dependency graph to study the accumulation of permissions. To extract associated permissions from code, we use `ts_morph`, an open-source TypeScript parser. Using the produced syntactic tree, we identify calls to standard Deno APIs and match them with their corresponding permissions. We create the mapping between APIs and permissions using the namespaces in Deno's official documentation[12], i.e, to run `Deno.Env()` API, the Deno runtime requires `--allow-env` permission to access the environmental variables. To collect the permissions from documentation, we parse the `README.md` file of all packages to look for example command-line usages of the package and extract the listed permissions. In some cases, documentation files do not clearly mention the execution command with the permissions, but they mention instead the list of required permissions in natural language. To solve this case, we also look for all Deno's available permissions, using regular expression matching of permission names.

B. Implications of URL-based importing

As mentioned earlier, Deno allows developers to import code from arbitrary URLs. Hence, each such domain acts as

[11] https://apiland.deno.dev/

[12] https://deno.land/api@v1.38.4

a content delivery networks for software packages. We hypothesize that there are non-uniform security practices among these domains, which might incur a variety of security risks. Below, we discuss several such threats, steaming from the decentralised software supply chain model.

a) ***Insecure transport of the code:*** Deno does not mandate the usage of secure transport protocol in package imports, allowing powerful man-in-the-middle attacks that can inject malicious code via hijacked dependencies. We find that 39 packages in our dataset are transmitted via HTTP, while 4,686 packages are imported via HTTPS protocol. For example, `outdent` is a popular package that uses insecure HTTP protocol in one of its import statements. However, `outdent` is used directly by other packages, i.e., `litre` and `ultra`, which in turn, might affect packages like `mesozoic` via transitive dependencies. Thus, dependencies might exacerbate the attack surface caused by insecure channels. To solve this issue, Deno should disallow code imports via HTTP.

b) ***Domain takeover:*** Insecure and expired domains can be hijacked by attackers to inject code via dangling URLs. In Table II, we show all the domains involved in dependency resolution for packages in our data set. While most of them are reputable CDNs with sensible policies in place, we also observe direct links to GitHub code, which can be easily changed, or personal websites like https://lcas.dev/. Two of the domains are permanently unavailable and we find that one of these domains is even available for sale for a low price, since early February 2023. This domain directly hosts three packages: `denopack`, `deno-react-minimal-frontend`, and `deno-react-minimal-fullstack`, which in turn can reach 12 other packages via transitive dependencies, e.g., `pluginutils`. We argue that attackers can probably hijack these packages by registering the mentioned domain and deploy malicious code in all applications relying on them. To prevent this attack, Deno team decided to register the problematic domain in response to our disclosure (V-A).

c) ***URL/package takeover:*** Some domains might allow the owner of the packages to delete them, as well as deletion of user accounts. This might give attackers the capability to look for deleted users or packages of hardcoded packages in the dependency chain and inject them with malicious code. Moreover, deno.land allows to take over package names under some conditions[13]. These conditions are connected to the frequency of package updates, the use of that package, and deno.land support team acceptance. Nonetheless, they might not be fair to some packages with rarely-changing functionality, e.g., `fonction` is a popular library whose last update was more than two years ago. We argue that such packages might be taken over, if an attacker uses sophisticated social engineering techniques to convince the deno.land team to replace the package with a malicious one. Nonetheless, package takeover is more likely to happen due to careless handover of maintainer rights[14].

[13] Package takeover is allowed under the conditions mentioned in the Q&A of https://deno.land/x

[14] https://github.com/denorg/qrcode/issues/7#issuecomment-1743566836

d) ***Unavailable URLs:*** In Figure 7, we show the availability of the URLs in our data set, over several months. We find that the median number of unavailable links is 220, directly impacting 380 packages. And in turn, unavailable packages affect 462 packages via transitive dependency. The more interesting part is that breaking changes are not limited to third-party packages: we find 21 permanently broken URLs redirecting to the standard library of Deno, `std`. We also find anecdotal evidence for this, in bug reports of popular packages[15] caused by breaking URL changes to the standard library. Figure 7 also shows that the transitive and direct dependencies are very sensitive to small variations in the number of unavailable URLs. For example, on 23rd of September 2023, 283 unavailable URLs resulted in 1,332 unavailable packages. However, `std` was the major culprit for the increase in unavailable packages on that date. We mention that we manually verified the unavailability of this package in the browser, on that day, so we are confident it was a short deno.land fluke, rather than a methodological problem. We find a bug report for deno.land[16], in which others reported similar problems with Deno's code distribution. We also mention that 59.44% of the total unavailable URLs on that day were available the day before, so it was a rather unusual, but coherent event, i.e., many other URLs were still available on the problematic day. Another example is on 5th of March 2023, when 276 unavailable URLs resulted in 1,015 unavailable packages. Packages from `denopkg.com` – which went down on that day – play a significant role in the ecosystem, with four packages from that CDN causing an increase in unavailable packages of 10.8%, versus the day before.

Let us now discuss a concrete example to illustrate how permanently broken URLs can lead to serious availability incidents. When attempting to install version 0.5.7 of `snel`, a relatively popular package with hundreds of GitHub stars:

```
import { VERSION as svltVersion } from "https://deno
    .land/x/snel@v0.5.7/compiler/build.ts";
```

Deno fails while trying to retrieve the dependency `deno-rollup` from `denopkg.com`. This package distribution failure was fixed by migrating away from the third-party domain back to deno.land[17], for retrieving the problematic dependency. While this failure would not be possible if the dependent package would be hosted on deno.land, it appears that `denopkg.com` has a more relaxed unpublishing policy and its effect can propagate to packages hosted on deno.land.

Overall, from the above results, we can say that deno.land packages play a major role on Deno's software supply chain, especially `std` as a standard library. Additionally, we note the fragility of the entire ecosystem: certain domains can go down on some days, making all packages that transitively depend on them unavailable, a phenomenon we observed multiple times. Thus, we recommend mirroring popular packages that are hosted on flaky domains.

