GHUNTER: Universal Prototype Pollution Gadgets in JavaScript Runtimes

Eric Cornelissen
KTH Royal Institute of Technology

Mikhail Shcherbakov
KTH Royal Institute of Technology

Musard Balliu
KTH Royal Institute of Technology

Abstract

Prototype pollution is a recent vulnerability that affects JavaScript code, leading to high impact attacks such as arbitrary code execution and privilege escalation. The vulnerability is rooted in JavaScript's prototype-based inheritance, enabling attackers to inject arbitrary properties into an object's prototype at runtime. The impact of prototype pollution depends on the existence of otherwise benign pieces of code (gadgets), which inadvertently read from these attacker-controlled properties to execute security-sensitive operations. While prior works primarily study gadgets in third-party libraries and client-side applications, gadgets in JavaScript runtime environments are arguably more impactful as they affect any application that executes on these runtimes.

In this paper we design, implement, and evaluate a pipeline, GHUNTER, to systematically detect gadgets in V8-based JavaScript runtimes with prime focus on Node.js and Deno. GHUNTER supports a lightweight dynamic taint analysis to automatically identify gadget candidates which we validate manually to derive proof-of-concept exploits. We implement GHUNTER by modifying the V8 engine and the targeted runtimes along with features for facilitating manual validation. Driven by the comprehensive test suites of Node.js and Deno, we use GHUNTER in a systematic study of gadgets in these runtimes. We identified a total of 56 new gadgets in Node.js and 67 gadgets in Deno, pertaining to vulnerabilities such as arbitrary code execution (19), privilege escalation (31), path traversal (13), and more. Moreover, we systematize, for the first time, existing mitigations for prototype pollution and gadgets in terms of development guidelines. We collect a list of vulnerable applications and revisit the fixes through the lens of our guidelines. Through this exercise, we also identified one high-severity CVE leading to remote code execution, which was due to incorrectly fixing a gadget.

1 Introduction

JavaScript's widespread adoption as a go-to programming language for full-stack development speaks to its popularity, but it also exposes the applications to heightened security risks. Researchers and practitioners are well-aware of these issues, as witnessed by a multitude of prior studies [17, 46, 48, 51]. JavaScript runtime environments, such as Node.js [4] and Deno [3], which lie at the heart of server-side JavaScript applications, become appealing targets for attackers [9, 11, 17, 29, 43, 45, 49]. Vulnerabilities in the runtime environments can compromise the security of applications running atop. In this paper, we set out to study the security implications of a recent vulnerability, prototype pollution, in JavaScript runtime environments.

Prototype pollution is a vulnerability affecting the JavaScript language [10]. JavaScript's prototype-based inheritance allows an object to inherit properties from its ancestors via the prototype chain. When accessing a property not present on the object, the prototype chain will be queried for that property instead. Unless explicitly changed, this chain connects all objects to a common root prototype. Pollution can occur when an attacker-controlled value is used to navigate an object's structure. Since each object has a runtime accessible reference to its prototype, the attacker may be able to pick that reference and add a new property. By doing this, the attacker can cause a change in behavior in another part of the application.

The security implications of prototype pollution depend on the presence of otherwise benign pieces of code (gadgets) that inadvertently read attacker-controlled properties from the root prototype to execute sensitive operations, e.g., arbitrary code. Gadgets in JavaScript runtime environments are particularly dangerous because they are shared by all applications, thus increasing the attack surface.

The vast majority of prior works focus on the detection of prototype pollution by static analysis [26, 29, 30, 43, 49], while the existence of gadgets remains largely unexplored [24, 31, 43, 44]. This work is inspired by the recent pioneering of work of Shcherbakov et al. [43], which uses static taint analysis for three Node.js APIs to find (combinations of) three gadgets, dubbed *universal gadgets*, leading to arbitrary code execution. Our thesis is that dynamic analysis should be preferable for

identifying universal gadgets for these reasons: (a) the sources of the analysis pertain to accesses of properties from the prototype, which is hard to identify statically; (b) the highly-dynamic nature of JavaScript poses significant challenges for static analysis, resulting in low precision and recall, and high manual effort [43]; (c) realistic gadgets should trigger in common use cases of API usages, which is best captured by the comprehensive test suite of runtime environments.

To address these challenges, we design, implement, and evaluate a semi-automated pipeline, GHUNTER, to comprehensively and systematically detect universal gadgets in V8-based JavaScript runtimes, Node.js and Deno. Deno is a particularly interesting target because it is proposed as a security-first runtime to counter the shortcomings of Node.js. Specifically, GHUNTER customizes Deno, Node.js, and the V8 engine to implement a lightweight dynamic taint analysis for automatically identifying gadget candidates, which we validate manually to derive proof-of-concept exploits. Driven by the test suite of a runtime environment, GHUNTER detects property accesses from an object's prototype, it injects a taint value, and monitors the execution to identify the effects of the taint value on security-sensitive sinks and unexpected terminations. Moreover, GHUNTER supports processing and representation of gadget candidates in SARIF format [36] for visualization to facilitate the manual analysis.

