State of Security in JavaScript Runtimes

The Web Security Seminar

Davide Bombelli
Saarland University
dabo00007@stud.uni-saarland.de

***Abstract*—A JavaScript Runtime Environment is the ambient in which JavaScript code is executed. The most common runtime environment for JavaScript is the web browser, where it is included to make web pages reactive. However, other runtime environments exist and are widely used. They allow JavaScript code to be executed outside of the web browser and to build full-stack applications.**

In this paper, we examine the security state of JavaScript runtime environments, with a special focus on Node.js and Deno, the most used ones. Specifically, we highlight the security concerns and implications tied to the choice of a specific runtime environment: the JavaScript engine used, the package distribution policies, and the permission systems. We base our claims on all the previous research conducted on the matter. Since the results of some of the tests conducted in the past might have changed, we recreated a study on domain availability in Deno that was performed before the release of Deno 2.0. Based on our findings, we suggest improvements that would help developers write secure code and mitigate existing vulnerabilities.

When structural mitigation cannot be implemented, detection of vulnerabilities is the best defense that can be used. Another point of study in this paper is prototype pollution and its automatic detection. Prototype pollution is an insidious vulnerability of prototype-based programming languages that allows rewriting objects' properties. We introduce the concept of a gadget and of a sink, two necessary pieces of successful exploitation of prototype pollution vulnerability. Then, we conduct a study using the SecBench.js benchmark. We run the test suite in Node.js and in Deno. The comparison serves to evaluate the results of Deno's effort in protecting against vulnerabilities, including prototype pollution. At last, we investigate the state of automatic and semi-automatic detection systems that have been implemented to help developers secure their websites, and we give our suggestions for future research.

All the work made is then used to answer specific research questions related to the topics covered.

I. Introduction

Despite being created by the effort of a single man in 10 days, JavaScript became one of the most used programming languages in the world. It is now used in both front-end and back-end, inside and outside of the browser. A lot of JavaScript's success is due to its notable elasticity and error tolerance. It is, in fact, a multi-paradigm, dynamic, prototype-based programming language. The syntax is inspired by Java and C languages, and some of the same data structures and principles are implemented. As a consequence, JavaScript supports object-oriented programming with object prototypes and classes, but also functional programming. Functions are treated as objects that can be created via expressions and then can be assigned like variables. While these properties make the language attractive for beginners and amateurs, they come at a cost. JavaScript has 11% of all the vulnerabilities found in programming languages in the last 10 years [46].

Nowadays, JavaScript comes in different flavours: from website graphics to application back-end, from being embedded in IoT devices to being used as a codebase for games. In all cases, the code has to be translated and run in a suitable environment. Concretely, an interpreter is necessary to run JavaScript code. Depending on the context, a different runtime environment has to be used. For example, the V8 engine is used in Chrome browser, while to compile code from the Linux Command Line, Node.js is a more suitable option. All modern engines make use of JIT compilation to make code faster.

The goal of this paper is to gather resources on JavaScript runtime environments, summarize the security directions the language is expanding to, and provide a base to answer the following research questions:

RQ1: Deno Permissions System. What are the advantages and disadvantages of Deno's permission system? What are the tradeoffs between usability and enhanced security? Since Deno introduces a completely new way of approaching security for a JavaScript runtime, it is necessary to look at the adopted system with particular care. We base our answer to this question on previous research on the topic and our experience gained while conducting our study.

RQ2: Deno Attack Surface. Was Deno successful in reducing its attack surface compared to Node.js? We analyze the results of Deno's effort using the number of exploitable vulnerabilities in the security benchmark SecBench.js [16].

RQ3: Deno Supply Chain. Is Deno's approach to package distribution a valid alternative to the traditional supply chain? We analyze the pros and cons of a decentralized supply chain and conduct a study on a possible issue that might arise in such a system.

RQ4: Prototype Pollution in Deno. Are the efforts into mitigating prototype pollution enough? We refer to our study and to the documentation to understand the way in which Deno can be a good alternative and what flaws are still present.

To better understand the topics covered in the paper, we explain the most important topics that we are touching. The next paragraphs and section serve this purpose.

A. *JavaScript Runtimes*

A runtime environment is where a programming language executes. It determines what global objects the program can access, and it impacts how it runs. Each runtime implements and manages in its own way data, variables, functions, and storage. The main components of the JavaScript runtime are: a JavaScript engine, the APIs with which the engine communicates with external resources, and the logic that handles all the background activities (callback queue, microtask queue, event loop).

The most common place where JavaScript code is executed is the web browser. This is possible because each browser embeds a JavaScript engine and acts as a runtime environment. For example, Mozilla developed and uses the SpiderMonkey engine, while Chrome developed and uses the V8 engine. Outside of websites, JavaScript is used in both front-end and back-end. The most common runtime environments for executing JavaScript outside of the browser are Node.js, Deno, and Bun. Node.js and Deno are based on the V8 engine, and Bun is based on JavaScriptCore. To provide even more performance and use case scenarios, most of the engines can now run WebAssembly. Moreover, even embedded systems and IoT devices are now able to run JavaScript through the JerryScript engine [29], [34].

B. *Prototype Pollution*

One of the most typical vulnerabilities that can occur when working with prototype-based languages is prototype pollution. It allows an attacker to inject or modify properties from the object tree at runtime. Based on the targeted property, prototype pollution can lead to other vulnerabilities, including XSS, Denial of Service, and Remote Code Execution.

The exploitation of prototype pollution relies on the use of otherwise benign pieces of code that read attacker-controlled values and execute them. They are called gadgets and can be found in many common libraries and frameworks. The function or DOM element that is accessed through the misuse of a gadget is called a sink.

II. JavaScript Runtimes

The goal of all the runtimes is to execute JavaScript code, but the way to achieve it can differ based on the specific rules and features of the environment. And adopting any of the JavaScript runtimes comes with its security implications. In this section, we take a look at some JavaScript runtime environments outside of the web browsers.