[15] https://github.com/eveningkid/denodb/issues/379

[16] https://github.com/denoland/deno/issues/24260

[17] https://github.com/crewdevio/Snel/pull/60

C. Package versioning

There is a well-known tension between keeping dependencies updated for security reasons and avoiding breaking changes by locking them to a specific version. Below, we measure code reuse that implements dependency locking. Additionaly, drawing from npm's lessons learned, deno.land advocates for immutable package versions, i.e., once a version is published on deno.land, it cannot be amended further. However, as seen above, other domains involved in the code distribution might not use such strict publication policies, hence, packages on deno.land might still be mutated via dependencies. Below, we aim to quantify this risk for the packages in our data set.

a) ***Locked vs. unlocked dependencies:*** We first extract the URL fragment corresponding to the locked version, from all the 10,544 unique URLs in the data set. We craft specific regular expression that identifies the packager versions, for each of the domains in the dataset. For example, dependency locking on deno.land, and denopkg.com can be indicated by the presence of "@" right after the package name and followed by the package version, e.g., std@0.213.0. On other domains like GitHub, it can be identified by a hash value in the URL. We find that 13% of the total URLs are not locked to specific package versions. This can lead to serious availability issues when the latest version removes certain files. For example, the fairly popular package `casualdb` imported its dependencies using unlocked URLs, which led to build failures[18]. In response, the developers locked the imports to a specific package version. However, this might incur significant delays in adopting critical security fixes. To the best of our knowledge, Deno does not provide builtin support for semantic versioning and tooling for automated dependency updating, as npm does.

b) ***Mutable package versions:*** We study the prevalence of domains that allow the author to change the package version after publishing it. We find that among the top 16 domains, five domains allow to change the the exact version of a package. To assess if a domain has this capability, we manually published our test package on each domain, and subsequently tried to change the code for a published version of our package. We find that GitHub allows to host code and share it with other domains and CDNs by representing the code as *raw content* link. However, GitHub does not guarantee the immutability of such URLs, since it allows the repository owners to change and update the exact version by using *amend* and other similar commands to rewrite history of a repository. Other domains i.e., denopkg.com, ghc.deno.dev, and ghuc.cc are indirectly affected by mutable package, since these domains act as mirroring domains for GitHub with various reformatings of packages' source code to facilitate Deno to import specific package files through them. cdn.jsdelivr.net allows the users to import code from several domains, e.g., GitHub, npm, esm.sh, skypack, unpkg. Although this CDN might interact with domains which allow to change package versions after

[18]https://github.com/campvanilla/casualdb/issues/16

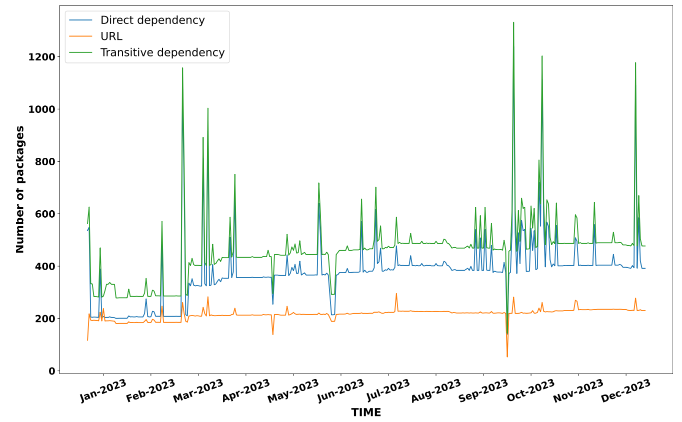

Fig. 7: The number of unavailable URLs across time and the number of affected packages at direct and transitive dependency levels.

publication, it caches the package into their domain where it maintains the code as it is without referring to its domain of origin. This allows cdn.jsdelivr.net to be resistant to version mutation, regardless of the domains which it delivers packages from. Unfortunately, we face issues with studying other domains due to lack of documentation, e.g., cdn.pagic.org, or main page not available, e.g., x.lcas.dev. However, we argue that such domains probably allow version mutation as well, since they are often personal pages, rather than content delivery networks.

To study the impact of mutable package versions on the entire software supply chain, we collect all packages in our dataset, which are hosted on domains that allow package mutability, and then, measure the influence of these packages on their transitive dependencies. This can give an insight into the amplification effect of reusing a mutable package version on the client package, possibly resulting in breaking change or security risks. We find that 224 packages are hosted by one of the domains that allow package mutability. Moreover, there are 807 packages that import one or more mutable packages transitively. denopkg.com is one of the domains that allow package mutability. This is because it acts as a GitHub mirror, and GitHub does not guarantee immutable content after publication. We found that out of 807 packages with at least one mutable package in the dependency chain, 74 packages are affected by a transitive dependency hosted on denopkg.com. For example, `sha1`[19], a popular package on deno.land has a transitive dependency hosted on denopkg.com, called `chiefbiiko`. This could affect the popular package, `sha1`, since mutating `chiefbiiko` could cause breaking changes on the dependency chain.

D. Declared vs. used permissions

To analyze possible usability issues of the security mechanism, we compare the permissions implied by the code of Deno packages and the requested permissions in their

[19]https://deno.land/x/sha1@v1.0.3

documentation. This helps us estimate the number of potentially over- or under-privileged packages. We also study how permissions accumulate due to transitive dependencies.

*a) **Documented permissions**:* We remind the reader that Deno requires users to explicitly grant permissions as command line arguments, for each privileged API invocation. In the absence of such permissions, the runtime execution is paused upon encountering a call to such an API and users are prompted to obtain consent to proceed. We observe that package creators often document the required permission for their package to assist consumers of the library with running the code with the necessary permissions. Thus, we perform a measurement of all the declared permissions in the documentation of the package. The main documentation format used in deno.land are markdown readme files, thus, we first collect all `README.md` files for each package in our dataset. If such a file does not exist for a given package, we assume that that package has no documented permission. Then, we parse each documentation file and attempt to locate the permission flags using regular expression matching. Since Deno permission flags are highly-specific, we do not expect that they would appear in the documentation of the packages by chance, i.e., for other purposes than to inform the clients about the required permission. Nonetheless, to verify this hypothesis, we sampled 50 packages with documented permissions and verified that indeed the documentation mentions that Deno should be run with the documented permissions, when using the target package. We do not encounter any false positive or false negative during our manual analysis. We distinguish between coarse-grained permissions vs. fine-grained permissions by locating the equal sign after the permission name, since all fine grained permissions are specified using the target immediately after the equal sign e.g., `allow-write="./log.txt"`.