We use GHUNTER in a comprehensive study of Node.js and Deno to identify universal gadgets pertaining to a range of vulnerabilities, including arbitrary code execution, server-side request forgery, privilege escalation, cryptographic downgrade, and more. After processing, GHUNTER automatically identifies 301 and 418 gadget candidates in Node.js and Deno, respectively. We manually verified the gadget candidates to find 56 universal gadgets in Node.js and 67 universal gadgets in Deno for a total of 28 person-hours. We further compare GHUNTER with Silent Spring [43], showing that it provides increased precision and recall, while reporting less gadget candidates for manual analysis. To support further research on the topic, we make available publicly both GHUNTER [14] and the gadgets [20].

We have responsibly disclosed our findings to the Node.js and Deno development teams. Both acknowledged our report but neither considers them within their current thread model. Node.js suggested a public discussion with their developers' community on the dangers of gadgets.

In light of these results, we systematize, for the first time, existing mitigations for prototype pollution and gadgets in terms of development guidelines. We then collect a list of applications with end-to-end exploits pertaining to prototype pollution, and revisit the fixes through the lens of our guidelines. Through this exercise, we also identify existing issues, including one high-severity CVE-2023-31414 leading to remote code execution, which was due to incorrectly fixing a gadget.

Our contributions can be summarized as follows:

```
1  const users = { };
2  router.post("/:uid", (req, res) => {
3    users[req.uid][req.key] = req.value;
4    exec("echo 'A value was stored at' $(date)");
5    res.status(200).send();
6  });
7  function exec(cmd, opts) {
8    opts = opts || {};
9    const shell = opts.shell || "/bin/sh";
10   op_spawn(`${shell} -c '${sanitize(cmd)}'`);
11 }
```

Listing 1: Example of prototype pollution and gadget.

- We design and implement a semi-automated pipeline, GHUNTER, to systematically detect universal gadgets in JavaScript runtimes (Section 4).
- We conduct a comprehensive analysis of Node.js and Deno to find 123 universal gadgets subject to a range of vulnerabilities (Section 5).
- We systematize existing mitigations against prototype pollution and gadgets, and outline directions for future work, including an in-depth case study leading to RCE (Section 6).

2 Technical Background

In this section, we overview the life cycle of exploits pertaining to prototype pollution vulnerabilities, and discuss the JavaScript runtime of interest and the threat model.

2.1 Prototype Pollution and Gadgets

Prototype pollution is a vulnerability that occurs in prototype-based languages like JavaScript [10]. An attacker manipulates a program's prototype-based inheritance, leading to runtime modification of objects and potentially causing otherwise benign code sequences, called gadgets, to execute dangerous operations. End-to-end exploitation of gadgets based in prototype pollution requires two steps. The prototype must be polluted first, for example when processing untrusted user data incorrectly, and then the gadget must be triggered.

To illustrate the vulnerability, Listing 1 shows an artificial server application which provides an in-memory key-value store for its users, logging every request to standard output. It is vulnerable to prototype pollution and uses function `exec` as a gadget. `exec` (line 7-11) is an otherwise benign runtime-provided function to execute a command. It accepts the command to execute as a string and an optional object `opts` to configure the shell in which to execute the command.

A request at `vuln.com/uid?key=value` causes the server to invoke the handler on line 2-6. It extracts the user ID and the key-value pair from the URL and stores it in memory (line 3). It then logs the time of the request (line 4) and responds with a `200` status code (line 5).

An attacker can use this handler to perform prototype pollution. The malicious request `vuln.com/__proto__?shell=node -e '...'` will add the property `shell` with the value `"node -e '...';"` to the root object prototype on line 3. This happens because the request instantiates the statement on line 3 as `users["__proto__"]["shell"] = "node -e '...';"`. In particular, `users["__proto__"]` gives a reference to `Object.prototype` which is then extended with the property `shell`.

The attacker can capitalize on the pollution of the `shell` property to turn the benign call to `exec` into a remote code execution gadget. In particular, because the application provides no options on line 4, line 8 assigns to `opts` an empty JavaScript object. When evaluating the expression `opts.shell` on line 9, the `shell` property, missing from `opts`, will be looked up in the prototype chain where it exists because of the pollution. Thus, `opts.shell` evaluates to `"node -e='...';"` and is used instead of the default `"/bin/sh"` to evaluate arbitrary JavaScript code.