A. *Node.js*

Node.js is an open-source JavaScript runtime environment built on the V8 engine developed by Chrome. It supports WebAssembly and Just-In-Time compilation for speed. It runs in a single process without spawning a thread for each incoming request. It allows blocking paradigms, but the default behaviour when performing an I/O operation is to execute other code while waiting for an answer and to resume as soon as the operation is finished. Node.js can handle thousands of concurrent connections with a single server with relative ease and is shown to be the most reliable runtime over high-throughput applications [37]. A basic HTTP server implementation is shown in Fig. 1. Node Package Manager (npm) is the default package manager for Node.js, and it is a centralized ecosystem of open-source libraries.

Fig. 1. Implementation of an HTTP server in Node

```
const http = require('http');

const server = http.createServer((req, res) =>{
  res.writeHead(200,
      { 'Content-Type': 'text/plain' });
  res.end('Hello from Node.js!');
});

server.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

The server can be launched via the command line:

```
> node node/index.js
```

B. *Deno*

Deno developers took a completely different approach. Deno is a JavaScript runtime environment built on the V8 engine that claims to be so secure that a developer can run any code on it without worrying about security. Deno has a built-in permission system that allows both coarse and fine-grained security, which should mitigate exploits of critical severity. For every imported module, the developer has the option to choose which permissions to grant, ranging from network access to read/write permission. For example, in Fig. 2, network permission must be granted to run an HTTP server. In the first case, any type of network connection to the web server is accepted, while in the second case, only localhost at the specific port 3000 can be accessed. No module can access a resource without being explicitly granted. There are 2 modes of granting privileges. The first method is to grant coarse-grained permission, as in the first command-line directives. The second method is to grant fine-grained permission, specifying which resource can be accessed. The difference between the two methods relies on the freedom that a user has over the deployed code. When possible, fine-grained should always be prioritized and used. On top of the permission systems, Deno introduced another drastic change to the JavaScript ecosystem. Deno has no default package manager, but instead relies on URLs to import libraries. The idea behind, is that getting rid of npm is helpful to reduce the attack surface, while granting more freedom to developers to include code from any source. In case of malicious code, the permission system should suffice to mitigate the potential vulnerabilities. An example of import from a URL is shown in Fig. 2, as the ”serve” module hosted by deno.land domain is used. This drastic change creates a decentralized ecosystem where multiple domains take over npm’s role.

Fig. 2. Implementation of an HTTP server in Deno

```
import { serve } from
"https://deno.land/std@0.114.0/http/server.ts";

console.log("Server running on port 3000");

serve(
  (_req) =>
    new Response("Hello from Deno!", {
      headers: { "Content-Type": "text/plain", }
    }), { addr: ":3000" }
);
```

The server can be launched via either of the following command line directives:

```
> deno --allow-net deno/index.ts

> deno --allow-net=0.0.0.0:3000
  deno/index.ts
```

C. *Bun*

Bun runtime is built on the JavaScriptCore engine, and it aims to optimize operations in order to achieve maximal speed. It is written in Zig, a low-level programming language that allows it to run closer to the operating system, and hence makes it generally faster than Node.js. Bun embeds npm as a package manager, but manages itself automatically missing dependencies, relieving developers from the burden of manually installing every single one. Moreover, Bun can also be used as a package manager only and can be imported into other projects. Bun startup time is 4 times faster than Node.js and twice as much as Deno [11]. It is worth mentioning also the energy consumption that can play a big role in the expenses for a big company. Bun is the runtime environment that requires the least amount of energy to function in comparison to both Node.js and Deno [47]. As done for the previous JavaScript runtimes, an implementation for an HTTP server in Bun is shown in Fig. 3.

D. *Other Runtimes*

Many other JavaScript runtime environments with specific roles exist. Hermes engine and runtime were released by Facebook and are optimized for fast start-up of React Native apps. It implements ahead-of-time compilation and modified bytecode. ChakraCore was the engine used in Microsoft Edge. Even though Microsoft stopped providing updates for it, ChakraCore continued as a community project and is now a JavaScript runtime for embedded systems. It is compatible with any C-compatible system and can be compiled on x64 processors as well as ARM. Jerryscript is a lightweight JavaScript engine for the Internet of Things that requires less than 64 KB of RAM to work.

E. *Javascript engine*

The most widely used JavaScript engines are Chrome's V8 and Mozilla's SpiderMonkey. They are both regarded as high-performance options that implement Just-In-Time compilation and WebAssembly to allow for better performance and security of critical operations. The use of JIT compilation could introduce security vulnerabilities in JavaScript interpreters due to its complexity. Various detection systems have been implemented to detect vulnerabilities during compilation, for example, through fuzzing [31], [26], [15], through side channels [18], or through semantic-aware code generation [33]. JIT spraying is also a typical issue that can be partially prevented [22]. Vulnerabilities at this lower level can be critical. As a C++ application, the V8 engine is subject to the same attacks as any C++ program, such as type confusion [3], heap exploitation [2], or out-of-bounds memory write [1]. In an effort towards memory safety, the V8 Sandbox has been introduced [7], separating the heap memory from the rest of the program memory to reduce the impact of memory corruption. Another proposed heap defense mechanism is enabling fine-grained memory access control for individual memory regions, to mitigate the escalation of attacks in the heap [50].

We suggest emerging JavaScript runtimes to thoroughly test their engine with the above-mentioned detection tools. They should also look into the possibility of implementing sandboxes for external code and adopt a permission system that can shield memory from heap attacks.

III. Package Distribution

Different runtimes have different ecosystems for distributing libraries. Node.js and Bun are built upon a centralized package, npm, where developers publish their libraries and from which they can be downloaded. On the contrary, Deno opted for a decentralized package system. Libraries can be imported with static links from any source. We analyze these two options.

A. *Node.js*

npm provides over 3.1 million software packages, tracks and manages their dependencies, and allows for agile code reuse. It is the main building block of Node.js and Bun, but since its release, npm has been the subject of many studies and often its security has been questioned [25], [43], [21], [10], [14], [30]. The supply chain can be easily corrupted, because it is enough that a single node out of the whole dependency tree is outdated or vulnerable to introduce attack vectors in the application [65], [12], [52], [63], [40], [39], [49]. The concept of a "trusted maintainer" might be used to decide which packages can be considered safe because reviewed by a trusted party [67]. Various npm dependency scanners have been developed over the years. Some of them rely on static analysis [51], [12], while others mitigate the impact by enforcing the least privilege principle on each package at run-time [27]. To prevent code injection during install time, application manifests have been tested to be effective, specifically by comparing auto-generated package-specific intentions (manifests) against user-controlled policies [64]. Another security issue is package confusion. It is a social engineering attack where a malicious party publishes

a package with a name similar to a famous one, for example, through typosquatting, and developers are lured into importing it [57], [17], [57]. To protect against typosquatting, registering domains with misspelled letters can help [55], but a big help against package confusion comes once again from automatic detection systems [48], [56], [5]. Moreover, many packages contain static URL imports of expired websites and dangling references that can be taken over by attackers to introduce malicious code in the application [32], [62].

B. *Deno*

Deno relies on a decentralized library system, where there is no registry to look up, but instead, static URLs are used to import packages. While shifting away from a centralized package manager can reduce risks like post-install hooks, URL-based importing has some security implications. First of all, libraries can be obtained from different sources that can have heterogeneous behaviours. Some of the following vulnerabilities are, in fact, only true for a subset of domains, but a tree is as weak as its weakest link. Deno does not enforce HTTPS, which leads to insecure imports and opens up the possibility of an adversary-in-the-middle scenario with code injection [13]. Expired domains and dangling URLs can lead to domain takeover [13]. Some domains allow owners of packages to delete them after publication. This gives attackers the possibility to register deleted domains and inject code in all applications that import the dangling package [13]. Moreover, unexpected downtime of domains can affect the operability of applications. Another issue arises with the versioning of already released packages. In some domains, it is possible to change the content or the dependencies of a package after its publication.

All of the above-mentioned problematic topics are also faced by the programming language Go. Similarly, they allow users to include code from any source by providing the respective URL. Because of its longevity and Google's support, Go came up with an effective solution that mitigates the disadvantages of adopting such an importing policy. They introduced a proxy server that caches modules in an immutable storage, a server that stores the checksum of the cached modules, and a server that advertises available modules [4]. The proxy server is opted in by default and enforces HTTPS usage, while providing defenses against domain takeover and manipulation of already published packages. Even though downtime of the proxy server can affect production code, the download of resources is 3 times faster during uptime.

In this paper, we recreated the availability study as performed by Alhamdan et al. [13]. We fetched for 1 week, every day, the same 21 domains as in the original work. The purpose is to gain hindsight on the domains' availability situation, 2 years after the original study. A domain is considered available if the response code to a GET request is 200, and the import of a hosted library is successful. Specifically, we fetched and included libraries in the time interval between July 4th 2025, and July 10th 2025. We observed that all the domains where we registered failed imports happened on sites that were also giving unavailable response codes. Therefore, when a site is deemed available, it means that both were successful.