We show the inferred permissions from documentation in column two and three of Table III. We find that 1,123 packages document the required permissions, out of which 53 use fine-grained permissions. 33 packages even instruct the user to run deno with all permissions enable, which is a clear indication of over-privileged packages. We believe that the tendency to prefer coarse-grained permissions over finer-grained ones is due to the poor usability of Deno's permission system. That is, when using fine-grained permissions, developers must provide an explicit command line argument for each allowlisted resource, which might lead to tens or even hundreds of command line arguments.

*b) **Inferred permissions from code**:* We collect the permissions from the source code of each package by inspecting the source code and looking for calls to privileged APIs, which trigger a permission check at runtime. Concretely, we parse each TypeScript file of a package using `ts-morph`[20], and locate invocation nodes corresponding to Deno built-in APIs, e.g., `Deno.readTextFile()`. We then map each such API to the corresponding permission flag using Deno's documentation [26]. In total, our static analysis maps 103 APIs to eight corresponding permissions.

[20]https://ts-morph.com/

```
1 import { Input, prompt,} from "https://deno.land/x/
      cliffy@v0.25.4/prompt/mod.ts";
2
3 const result = await prompt([{
4     message: "What's your name?",
5     suggestions: ["John", "Fritz"],
6     type: Input,
7     //files: true
8 }]);
```

Fig. 8: Example usage of the `cliffy` package, showing an unexpected permission prompt. When uncommenting line 7, Deno will ask the user to grant a runtime permission.

We present the inferred permissions from code in column four of Table III. We find that both file system permissions (`--allow-read` and `--allow-write`) are the most used permissions, which is not entirely unexpected. However, the relatively large number of permissions corresponding to running subprocesses (`--allow-run`) suggests that developers often run code that is not mitigated by the permission system [12], resulting again in over-privileged Deno applications. We also note the large difference between permissions inferred from code and documentation, suggesting that package developers often do not make explicit the required permissions for their packages. This, in turn, might lead to decreased usability, as Deno will then often pause at runtime to obtain additional permissions for undeclared API calls in library code. Let us consider the case of `cliffy` in Figure 8, a package that allows developers to easily build complex command line applications. It allows creating prompts with input suggestions for each message. When the `files` option is enabled in line 7, Deno requires a file system read permission. We did not find this information anywhere in the documentation. However, our code-based permission inference identified a `readDir()` invocation in the package's source code. Such poorly-documented APIs can surprise users of Deno application with runtime permission checks, even in production.

*c) **Accumulation of permissions**:* Building reusable Deno packages often involves depending on other packages. We hypothesize that this might have serious usability implications for the user, as illustrated by the example above. On average, we measure that a typical package in our data set depends on 2.94 packages directly and on 3.14 additional ones transitively. We study the accumulation of permissions through dependencies by aggregating for each package all the permissions of all the transitive dependencies, either inferred from code or documentation. Columns 5-7 in Table III show the accumulation effect for each Deno permission. We find that dependencies might use many privileged APIs, which are rather rare in the actual code of the package. For example, we only infer the presence of network-related API calls in 237 packages, but in the dependency code of 2,714 packages. Even though our analysis is coarse-grained and certain parts of the dependency code might not be used, we note the difficulty

Permission	Package-only analysis			Including transitive dependencies		
	Documentation		Package	Documentation		Package
	Coarse-grained	Fine-grained	Code analysis	Coarse-grained	Fine-grained	Code analysis
–allow-all	33	N/A	N/A	39	N/A	N/A
–allow-write	246	30	1031	283	36	2729
–allow-read	409	36	1491	491	48	2743
–allow-env	177	18	574	213	18	2729
–allow-ffi	16	0	42	75	0	139
–allow-sys	1	0	20	1	0	30
–allow-run	148	12	532	178	13	2713
–allow-net	385	29	237	488	55	2714
–allow-hrtime	11	N/A	177	13	N/A	2721

TABLE III: Statistics about the usage of different permissions, for the packages in our data set. We infer the permissions in two ways: by extracting Deno command line arguments from the documentation of the package or by static analysis of the package's code to infer calls to privileged APIs. We show results for both the number of packages directly requiring a permission (columns 2-4) and for the number of packages whose dependencies require a given permission (columns 5-7).

in assessing if a particular third-party API call will trigger a permission check at runtime or not. We see a similar effect for documentation-inferred permissions, where we measure that only 409 packages explicitly ask for a file system read, even though, there are 82 more packages for which one of their dependencies explicitly ask for this permission. Let us consider the case of the `open` package[21] to illustrate why this might cause usability issues for users. This package advertises itself as a solution for opening URLs or executables, using the corresponding operating system functionality, e.g., a browser for opening URLs. Since this package does not list the required permissions in the documentation, users need to use their best judgment when constructing static policies. A good guess would be to grant an `--allow-run` permission, assuming that the package's code runs custom operating system commands. This is indeed a valid hypothesis, and running the following code example with the run permission executes flawlessly on Windows:

```
import { open } from 'https://deno.land/x/open/index
    .ts';
await open('https://google.com');
```

However, when running the same code on Linux, it requires an additional file system read permission, due to a call to `is_wsl` package[22]. This package, in turn, properly documents the need for an `--allow-read` permission, but this information is not propagated in the dependency chain to the users of `open`. Thus, we advocate for better automated tools that assist the users with constructing comprehensive security policies that grant the necessary static permissions, for smooth execution of third-party code.