As Table 1 shows, we found that three domains were not available anymore, each for different reasons. The first case is "cdn.pagic.org", which shows signs of a mismanaged domain, or at least it fell into disuse. They are still online, but never adopted HTTPS, and their landing page is an uncaught error message. A similar case is "cdn.dreg.dev". Accessing the domain returns an error code, but trying to import a library reveals the underlying problem. They have an internal DNS error that causes the address lookup to fail. The last unavailable domain is "x.nest.land". Out of the three, it was the largest library host for Deno and the one that had the biggest impact on existing applications. Any type of request to the site returned a 503 response code. It is unclear whether the website has been taken down by its administrator or if they are having temporary issues.

The libraries hosted by the studied unavailable domains make up for 0.4847% of the URLs in the data set of Alhamdan et al. [13]. While this number might not seem worrying, it is a large part of the ecosystem. As an example, we can imagine applications with only 10 imported libraries. In this scenario, on average, there would be 1 malfunctioning application every 22. And since applications normally include more than 10 modules, the threat could be even larger in the real world.

TABLE I
RESULTS OF THE DOMAINS AVAILABILITY STUDY

Domain tested	Available	Reason
deno.land	Yes	-
esm.sh	Yes	-
cdn.skypack.dev	Yes	-
cdn.jsdelivr.net	Yes	-
raw.githubusercontent.com	Yes	-
jspm.dev	Yes	-
x.nest.land	No	Service unavailable
unpkg.com	Yes	-
dev.jspm.io	Yes	-
denopkg.com	Yes	-
ghuc.cc	Yes	-
cdn.esm.sh	Yes	-
gist.githubusercontent.com	Yes	-
cdn.pika.dev	Yes	-
lib.deno.dev	Yes	-
cdn.pagic.org	No	Certificate expired
crux.land	Yes	-
cdn.shopstic.com	Yes	-
x.lcas.dev	Yes	-
ghc.deno.dev	Yes	-
cdn.dreg.dev	No	DNS error

IV. PERMISSION SYSTEM

Different runtimes implement different permission systems in their environments. The risk is that an attacker could exploit a vulnerability and subsequently be able to access the machine and the file system of the host running the runtime. We take a look now at how a scenario like this could happen and what the countermeasures are.

A. *Node.js*

During the installation or uninstallation of an npm module, Node.js fires, respectively, InstallEvent and UninstallEvent. If the module implements a hook for one of these events, the module will be able to modify how npm is run and execute additional commands. It was found that 43.6% of module downloads were run with root privileges [58], which ultimately allows the post-install hook to execute commands on the machine with root privileges too. Developers of Node.js addressed post-install hooks threats by implementing a permission system that restricts what system resources and what actions a Node.js process can access. The permission flags will be copied to eventual child processes, web workers, and web assembly spawned by the main process. Only two actions can be restricted against the file system: –allow-fs-read and –allow-fs-write. Moreover, the permission system is not enabled by default and has to be specified as a flag when running the project with –permission. This countermeasure is not enough because it relies on developers' awareness, and it does not cover most cases. Furthermore, Node.js trusts any code that it is asked to run. Table II will be discussed more in depth in the next subsection, but it can be seen already how damaging this Node's policy is. The vast majority of vulnerability samples in the benchmark were easily exploitable, including ACE and command injection. No defense is in place. A more efficient way to implement a permission system is to use a fine-grained read-write-execute permission system, similar to Linux can be applied [60]. Limiting the attack surface by permitting only allow-listed system calls has proven effective too [61]. On top of the permission system, mitigation of post-install scripts can happen in the runtime environment. It is possible to spawn different threads for the execution of different untrusted external pieces of code. Imported modules would run on different threads with limited capabilities [23]. Another very similar idea is to run imported modules in sandboxed compartments with limited permissions [59]. Finally, if a developer wants a ready-to-deploy solution to post-install scripts, Yarn and pnpm are alternative package managers that disallow scripts by default.

B. *Deno*

Deno was developed with security in mind. A program running with Deno has no access to sensitive APIs. To access or modify any external resource, such as the file system, network connectivity, or system calls, the developer has to grant additional permissions through command-line flags. The results of this security-oriented policy can be seen in Table II. The experiment has been conducted by running the security benchmark SecBench.js on the JavaScript environment of Node, and then on Deno. The goal is to find out how the two systems behave in the presence of vulnerabilities in the code that they are deploying. Considerations for Node have already been drawn. Deno's part of the experiment consisted of 2 phases. In the first phase, we focused on minimal permissions, paying attention to grant only the privileges requested and nothing more than that. The second phase of the experiment repeated the same measuring, but this time granted the highest privileges that Deno can give, through the –allow-all flag. Contrarily to Node.js, when running the test suite in the SecBench.js benchmark, most of the vulnerabilities are mitigated by Deno's environment. It is particularly significant when it comes to built-in defenses in the environment against prototype pollution and command injection. On the other hand, path traversal and regular expression denial of service (ReDoS) are mostly still exploitable. We believe the reason for the first is that even the lowest privileges requested were somewhat intrusive and enough to achieve path traversal. In the latter case, it was shown in previous research that resource exhaustion in Deno is still not addressed [13]. The second phase focused on the effect of granting all privileges to the test suite of SecBench.js. To our surprise, in most cases –allow-all decreased the vulnerable code base, despite the increased attack surface. The value that stands out the most is for path traversal, with not a single vulnerable test found. In this folder, all the tests deemed vulnerable with the lowest privilege were instead detected as information leaks in this setup. During evaluation, we found a severe usability issue in the way Deno enforces permissions. If a module tries to access a resource without having the relative privilege, the program will halt and prompt the developer to grant access at run time. Oftentimes, the test run had to be conducted again from the start because not enough specific permissions were specified in the launch command. This behaviour can become a problem in complex applications with multiple imports, because the developer has to specify for each module permission flags, spawning even hundreds of lines. In the implementation example we provided at Fig. 2, it is already clear how setting permissions explicitly makes Deno more verbose than its competitors, as well as complex. A developer is easily overwhelmed by all the possible security settings and often resorts to just granting all privileges. Moreover, Deno has a wide range of permission options. The permission system gives the possibility to specify both coarse and fine-grained permissions to modules. Again, the example command line directives at Fig. 2 serves to show the difference and complexity between them. Uninformed developers might tend to give broad, coarse-grained permissions, making the whole system completely insecure. Unfortunately, static import of libraries is outside the scope of the permission system. In case an attacker has –allow-read permission, they can achieve exfiltration of data even without having network permissions. Since URL imports can contain GET parameters, the attacker can import from a controlled endpoint while passing as a GET parameter the content of a file. Another security pitfall in Deno's permission system is shadow permissions. When granting permissions like read or write access to the file system, the developer is also granting additional permissions, which we call shadow permissions. Shadow permissions in combination with coarse-grained permissions can lead to unexpected and insecure results. For example, coarse-grained –allow-write allows the module to access and modify any file in the whole operating system, potentially rewriting application code,

TABLE II
SECBENCH.JS VULNERABILITY STUDY

Vulnerability	Node		Deno		Deno –allow-all	