V. Discussion

Our results show that the two Deno promises listed in Section II-A are partially broken. Running untrusted code without code audit might result in total compromise of the underlying system. In particular, supply chain attacks can leverage careless file system permissions for executing code outside of the sandbox (Section III-B) or for performing arbitrary network requests (Figure 4). While running Deno without any permissions would indeed allow users to see any *explicit* attempt to invoke sensitive APIs, due to shadow permissions, users will have a hard time understanding the implications of grading certain permissions, e.g., write permission to a JavaScript file that is later loaded in the runtime.

Below, we discuss ways of improving Deno to address the identified shortcomings: we start with describing our responsible disclosure process and the changes in Deno directly triggered by our work, we then present additional suggestions for improving the security of Deno's ecosystem, and close with a discussion of future work ideas.

A. Vulnerability disclosure and chances to the runtime

While some of our findings can be considered weak parts of Deno's design, which the security team might simply embrace, we believe that others are clear implementation bugs that should be fixed as soon as possible. Following Deno's security policy, we reported our import abuse attack and shadow permission findings, in early 2023. We note that these problems of the permission system are mostly Deno-specific and were never discussed in the literature. However, Barrera et al. [18] warned about the dangers of coarse-grained permissions and the lack of expressiveness in permission systems. Also, the dangers or URL-based code imports, e.g., expired domains [27], were previously reported in the client-side, but never in the context of server-side code.

After a lively and positive exchange of emails with the Deno team, in which we proposed ways of remedying the discovered issues, e.g., adding a blocklist of sensitive resources for file system operations, the Deno team stopped responding to our messages. We then reached out to a third-party company specializing in vulnerability disclosure, in the end of 2023, to report the same findings plus the symbolic link attack. They confirmed that the problems are reproducible and they considered them as security issues. They also issued two security advisories for our findings: CVE-2024-21487 – a

[21] https://deno.land/x/open@v0.0.6
[22] https://github.com/skoshx/deno-open/blob/7645fe0efdeabbcdeece24c8d6159e4ab8447ff3/index.ts#L121

high severity advisory for the symbolic link findings, and CVE-2024-21486 – a moderate severity advisory for the static import issue. They also proceeded to disclose the findings to Deno directly.

To our surprise, in this instance, the Deno team was more cooperative, promising to restrict the unprivileged static imports to a list of trusted domains in Deno version 2.0. Moreover, they also decided to purchase the problematic domain described in Section IV-B to prevent package hijacking. We also directly contacted the maintainers of the packages on deno.land that transitively import code from this domain to directly notify them about the risk of hijacking. However, at the time of writing none of them responded to our request. Thus, thanks to a successful disclosure process, our work has a positive impact on all Deno users, leading to concrete fixes and improvements in future versions of the runtime.

B. Further recommendations

Additionally, to make the associated network requests explicit, we recommend that Deno considers adding support for fine-grained import permissions, as proposed for Node.js by Vasilakis et al. [10]. However, we acknowledge that this might be cumbersome to implement, and it can break backward compatibility. A quick and dirty fix against the attack proposed in Figure 4 is to somehow track what files changed and prevent those from loading later in the runtime. Alternatively, Deno's cache could keep track of static imports for packages and prevent new ones from showing up at execution time. Another idea is to use the existing import maps as a contract between package producers and consumers. That is, packages are only allowed to import what is declared upfront in these files and the files' integrity is protected using hashes of the file content.

We also stress that Deno should consider the adoption of compartmentalization [11], [10], [28] to separate code coming from different domains and offering support for manifest files that specify fine-grained permissions per module. We note that there were several open issues for Deno asking for exactly this feature[23]. Since Deno effectively advocates for a client-side mindset in an out-of-browser JavaScript environment, we believe that it is only natural that it also reuses some of the well-known browser security mechanisms. Isolating untrusted code from suspicious domains with an `iframe`-like primitive is one of them, another being the adoption of a configurable security policy akin to CSP to allow users to configure various aspects of the runtime, e.g., trusted domains from which code can be loaded.

While we salute Deno's proposal of a fully decentralized software supply chain, which promises increased resilience, we note that in its current form, deno.land is quite fragile, as shown in Section IV-B. It is especially concerning the reliance on low-quality domains and the non-uniform version amending policy, across domains. We recommend that deno.land employs vetting of domains, as promised by the Deno team during the disclosure process, and warn users or even drop the request when they attempt to use an unverified domain. Efforts should also be invested in uniforming the (security) policies of such domains, e.g., to prevent updates to published package versions, across the ecosystem. Finally, for the most depended-upon packages that are hosted outside deno.land, we also recommend automatic mirroring to increase the resilience of the supply chain. A similar proposal was made by the community in the form of immutability proxy[24], but it is still to be adopted by the deno.land team, which considers it a "userland issue".

[23]https://github.com/denoland/deno/issues/3675

[24]https://github.com/denoland/deno/issues/3616

C. Future work

During our study, we observed poor tooling support in Deno for certain security-relevant tasks. First, the proposed decentralized software supply chain comes with big challenges for the widely-adopted practice of semantic versioning. Pashchenko et al. [29] find that developers often struggle with this task, in the context of modern software supply chains. We could not find any good tool for Deno that assists the user with automatically updating dependencies. As seen in Table I, such a tool needs to support different domains, using different URL fragments for marking the version of the imported code. We also note in Section IV-D the challenge of building meaningful security policies, when dealing with third-party code. It is extremely challenging for users to understand which permissions a particular API call needs. Hence, future work should propose automatic tools for inferring such permissions, e.g., using testing, and for documenting these permissions in a way that IDEs can consume and communicate them to the users. Wang et al. [17] perform a similar study for runtime permissions in Android, but they do not target library code in their testing. Finally, we also note that future work should carry user studies with actual Deno users to understand how developers use the permission system, potential pain points, and whether their self-written policies result in over- or under-privileged Deno applications, in particular, when reusing third-party code from deno.land. Our results in Section IV-D suggest that these packages are poorly documented and they might lead to over-privileged applications, as Felt et al. [15] report is often the case in Android.

VI. Related work

In this section, we survey related work on permission systems and their limitations, followed by JavaScript isolation, software supply chain security and client-side security.

a) ***Permission systems:*** Permission systems have been implemented and successfully deployed in other programming languages, for various security reasons, e.g., in C/C++ to isolate the virtual memory [30]. Leontie et al. [31] propose a permission system that consists of both hardware and software support to make C++ compiler more secure, while assigning pointers for *`private`* variables and functions in OPP setup. Vasilakis et al. [32] proposes a permissions system to control the communication of JavaScript programs with third-party

code hosted in a separate process. Heule et al. [33] proposes a mandatory permission systems for browser extensions to protect the users' privacy from third-party code that could be abused by the browser extensions, especially in the context of over-permissioned extensions. Felt et al. [34] evaluate the benefits of permission systems, in the context of Android applications and browser extensions. The study shows a net positive benefit from adopting such a system, in both considered use cases. Marouf et al. [35] proposes REM, a browser extension implemented to monitor, at runtime, the used privileges by the other browser extensions and report them to the user, so that they can better control these extensions' permissions. Zhang et al. [36] proposes VetDroid, a dynamic analysis framework to monitor the used permissions by the running app. Reardon et al. [37] discusses another framework to monitor mobile apps behaviour at the runtime, the key difference being the support for monitoring the network activity alongside the covert channels of the tested apps to look for mobile apps permissions abuse. Finally, Scoccia et al. [38] show that bug reports about the Android apps usability involving permission violations are prevalent in open source projects, but that developers tend to quickly address such issues. We believe that Deno can adopt many of the features present in these prior permission systems. In particular, the runtime should adopt Android-style manifests to better manage fine-grained permissions and import permissions, as proposed in MIR [10].

*b) **Shortcomings of permission systems**:* Davi et al. [16] were the first to report that Android's permission system allow privilege escalation attacks via inter-app function invocations of privileged functionality. Moreover, Orthacker et al. [39] argue that Android's permission system is ineffective in the presence of colluding apps that join their permissions. This is mainly enabled by permission redelegation, as defined by Felt et al. [40]. Bugiel et al. [41] propose a solution to these problems by interposing runtime monitors on inter-app calls and by using mandatory access control at file system level. Dawoud and Bugiel [28] propose an even stricted capability-based model that isolates untrusted parts of an app at process level. Other weak parts of the Android permission system that can be exploited by attackers are custom permissions [42], preinstalled apps [43], and extensions to the default access control policy [14]. All this amendments of the standard security model leave the door open to new attacks and require a dedicated approach to detect their misuses. Felt et al. [15] measure that most Android applications are overprivileged, mainly due to poor documentation of certain features. Wang et al. [17] show how unexpected runtime permissions can cause crashes of Android apps or other usability issues. Barrera et al. [18] measure permission usage in Android and identify multiple expressiveness issues, e.g., too coarse-grained permissions, lack of hierarchical permissions. Olejnik et al. [44] propose improving Android's availability by predicting permission granting at runtime. Roesner et al. [45] propose another improvement in which access control gadgets are guarding UI-level actions. While a great source of inspiration for our work, none of these papers target Deno or a similar server-side runtime that allows URL-based code imports.

*c) **Isolation and compartmentalization**:* Multiple techniques can be used to isolate JavaScript code, at various levels of the software stack. For example, JSand [46], Ahmadpanah et al. [47], NodeSentry [48], JaTE [49] implement isolation at the language level. They use techniques like membranes, shadow global functions, functions rewriting and wrappers. A more relevant technique for Deno's sandbox, is to combine isolation with permission systems to implement both isolation and communication control between the host and the untrusted code. For example, DecentJS [50] supports security policies on top of isolation to limit the functionality of JavaScript code. Similarly, Ferreira et al. [51] show how permissions systems and program rewriting can be used to defend against supply chain attacks. However, recent work by AlHamdan and Staicu [52] shows that these techniques are ineffective in practice. There are also OS-level isolation techniques for JavaScript, which are more generic, but they might incur a larger overhead. Isolation at the OS level can be done by running the guest code in a different runtime [53], compartmentalize the guest code by launching new web workers [54], isolate the untrusted code in a different operating system process [11], or intercept operations from the guest code by modifying the browser [55]. Isolation of JavaScript code can also be done at lower levels, e.g., sandboxing the x86 code with Native Client [56], isolating libraries in Firefox browser in a lightweight sandbox [57] to reduce the impact of component compromise, or even adding hardware support for isolation by using both protection keys and binary inspection [58]. Preventing memory corruption attacks can be done using memory-access permissions that implement in-process memory isolation [59], or adding more programming languages constructs to isolate untrusted code [60]. Moreover, Wyss et al. [23] propose to monitor and limit the capabilities of install-time hooks via system call filtering, to prevent supply chain attacks. While related, none of this work concerns Deno's permission system. Abbadini et al. [12] is the only prior work that follows this goal, by extending Deno's permission system with additional restrictions for subprocesses. Concretely, they use Landlock and eBPF Linux kernel modules to enforce low-level privilege reduction. However, they do not study the security of Deno's existing runtime enforcement, nor the nature of code reuse in deno.land.

*d) **Software supply chain security**:* Code reuse allows developers to cut their development cost by leveraging massive amounts of open-source code. However, widely-reused code can also be a good target for adversaries to inject malicious code, a security incident that was recently termed supply chain attack. There exist recent studies on the software supply chain security and attacks in several programming and scripting languages, i.e., npm [61], [7], [1], PyPI [62], and RubyGem [63]. So et al. [64] discuss key insights to implement robust integrity for the software supply chain. Zahan et al. [65] presents a method to study the weak links in the npm software supply chain through package metadata, which could help the developers to reduce the possibility of