	Vulnerable	Not Vulnerable	Vulnerable	Not Vulnerable	Vulnerable	Not Vulnerable
Prototype Pollution	191	1	12	180	11	181
Regular expression Denial of Service (ReDoS)	96	2	59	39	59	39
Command Injection	94	7	29	72	2	99
Path Traversal	164	6	139	31	0	170
Arbitrary Code Injection	39	1	11	29	25	15

storage memory, and environment variables. Similarly, coarse-grained –allow-read gives the module the capacity of reading environment variables and memory of the underlying operating system, capabilities normally granted using –allow-end and –allow-sys. At last, even in case an informed developer decides to grant fine-grained permissions, security issues may arise. In particular, Deno gives all the files reached through a symbolic link the same permissions as the symbolic link itself. This means that if –allow-write is granted to an attacker-controlled folder, for example the .cache folder of the application, an attacker could place inside the folder a symbolic link to a system resource and be able to modify it. Unclear instructions in the installation documentation of libraries exacerbate the complexity of the permission system. Many libraries do not provide any instructions on the minimal permissions needed to run their code, and some libraries suggest applying large, coarse-grained permissions as a default. Because of the severity of the issues underlined in this chapter, some improvements to the Deno permission system have been proposed. In the first place, support for symbolic links should be fixed. Then, Deno should remove read/write access to sensitive operating system paths and add support for permission policy files [13]. Adding a restrictive sandbox around Deno's environment has been proven effective. It leverages Landlock, eBPF, and Seccomp to mediate the interaction between Deno with the file system, the IPC, and the network resources [9]. Another interesting research explores the adoption of a specific sandbox with minimal permissions for running untrusted third-party code [8].

V. PROTOTYPE POLLUTION

Prototype pollution is a vulnerability that affects prototype-based programming languages, including JavaScript. Its severity depends on case to case, ranging from simple client functionality disruption to remote code execution on the machine running the JavaScript program. In combination with previously discussed permission systems flaws in JavaScript runtime environments, it can lead to critical consequences. A prototype pollution vulnerability happens when unsanitized input is passed to a JavaScript function that accesses global objects and modifies the properties of the prototype (or constructor) of an object. The polluted object is called a sink. For the prototype pollution to affect a system is necessary that an otherwise benign piece of code use the sink in an insecure way. This piece of code is called a gadget. An example of a gadget could be a JavaScript function in a website that reads a file of the underlying system with the name of an object's property. If an attacker can exploit a prototype pollution in the website, they would be able to read from any file on the remote machine.

In this section, we examine the state-of-the-art detection systems for prototype pollution. Exploitation of the vulnerability is out of the scope of this paper.

A. *Static Analysis*

Static analysis of prototype pollution vulnerability can take various forms. For example, Joern is an open-source static analysis tool that scans code using queries formulated in a Scala-based query language. Semantic analysis uses a code property graph representation, obtained by merging the program abstract syntax tree, the control flow graph, and the program dependence graph. The code property graph's nodes are stored in the database and represent the different constructs of the program. Taint analysis is then performed using user-defined sinks and sources [28]. DAPP is another automatic prototype pollution detection tool that targets all imported npm modules of an application. It performs static analysis based on the abstract syntax tree and the control flow graph, and has been proven to be effective in real-world scenarios [36]. ObjLupAnsys is a flow-, context-, and branch-sensitive static taint analysis tool. It performs targeted object lookup analysis, which expands source and sink objects into clusters until a built-in function can be redefined. In its implementation study, ObjLupAnsys was able to find 61 zero-day vulnerabilities [41]. Other studies focused not only on the detection of prototype pollution but also on its implications. Building on top of the detection tool CodeQL, the Silent Spring tool found several universal gadgets in popular Node.js APIs. The gadgets have been then exploited to a chain of vulnerabilities leading to arbitrary code execution and remote code execution [53]. An innovative approach consists of adopting symbolic execution in combination with dynamic analysis to chain gadgets and sinks. The researchers implemented this concolic execution with undefined properties as symbols and were able to outperform Silent Spring in gadget detection, as well as false positive and false negative rates [44].

B. *Dynamic Analysis*

Despite its complexity, dynamic analysis can give greater results if contextualized. As an example, Dasty is a semi-automatic pipeline that helps developers identify gadgets in

their application's software supply chain. Since Dasty is created as a quality-of-life improvement for developers, it also embeds other useful features to support them in keeping applications safe. In particular, Dasty targets server-side Node.js applications and relies on improved dynamic taint analysis implemented with dynamic abstract tree syntax instrumentation. It also provides visual support for the visualization of code flaws with an IDE, while also facilitating building proof-of-concept exploits [54]. Dynamic analysis is also suitable for deeper workflows. Thanks to this property, it was possible to deploy a framework, ProbeTheProto, able to both detect prototype pollution and chain prototype pollution exploitation with other vulnerabilities such as XSS, cookie exfiltration, and URL manipulation. ProbeTheProto consists of two parts. The first part is a dynamic taint analysis module that tracks the taint flows connecting property lookups and assignments. The second part is an input/exploit generation that guides the taint flows into the final sinks, useful for exploitation of further vulnerabilities. ProbeTheProto has been used to study the security of one million websites, and it has found 2,917 zero-days [35]. Lastly, GHunter is a pipeline implemented that modifies the V8 engine that uses dynamic taint analysis to automatically identify gadget candidates. It works in 4 steps. The first step is to collect source properties that influence the behaviour of a runtime API. They must be undefined so that they are looked up the prototype chain. The second step consists of identifying the reachability of polluted values into dangerous sinks. If the sink is reached, it means that the prototype pollution was successful. The third step is to automatically recognize unexpected program termination due to the polluted values. The final step is then manual validation, supported by GHunter helper features. GHunter was deployed against Node.js and Deno, two runtime environments that rely on the V8 engine, and found several new prototype pollution universal gadgets [24].

C. *Deno*

We want to focus now on the Deno runtime environment for its efforts in mitigating prototype pollution. Because of its goal to reduce the attack surface and have built-in defense mechanisms against known vulnerabilities, Deno adopted different rules compared to Node.js. First of all, an Object's prototype can be accessed in many different ways. Some of these are now deprecated, but still in use to ensure backward compatibility for the legacy code base. An example is using `Object.prototype.__proto__` to change a property of the prototype and consequently of all its inheriting objects. Deno decided to disable this method as it is a common pollution path. However, many other ways to access the prototype at runtime are still available, and can still cause harm. Another defense mechanism present in Deno is primordial objects. Primordials are a frozen set of all intrinsic objects in the runtime. As they cannot be altered, they should be preferred in Deno's internal environment to avoid the risk of prototype pollution. An example usage is shown in Fig. 3, where a frozen array object is used.

Fig. 3. Primordial usage example