failure in supply chain attacks. Zimmermann et al. [1] study vulnerable code reuse, and supply chain attacks, and show that the average npm package relies on tens of other packages and maintainers, transitively. Considering that many of the code in transitive dependencies is actually not used, Koishybayev and Kapravelos [66] propose reducing the attack surface of Node.js applications by removing dead code, a process they term debloating. Rabe et al. [67] present a study on trivial npm packages and measure the developers' awareness of using them in terms of benefits and challenges. Duan et al. [7] perform an extensive study of the software package managers for the interpreted languages and propose a hybrid program analysis technique to detect malicious packages that perform unwanted actions. Additionally, Ohm et al. [68] present a study on a set of supply chain attack detection mechanisms and where to employ each of them. Shcherbakov et al. [20] show that prototype pollution is a serious security problem in Node.js, which can be elevated to remote code execution. Considering our results in Section II-B, we expect that this is the case at least partially for Deno as well. Li et al. [21] propose a sophisticated static analysis solution for detecting prototype pollution, which they then generalize to other security problems [22]. Recently, Rack et al. [69] present the first large-scale study of JavaScript bundles prevalence and employ static analysis to reverse engineer bundles and study code provenance. They show that a typical bundle includes tens of third-party dependencies, which should be separated and reasoned about in isolation. None of this work considers the possibility of supply chain attacks in Deno and the weaknesses of the permission system that such an attack might exploit.

*e) **Client-side security:*** Lauinger et al. [27] show that domain takeover via malicious re-registration of expired domains is common. In our work, we show that this problem also affects Deno and it can lead to malicious code injection in otherwise immutable packages. Browser security and extension security are also related topics, with infamous attacks like XSS, XSRF and attacker-controlled extensions. Stock et al. [70] present a longitudinal study of the evolution of client-side security vulnerabilities and CSP adoption. Agarwal et al. [71] perform a study of how browser extensions temper security-relevant HTTP headers. Fass et al. [72] present DoubleX, a static analysis tool that aims to detect security and privacy threats in browser extensions, by detecting suspicious data and control flows using a sophisticated extension dependence graph. Roth et al. [73] perform a study of the subtle variations in CSP policies based on factors like user agent or geographic locations. Rautenstrauch et al. [74] developed an automated approach to detect observable channels that lead to cross-site leaks in web browsers. However, none of this work is directly applicable beyond the client-side environment, e.g., to Deno.

VII. Conclusion

In this paper, we study Deno, a popular, emerging JavaScript runtime that promises security by design. We show that thanks to its default-on permission system, Deno has indeed a lower attack surface than other runtimes for server-side code. However, we find that, despite its promise, Deno still allows certain supply chain attacks due to weaknesses in its permission system. We also raise serious questions about the utility and risks of the URL-based importing mechanism, which led to restrictions of this feature in future versions of Deno. Moreover, we identify multiple challenges posed by decentralized software supply chains outside of a browser: difficulty to guarantee transitive immutability for dependencies, challenging version control, and availability issues due to transient URLs. Finally, we highlight Deno's usability issues that prevents effective specification of policies and the communication of required permissions for open-source packages. We disclosed all our findings to Deno's security team, which lead to the publication of two security advisories for Deno. We also provide concrete guidelines for remedying the discovered security problems.

We recommend that Deno adds permission checks for third-party code imports and fixes its support for symbolic links to enable sound enforcement of fine-grained file system permissions. We also recommend refining the coarse-grained permissions to blocklist certain file paths or network URLs and thus, prevent shadow permissions. While we believe that Deno's decentralized software supply chain has a lot of potential, we also think that the community should develop mirroring solutions to deal with the problem of transient URLs. Considering the many similarities between Deno and browser environments, we advocate for adopting security mechanisms akin to the ones in browsers: compartmentalizing application code with iframe-like separation or fine-grained CSP-like policies that allow configuring permissions per library or origin. All in all, we hope that our work contributes to a better understanding of Deno's security model and provides a clear road map for improving it.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable feedback, and Snyk for assisting us with the vulnerability disclosure. We also like to thank Deno team, in particular, Luca, Alon Bonder, and Kaleigh Hedges, for their support in proceeding with the project. This work was conducted in the scope of a dissertation at the Saarbrücken Graduate School of Computer Science.

References

[1] M. Zimmermann, C. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *USENIX Security Symposium*, 2019.

[2] C. Staicu, M. Pradel, and B. Livshits, "SYNODE: understanding and automatically preventing injection attacks on NODE.JS," in *Network and Distributed System Security Symposium (NDSS)*, 2018.

[3] J. C. Davis, E. R. Williamson, and D. Lee, "A sense of time for javascript and node.js: First-class timeouts as a cure for event handler poisoning," in *USENIX Security Symposium*, 2018.

[4] J. C. Davis, C. A. Coghlan, F. Servant, and D. Lee, "The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale," in *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2018.

[5] C. Staicu and M. Pradel, "Freezing the web: A study of ReDoS vulnerabilities in JavaScript-based web servers," in *USENIX Security Symposium*, 2018.

[6] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "Sok: Taxonomy of attacks on open-source software supply chains," in *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 2023, pp. 1509–1526. [Online]. Available: https://doi.org/10.1109/SP46215.2023.10179304

[7] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards measuring supply chain attacks on package managers for interpreted languages," in *Network and Distributed System Security Symposium (NDSS)*, 2021.

[8] D. L. Vu, F. Massacci, I. Pashchenko, H. Plate, and A. Sabetta, "Lastpymile: identifying the discrepancy between sources and packages," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. ACM, 2021, pp. 780–792. [Online]. Available: https://doi.org/10.1145/3468264.3468592

[9] Y. Gu, L. Ying, Y. Pu, X. Hu, H. Chai, R. Wang, X. Gao, and H. Duan, "Investigating package related security threats in software registries," in *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 2023, pp. 1578–1595. [Online]. Available: https://doi.org/10.1109/SP46215.2023.10179332

[10] N. Vasilakis, C.-A. Staicu, G. Ntousakis, K. Kallas, B. Karel, A. DeHon, and M. Pradel, "Mir: Automated Quantifiable Privilege Reduction Against Dynamic Library Compromise in JavaScript," in *Conference on Computer and Communications Security (CCS)*, 2021.

[11] N. Vasilakis, B. Karel, N. Roessler, N. Dautenhahn, A. DeHon, and J. M. Smith, "BreakApp: Automated, Flexible Application Compartmentalization," in *Network and Distributed System Security Symposium (NDSS)*, 2018.