```
const arr = getSomeArrayOfNumbers();

// vulnerable implementation
const evens = arr.filter((val)=> val % 2 === 0);

// safe implementation
const evens = primordials.ArrayPrototypeFilter(
                    arr, (val)=> val % 2 === 0);
```

VI. Evaluation

The files and logs that we used to answer the research questions are made available at our Github [6].

RQ1: Deno Permissions System. Deno's permission system allows for fine-grained and coarse-grained allow-list permissions. It is even possible to deny specific resource access. Everything must be set up using command-line flags at launch time. In case any are missing but are required for the system to work, the program halts and waits until a decision is made on the shell. We believe overall, Deno's permission system is a step towards the correct direction. In the worst case scenario, it will trust code in the same way Node.js does, while in the best case will improve the security of the system. The drawback of such a system is the complexity of managing it, and the lack of tools to aid developers in using minimal permissions. Moreover, usability is weakened by shadow permissions and counterintuitive behaviour, such as –allow-all flag, as our study shows.

RQ2: Deno Attack Surface. Our study reveals that Deno is indeed successful in mitigating many of the vulnerabilities that are instead effective in the Node.js environment. The developers reduced the attack surface by using a memory-safe programming language as Rust, adding an in-depth permission system, and disallowing dangerous and deprecated instructions. At the same time, the claim that all code is secure by default if run on Deno is misleading. Not all vulnerability paths are excluded, and not all vulnerabilities are addressed. Resource exhaustion, which can lead to DoS, was not covered. Our study demonstrates how vulnerabilities are only partially mitigated.

RQ3: Deno Supply Chain. Being the first JavaScript runtime to adopt a fully decentralized supply chain, most the developers are not aware of the risk it represents. Importing code from obscure, possibly malicious domains is a real threat that can lead to data exfiltration, or worse, in case broader permissions are granted. With the release of Deno 2.0, some problems have been addressed. For example, expired domains and dangling imports were resolved by buying the available domains. This is the case for "x.lcas.dev". The decentralized supply chain offers more freedom and power in the hands of users. However, domain takeover from expired domains is a cyclic issue. Based on our availability study, we highlight the problems that legacy code might encounter

when some previously maintained domains fall into disuse. HTTPS usage is not enforced and can lead to adversary-in-the-middle attacks. Moreover, Deno now caches all remote modules for an undefined amount of time. The reason is to mitigate modification of packages after release, as the local version will be cached. However, the root problem is still present. Finally, downtime of popular domains still affects the functioning of programs deployed in Deno.

RQ4: Prototype Pollution in Deno. Prototype pollution has been carefully considered by Deno's developers. To mitigate it, they disallowed some known vulnerable built-in methods that act as gadgets. The possibility of using primordial objects is also a step in the right direction. However, the work by Cornelissen et al. [24] highlights that the standard library of Deno is still implemented without prototype pollution in mind. They showed that Deno presents 67 universal gadgets. Namely, prototype pollution gadgets present in the runtime environment and usable by an attacker freely, as long as the respective module is imported. Despite the efforts in mitigating prototype pollution, complete mitigation cannot be achieved until Deno APIs fix their implementation to remove the use of unsafe, not safety-checked objects.

VII. Future Work

We discuss now the changes that would improve the security of the topics we reviewed in this paper.

A. *Node.js*

Node.js was built following weak security protocols. We suggest increasing the number of permission choices that the developer can choose from and enabling the already available ones by default. Another feature in this direction is to add compartmentalization for imported modules, for example, through sandboxes or threads with minimal permissions. Since post-install hooks are often exploited to execute unsupervised code on the victim machine, we suggest disabling this feature unless the developer explicitly enables it for a library. Many attacks target the npm ecosystem. Improving the security of package distribution and import should be a priority. A range of scanners that detect malicious npm libraries is already available, and Node.js should adopt one of them. Another suggestion to Node.js developers is to improve the npm audit command. The command returns a description of the dependencies in the project and a report of known vulnerabilities. npm audit could embed one of the already available automatic detection and exploit generation tools to provide a real insight into the security of the project [45], [20], [42], [19]. Another way to improve the npm audit command is to make it run the project while importing SecBench.js. SecBench.js is a framework that comprises 600 vulnerabilities and their relative exploits. If the tested project is affected by a vulnerability, SecBench.js will be able to exploit it, making the developer aware of the security issues in the code [16]. npm prune is another command that needs improvements. At the moment of writing, it removes modules not listed in the package.json file. We argue that it could also remove imported code and dependencies left unused. It would reduce the attack surface and the project size, as shown in previous research [38].

B. *Deno*

The issues about Deno that we highlighted in this paper can be addressed with improvements to both the package distribution ecosystem and the runtime environment itself. First of all, the developers should prevent a running program from accessing sensitive resources of the underlying file system. This way, shadow permission and misconfigured coarse-grained permission will not affect the machine running the application. On top of this, support for symbolic links should be fixed. To mitigate import attacks while keeping the decentralized approach, Deno should add support for fine-grained import permissions and possibly restrict imports to a list of trusted domains. Moreover, Deno should adopt compartmentalization to isolate code coming from different domains. A challenge for developers is the lack of tooling, exacerbated by the complexity of the runtime. Visual support and IDE extensions should be developed to help update dependencies, to infer API permissions, and to write minimal permissions for each module.

C. *Prototype pollution*

Most of the research revolving around prototype pollution focuses on the client-side vulnerability. Few papers focus on JavaScript runtime environment detection. Moreover, the use of concolic execution to detect prototype pollution has never been applied to a back-end environment. Another approach could be centered around the JavaScript programming language. It was possible to significantly reduce the number of client-side vulnerabilities by implementing a mandatory fine-grained, object-oriented permission system for browsers [66]. A similar solution can be studied for JavaScript runtimes outside of the browser.

VIII. Conclusion

In this paper, we discussed JavaScript runtime environments, with a special focus on Deno. We covered the most critical aspects of a JavaScript runtime that could introduce vulnerabilities if overlooked, and we evaluated suggested improvements and defenses. First, JavaScript engines and interpreters are examined. Then, package distribution is investigated. Node.js and Deno have two distinct approaches, introducing different types of vulnerabilities. The permission systems of Node.js and Deno are evaluated, with a specific focus on the security implications for the underlying system running the JavaScript runtime. Lastly, the prototype pollution vulnerability is discussed. We investigated the state-of-the-art tools for prototype pollution detection, which takes the form of static or dynamic analysis. We notice how both can produce considerable results, but while static analysis focuses on full attack surface coverage, dynamic analysis tends to target deeper workflows and vulnerability chaining. For future research, we suggest studying the counterintuitive behavior of Deno –allow-all flag and its implications in the runtime environment security.

References

[1] Cve-2024-5830. https://nvd.nist.gov/vuln/detail/cve-2024-5830, last accessed on 23-05-2025.

[2] Cve-2024-7965. https://nvd.nist.gov/vuln/detail/cve-2024-7965, last accessed on 23-05-2025.

[3] Cve-2024-8904. https://nvd.nist.gov/vuln/detail/CVE-2024-8904, last accessed on 23-05-2025.

[4] Go module mirror, index, and checksum database. https://proxy.golang.org/, last accessed on 25-07-2025.

[5] Socket.dev. https://socket.dev/features/github, last accessed on 24-05-2025.

[6] State of security in javascript runtimes, github. https://github.com/davidebombelli/State-of-Security-in-JavaScript-Runtimes.

[7] The v8 sandbox. https://v8.dev/blog/sandbox, last accessed on 23-05-2025.

[8] Marco Abbadini, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, and Stefano Paraboschi. Cage4deno: A fine-grained sandbox for deno subprocesses. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, pages 149–162, 2023.

[9] Marco Abbadini, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, and Stefano Paraboschi. Natisand: Native code sandboxing for javascript runtimes. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '23, page 639–653, New York, NY, USA, 2023. Association for Computing Machinery.

[10] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad 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*, pages 385–395, 2017.

[11] Md Feroj Ahmod. *Javascript runtime performance analysis: Node and Bun*. PhD thesis, Master's thesis, Tampere University, 2023.

[12] Mahmoud Alfadel, Diego Elias Costa, Emad Shihab, and Bram Adams. On the discoverability of npm vulnerabilities in node. js projects. *ACM Transactions on Software Engineering and Methodology*, 32(4):1–27, 2023.

[13] Abdullah Alhamdan and Cristian-Alexandru Staicu. Welcome to jurassic park: A comprehensive study of security risks in deno and its ecosystem. 2 2025.

[14] Ellen Arteca and Alexi Turcotte. npm-filter: Automating the mining of dynamic information from npm packages. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 304–308, 2022.

[15] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. Jit-picking: Differential fuzzing of javascript engines. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 351–364, 2022.

[16] Masudul Hasan Masud Bhuiyan, Adithya Srinivas Parthasarathy, Nikos Vasilakis, Michael Pradel, and Cristian-Alexandru Staicu. Secbench.js: An executable security benchmark suite for server-side javascript. In *International Conference on Software Engineering (ICSE)*, 2023.

[17] Alex Birsan. Dependency confusion: How i hacked into apple, microsoft and dozens of other companies, 2021. https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610, last accessed on 24-05-2025.

[18] Tegan Brennan, Nicolás Rosner, and Tevfik Bultan. Jit leaks: Inducing timing side channels through just-in-time compilation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1207–1222, 2020.

[19] Tiago Brito, Mafalda Ferreira, Miguel Monteiro, Pedro Lopes, Miguel Barros, José Fragoso Santos, and Nuno Santos. Study of javascript static analysis tools for vulnerability detection in node. js packages. *IEEE Transactions on Reliability*, 72(4):1324–1339, 2023.

[20] Darion Cassel, Nuno Sabino, Min-Chien Hsu, Ruben Martins, and Limin Jia. Nodemedic-fine: Automatic detection and exploit synthesis for node. js vulnerabilities. In *Proceedings of the 2025 Network and Distributed System Security Symposium (NDSS'25). doi*, volume 10, 2025.

[21] Kyriakos C Chatzidimitriou, Michail D Papamichail, Themistoklis Diamantopoulos, Michail Tsapanos, and Andreas L Symeonidis. Npm-miner: An infrastructure for measuring the quality of the npm registry. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 42–45, 2018.

[22] Ping Chen, Yi Fang, Bing Mao, and Li Xie. Jitdefender: A defense against jit spraying attacks. In *Future Challenges in Security and Privacy for Academia and Industry: 26th IFIP TC 11 International Information Security Conference, SEC 2011, Lucerne, Switzerland, June 7-9, 2011. Proceedings 26*, pages 142–153. Springer, 2011.

[23] George Christou, Grigoris Ntousakis, Eric Lahtinen, Sotiris Ioannidis, Vasileios P Kemerlis, and Nikos Vasilakis. Binwrap: Hybrid protection against native node. js add-ons. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, pages 429–442, 2023.

[24] Eric Cornelissen, Mikhail Shcherbakov, and Musard Balliu. GHunter: Universal prototype pollution gadgets in JavaScript runtimes. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 3693–3710, Philadelphia, PA, August 2024. USENIX Association.

[25] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th international conference on mining software repositories*, pages 181–191, 2018.

[26] Jueon Eom, Seyeon Jeong, and Taekyoung Kwon. Fuzzing javascript interpreters with coverage-guided reinforcement learning for llm-based mutation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1656–1668, 2024.

[27] Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. Containing malicious package updates in npm with a lightweight permission system. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1334–1346. IEEE, 2021.

[28] Tobias Fröberg. Detection of prototype pollution using joern: Joern's detection capability compared to codeql's, 2023.

[29] Evgeny Gavrin, Sung-Jae Lee, Ruben Ayrapetyan, and Andrey Shitov. Ultra lightweight javascript engine for internet of things. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, pages 19–20, 2015.

[30] Pronnoy Goswami, Saksham Gupta, Zhiyuan Li, Na Meng, and Daphne Yao. Investigating the reproducibility of npm packages. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 677–681. IEEE, 2020.

[31] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. Fuzzilli: Fuzzing for javascript jit compiler vulnerabilities. In *NDSS*, 2023.

[32] Yacong Gu, Lingyun Ying, Yingyuan Pu, Xiao Hu, Huajun Chai, Ruimin Wang, Xing Gao, and Haixin Duan. Investigating package related security threats in software registries. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1578–1595. IEEE, 2023.