[12] M. Abbadini, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi, "Cage4deno: A fine-grained sandbox for deno subprocesses," in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 149–162. [Online]. Available: https://doi.org/10.1145/3579856.3595799

[13] F. Brown, S. Narayan, R. S. Wahby, D. R. Engler, R. Jhala, and D. Stefan, "Finding and preventing bugs in javascript bindings," in *Symposium on Security and Privacy (S&P)*, 2017.

[14] Y. T. Lee, W. Enck, H. Chen, H. Vijayakumar, N. Li, Z. Qian, D. Wang, G. Petracca, and T. Jaeger, "Polyscope: Multi-policy access control analysis to compute authorized attack operations in android systems," in *USENIX Security Symposium*, 2021.

[15] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. A. Wagner, "Android permissions demystified," in *Conference on Computer and Communications Security (CCS)*, 2011.

[16] L. Davi, A. Dmitrienko, A. Sadeghi, and M. Winandy, "Privilege escalation attacks on android," in *International Conference on Information Security (ISC)*, ser. Lecture Notes in Computer Science, vol. 6531. Springer, 2010, pp. 346–360.

[17] S. Wang, Y. Wang, X. Zhan, Y. Wang, Y. Liu, X. Luo, and S. Cheung, "APER: evolution-aware runtime permission misuse detection for Android apps," in *International Conference on Software Engineering (ICSE)*, 2022.

[18] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji, "A methodology for empirical analysis of permission-based security models and its application to Android," in *Conference on Computer and Communications Security (CCS)*, 2010.

[19] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC'19. USENIX Association, 2019.

[20] M. Shcherbakov, M. Balliu, and C.-A. Staicu, "Silent spring: Prototype pollution leads to remote code execution in Node.js," in *USENIX Security Symposium*, 2023.

[21] S. Li, M. Kang, J. Hou, and Y. Cao, "Detecting Node.js prototype pollution vulnerabilities via object lookup analysis," in *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021.

[22] ——, "Mining Node.js vulnerabilities via object dependence graph and query," in *USENIX Security Symposium*, 2022.

[23] E. Wyss, A. Wittman, D. Davidson, and L. D. Carli, "Wolf at the door: Preventing install-time attacks in npm with latch," in *Asia Conference on Computer and Communications Security (ASIA CCS)*, 2022.

[24] C.-A. Staicu, S. Rahaman, Á. Kiss, and M. Backes, "Bilingual problems: Studying the security risks incurred by native extensions in scripting languages," in *USENIX Security Symposium 2023*, 2023.

[25] D. Lehmann, J. Kinder, and M. Pradel, "Everything old is new again: Binary security of WebAssembly," in *USENIX Security Symposium*, 2020.

[26] deno.land, " Deno's runtime APIs ," https://deno.land/api@v1.39.1.

[27] T. Lauinger, A. Chaabane, A. S. Buyukkayhan, K. Onarlioglu, and W. Robertson, "Game of registrars: An empirical analysis of post-expiration domain name takeovers," in *USENIX Security Symposium*, 2017.

[28] A. Dawoud and S. Bugiel, "Droidcap: OS support for capability-based permissions in Android," in *Network and Distributed System Security Symposium, (NDSS)*, 2019.

[29] I. Pashchenko, D. L. Vu, and F. Massacci, "A qualitative study of dependency management and its security implications," in *Conference on Computer and Communications Security (CCS)*, 2020.

[30] Y. Younan, W. Joosen, and F. Piessens, "Runtime countermeasures for code injection attacks against c and c++ programs," *ACM Comput. Surv.*, vol. 44, no. 3, jun 2012. [Online]. Available: https://doi.org/10.1145/2187671.2187679

[31] E. Leontie, G. Bloom, and R. Simha, "Hardware and software support for fine-grained memory access control and encapsulation in c++," in *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity*, ser. SPLASH '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 81–82. [Online]. Available: https://doi.org/10.1145/2508075.2508091

[32] N. Vasilakis, C.-A. Staicu, G. Ntousakis, K. Kallas, B. Karel, A. DeHon, and M. Pradel, "Preventing dynamic library compromise on node.js via rwx-based privilege reduction," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1821–1838. [Online]. Available: https://doi.org/10.1145/3460120.3484535

[33] S. Heule, D. Rifkin, A. Russo, and D. Stefan, "The most dangerous code in the browser," in *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. Kartause Ittingen, Switzerland: USENIX Association, May 2015. [Online]. Available: https://www.usenix.org/conference/hotos15/workshop-program/presentation/heule

[34] A. P. Felt, K. Greenwood, and D. Wagner, "The effectiveness of application permissions," in *2nd USENIX Conference on Web Application Development (WebApps 11)*, 2011.

[35] S. Marouf and M. Shehab, "Towards improving browser extension permission management and user awareness," in *8th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, 2012, pp. 695–702.

[36] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting undesirable behaviors in android apps with permission use analysis," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 611–622. [Online]. Available: https://doi.org/10.1145/2508859.2516689

[37] J. Reardon, Á. Feal, P. Wijesekera, A. E. B. On, N. Vallina-Rodriguez, and S. Egelman, "50 ways to leak your data: An exploration of apps' circumvention of the android permissions system," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 603–620. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/reardon

[38] G. L. Scoccia, A. Peruma, V. Pujols, I. Malavolta, and D. E. Krutz, "Permission issues in open-source android apps: An exploratory study," in *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2019, pp. 238–249.

[39] C. Orthacker, P. Teufl, S. Kraxberger, G. Lackner, M. Gissing, A. Marsalek, J. Leibetseder, and O. Prevenhueber, "Android security permissions - can we trust them?" in *Security and Privacy in Mobile Information and Communication Systems (MobiSec)*, 2011.

[40] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *USENIX Security Symposium*, 2011.

[41] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on Android," in *Network and Distributed System Security Symposium, (NDSS)*, 2012.