[33] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In *NDSS*, 2019.

[34] Hongfei Hu, Yimei Xu, Yongheng Liu, and Huijing Wang. Research and design of graphic display technology for embedded devices based on ultra-lightweight javascript engine. In *2024 5th International Conference on Computer Engineering and Application (ICCEA)*, pages 1083–1086. IEEE, 2024.

[35] Zifeng Kang, Song Li, and Yinzhi Cao. Probe the proto: Measuring client-side prototype pollution vulnerabilities of one million real-world websites. In *NDSS*, 2022.

[36] Hee Yeon Kim, Ji Hoon Kim, Ho Kyun Oh, Beom Jin Lee, Si Woo Mun, Jeong Hoon Shin, and Kyounggon Kim. Dapp: automatic detection and analysis of prototype pollution vulnerability in node. js modules. *International Journal of Information Security*, 21(1):1–23, 2022.

[37] I Kniazev and A Fitiskin. Choosing the right javascript runtime: an in-depth comparison of node. js and bun. *ECONOMIC SCIENCES*, 108:72, 2023.

[38] Igibek Koishybayev and Alexandros Kapravelos. Mininode: Reducing the attack surface of node.js applications. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 121–134, San Sebastian, October 2020. USENIX Association.

[39] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. Sok: Taxonomy of attacks on open-source software supply chains. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1509–1526. IEEE, 2023.

[40] Piergiorgio Ladisa, Merve Sahin, Serena Elisa Ponta, Marco Rosa, Matias Martinez, and Olivier Barais. The hitchhiker's guide to malicious third-party dependencies. In *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, pages 65–74, 2023.

[41] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. Detecting node. js prototype pollution vulnerabilities via object lookup analysis. In *Proceedings of the 29th ACM Joint Meeting on European Software*

Engineering Conference and Symposium on the Foundations of Software Engineering, pages 268–279, 2021.

[42] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. Mining node. js vulnerabilities via object dependence graph and query. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 143–160, 2022.

[43] Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. In *Proceedings of the 44th International Conference on Software Engineering*, pages 672–684, 2022.

[44] Zhengyu Liu, Kecheng An, and Yinzhi Cao. Undefined-oriented programming: Detecting and chaining prototype pollution gadgets in node. js template engines for malicious consequences. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4015–4033. IEEE, 2024.

[45] FILIPE MARQUES, MAFALDA FERREIRA, ANDRÉ NASCIMENTO, MIGUEL E COIMBRA, NUNO SANTOS, LIMIN JIA, and JOSÉ FRAGOSO SANTOS. Automated exploit generation for node. js packages. 2025.

[46] mend.io. The most secure programming languages. https://www.mend.io/most-secure-programming-languages, last accessed on 17-05-2025.

[47] Juan J Merelo-Guervós, M García-Valdez, and Pedro A Castillo. An analysis of energy consumption of javascript interpreters with evolutionary algorithm workloads. In *Proceedings of the 18th International Conference on Software Technologies, ICSOFT*, pages 175–184, 2023.

[48] Shradha Neupane, Grant Holmes, Elizabeth Wyss, Drew Davidson, and Lorenzo De Carli. Beyond typosquatting: an in-depth look at package confusion. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3439–3456, 2023.

[49] Marc Ohm and Charlene Stuke. Sok: Practical detection of software supply chain attacks. In *Proceedings of the 18th International Conference on Availability, Reliability and Security*, pages 1–11, 2023.

[50] Taemin Park, Karel Dhondt, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. Nojitsu: Locking down javascript engines. In *Proceedings 2020 Network and Distributed System Security Symposium*. Internet Society, 2020.

[51] Simone Scalco, Ranindya Paramitha, Duc-Ly Vu, and Fabio Massacci. On the feasibility of detecting injections in malicious npm packages. In *Proceedings of the 17th international conference on availability, reliability and security*, pages 1–8, 2022.

[52] Adriana Sejfia and Max Schäfer. Practical automated detection of malicious npm packages. In *Proceedings of the 44th international conference on software engineering*, pages 1681–1692, 2022.

[53] Mikhail Shcherbakov, Musard Balliu, and Cristian-Alexandru Staicu. Silent spring: Prototype pollution leads to remote code execution in node.js. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5521–5538, Anaheim, CA, August 2023. USENIX Association.

[54] Mikhail Shcherbakov, Paul Moosbrugger, and Musard Balliu. Unveiling the invisible: Detection and evaluation of prototype pollution gadgets with dynamic taint analysis. In *Proceedings of the ACM Web Conference 2024*, pages 1800–1811, 2024.

[55] Matthew Taylor. Defending against typosquatting attacks in programming language-based package repositories. Master's thesis, University of Kansas, 2020.

[56] Matthew Taylor, Ruturaj Vaidya, Drew Davidson, Lorenzo De Carli, and Vaibhav Rastogi. Defending against package typosquatting. In *Network and System Security: 14th International Conference, NSS 2020, Melbourne, VIC, Australia, November 25–27, 2020, Proceedings 14*, pages 112–131. Springer, 2020.

[57] Minh Tien Truong. *Typosquatting Attacks and Mitigations*. PhD thesis, University of Applied Sciences, 2023.

[58] Nikolai Philipp Tschacher. *Typosquatting in programming language package managers*. PhD thesis, Universität Hamburg, Fachbereich Informatik, 2016.

[59] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M Smith. Breakapp: Automated, flexible application compartmentalization. In *NDSS*, 2018.

[60] Nikos Vasilakis, Cristian-Alexandru Staicu, Grigoris Ntousakis, Konstantinos Kallas, Ben Karel, André DeHon, and Michael 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*, pages 1821–1838, 2021.

[61] Wenya Wang, Xingwei Lin, Jingyi Wang, Wang Gao, Dawu Gu, Wei Lv, and Jiashui Wang. Hodor: Shrinking attack surface on node. js via system call limitation. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 2800–2814, 2023.

[62] Elizabeth Wyss, Drew Davidson, and Lorenzo De Carli. What's in a url? an analysis of hardcoded urls in npm packages. In *Proceedings of the 2024 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, pages 26–32, 2023.

[63] Elizabeth Wyss, Lorenzo De Carli, and Drew Davidson. What the fork? finding hidden code clones in npm. In *Proceedings of the 44th international conference on software engineering*, pages 2415–2426, 2022.

[64] Elizabeth Wyss, Alexander Wittman, Drew Davidson, and Lorenzo De Carli. Wolf at the door: Preventing install-time attacks in npm with latch. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, pages 1139–1153, 2022.

[65] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, and Laurie Williams. What are weak links in the npm supply chain? In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pages 331–340, 2022.

[66] Rui Zhao. Beast in the cage: A fine-grained and object-oriented permission system to confine javascript operations on the web. In *Proceedings of the ACM on Web Conference 2025*, pages 3171–3182, 2025.

[67] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX security symposium (USENIX Security 19)*, pages 995–1010, 2019.