[42] R. Li, W. Diao, Z. Li, J. Du, and S. Guo, "Android custom permissions demystified: From privilege escalation to design shortcomings," in *Symposium on Security and Privacy (S&P)*, 2021.

[43] J. Gamba, M. Rashed, A. Razaghpanah, J. Tapiador, and N. Vallina-Rodriguez, "An analysis of pre-installed android software," in *Symposium on Security and Privacy (S&P)*, 2020.

[44] K. Olejnik, I. Dacosta, J. S. Machado, K. Huguenin, M. E. Khan, and J. Hubaux, "Smarper: Context-aware and automatic runtime-permissions for mobile devices," in *Symposium on Security and Privacy (S&P)*, 2017.

[45] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan, "User-driven access control: Rethinking permission granting in modern operating systems," in *Symposium on Security and Privacy (S&P)*, 2012.

[46] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens, "JSand: complete client-side sandboxing of third-party JavaScript without browser modifications," in *Annual Computer Security Applications Conference (ACSAC)*, 2012.

[47] M. M. Ahmadpanah, D. Hedin, M. Balliu, L. E. Olsson, and A. Sabelfeld, "SandTrap: Securing JavaScript-driven Trigger-Action Platforms," in *USENIX Security Symposium*, 2021.

[48] W. D. Groef, F. Massacci, and F. Piessens, "Nodesentry: least-privilege library integration for server-side javascript," in *Proceedings of the 30th Annual Computer Security Applications Conference, (ACSAC)*, 2014.

[49] T. Tran, R. Pelizzi, and R. Sekar, "JaTE: Transparent and efficient JavaScript confinement," in *Annual Computer Security Applications Conference, (ACSAC)*, 2015.

[50] M. Keil and P. Thiemann, "Transaction-based sandboxing for JavaScript," *CoRR*, vol. abs/1612.00669, 2016.

[51] G. Ferreira, L. Jia, J. Sunshine, and C. Kästner, "Containing malicious package updates in npm with a lightweight permission system," in *International Conference on Software Engineering (ICSE)*, 2021.

[52] A. AlHamdan and C.-A. Staicu, "SandDriller: A Fully-Automated approach for testing Language-Based JavaScript sandboxes," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 3457–3474. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/alhamdan

[53] X. Dong, M. Tran, Z. Liang, and X. Jiang, "AdSentry: comprehensive and flexible confinement of JavaScript-based advertisements," in *Annual Computer Security Applications Conference, (ACSAC)*, 2011.

[54] L. Ingram and M. Walfish, "Treehouse: JavaScript sandboxes to help web developers help themselves," in *USENIX Annual Technical Conference (ATC)*, 2012.

[55] M. Zhang and W. Meng, "JSISOLATE: lightweight in-browser JavaScript isolation," in *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2021.

[56] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *Symposium on Security and Privacy (S&P)*, 2009.

[57] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan, "Retrofitting fine grain isolation in the Firefox renderer," in *USENIX Security Symposium*, 2020.

[58] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: secure, efficient in-process isolation with protection keys (MPK)," in *USENIX Security Symposium*, 2019.

[59] T. Frassetto, P. Jauernig, C. Liebchen, and A. Sadeghi, "IMIX: in-process memory isolation extension," in *USENIX Security Symposium*, 2018.

[60] A. Ghosn, M. Kogias, and M. Payer, "Enclosure: Language-Based Restriction of Untrusted Libraries," in *International Conference on Architectural Support for Programming Languages and Operating Systems, (ASPLOS)*, 2021.

[61] S. Scalco, R. Paramitha, D.-L. Vu, and F. Massacci, "On the feasibility of detecting injections in malicious npm packages," in *Proceedings of the 17th International Conference on Availability, Reliability and Security*, 2022, pp. 1–8.

[62] D. L. Vu, I. Pashchenko, F. Massacci, H. Plate, and A. Sabetta, "Towards using source code repositories to identify software supply chain attacks," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 2093–2095. [Online]. Available: https://doi.org/10.1145/3372297.3420015

[63] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17*. Springer, 2020, pp. 23–43.

[64] J. So, M. Ferdman, and N. Nikiforakis, "The more things change, the more they stay the same: Integrity of modern javascript," in *Proceedings of the ACM Web Conference 2023*, ser. WWW '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 2295–2305. [Online]. Available: https://doi.org/10.1145/3543507.3583395

[65] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, and L. Williams, "What are weak links in the npm supply chain?" in *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE Computer Society, 2022, pp. 331–340.

[66] I. Koishybayev and A. Kapravelos, "Mininode: Reducing the attack surface of node.js applications," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[67] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, "Why do developers use trivial packages? an empirical case study on npm," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 385–395. [Online]. Available: https://doi.org/10.1145/3106237.3106267

[68] M. Ohm and C. Stuke, "Sok: Practical detection of software supply chain attacks," in *Proceedings of the 18th International Conference on Availability, Reliability and Security*, ser. ARES '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3600160.3600162

[69] J. Rack and C.-A. Staicu, "Jack-in-the-box: An empirical study of javascript bundling on the web and its security implications," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 3198–3212. [Online]. Available: https://doi.org/10.1145/3576915.3623140

[70] B. Stock, M. Johns, M. Steffens, and M. Backes, "How the web tangled itself: Uncovering the history of Client-Side web (In)Security," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 971–987. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/stock

[71] S. Agarwal, "Helping or hindering? how browser extensions undermine security," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 23–37. [Online]. Available: https://doi.org/10.1145/3548606.3560685

[72] A. Fass, D. F. Somé, M. Backes, and B. Stock, "Doublex: Statically detecting vulnerable data flows in browser extensions at scale," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1789–1804. [Online]. Available: https://doi.org/10.1145/3460120.3484745

[73] S. Roth, S. Calzavara, M. Wilhelm, A. Rabitti, and B. Stock, "The security lottery: Measuring Client-Side web security inconsistencies," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 2047–2064. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/roth

[74] J. Rautenstrauch, G. Pellegrino, and B. Stock, "The leaky web: Automated discovery of cross-site information leaks in browsers and the web," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 2744–2760.