2.2 JavaScript Runtimes: Node.js and Deno

In this work, we study universal gadgets in JavaScript runtime environments. Two such runtime environments are Node.js and Deno. Both are open source software projects built on top of the V8 JavaScript engine from Chromium. Node.js is a popular JavaScript runtime [4] written in C++, commonly used for server application development. Deno was created in response to Node.js with a focus on security [3]. It is written in Rust and uses TypeScript. The native (C++/Rust) parts of these runtimes are what provides access to system resources and common functionality such as buffers and cryptography libraries. In this work we focus on these runtimes because of their popularity and shared JavaScript engine.

Deno's focus on security is interesting for our work because it adds guardrails for both pollution and gadgets. On the pollution side, Deno removed the `__proto__` property, rendering the attack described on Listing 1 infeasible. However, prototype pollution is still possible through, e.g., object merge functions, a common source of prototype pollution. On the gadget side, Deno has a permission system to reduce access to system resources and by extension the impact of gadgets. However, we observe that the presence of a gadget implies some access to the corresponding resource must have been granted to the application, thus allowing exploits nonetheless.

2.3 Threat Model

Our threat model focuses on server-side JavaScript/TypeScript applications running on either Node.js or Deno. We assume the application is vulnerable to prototype pollution, either directly or through third-party code. Our aim is to find exploitable universal gadgets present in the JavaScript runtime for the purpose of one of (directly or indirectly):

- Arbitrary Code/Command Execution (ACE). Gadgets that allow an attacker to execute arbitrary JavaScript code or start an arbitrary command.
- Server Side Request Forgery (SSRF). Gadgets that allow an attacker to make arbitrary network requests.
- Privilege Escalation. Gadgets that allow an attacker to perform an action their normal privileges do not allow.
- Cryptographic Downgrade. Gadgets that downgrade the cryptography used by the application to be weaker.
- Path Traversal. Gadgets that allow the attacker to manipulate the path of file system operations.
- Unauthorized Modifications. Gadgets that allow the attacker to trigger modifications that should not happen as a result of normal operation.
- Log Pollution. Gadgets that change or control the contents of program logs.
- Denial of Service (DoS). Gadgets that deny access to the application.

We posit that many applications use some of these APIs in practice because of the importance of the functionality they provide. Furthermore, we assume that the runtime's own test suite contains a representative sample of ways to use the APIs. As a direct consequence, the presence of a gadget in a runtime implies vulnerabilities in real-world applications.

3 Overview

At a high level we develop a semi-automated dynamic analysis pipeline, GHUNTER, for finding gadgets in runtime environments, as depicted in Figure 1. To achieve this goal, GHUNTER operates in three automated steps and one manual step. Driven by the runtime's test suite, the first step identifies candidate properties for prototype pollution by detecting undefined property accesses. In the second and third step, GHUNTER uses these candidate properties to simulate pollution and detect reachability of dangerous sinks and unexpected termination, respectively. These steps also rely on the runtime's test suite and generate output for gadget identification. The final step consists in manually verifying the results of the second step, after preprocessing, using visualization of SARIF files in IDEs, and generating proof-of-concept exploits.

Listing 2 shows a universal gadget in Deno, which we will use to illustrate the workflow of GHUNTER along with the different challenges we have to tackle. Consider an application that uses the runtime API `fetch`, defined in Listing 2, to fetch user details from another service, for a given trusted user identifier `uid`. The application will eventually execute the command `fetch("https://192.168.3.14/users/"+uid)` to safely retrieve user information. Given the assumption that the application is vulnerable to prototype pollution, our goal is to find out how we can use prototype pollution to turn this seemingly benign request into a malicious gadget.

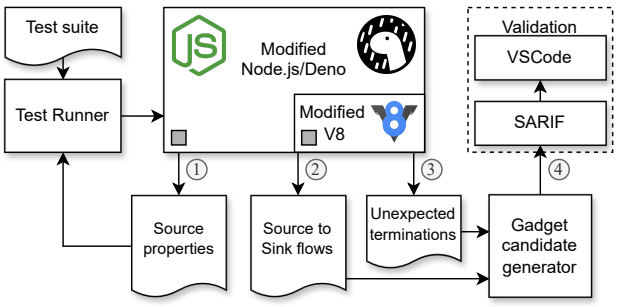

Figure 1: Architecture and workflow of GHUNTER.

Step 1: Collecting source properties A key requirement is to find properties that influence the behavior of a runtime API. These properties must not be defined so that they are looked up in the prototype chain and a polluted value is used instead. Hence, GHUNTER needs to determine which undefined property accesses happen as a result of normal usage of a target runtime API. This is achieved by observing the runtime behavior of code and taking note of undefined property accesses. Moreover, GHUNTER uses the runtime environment's test suite as a representative sample of normal usage of the API.

For the `fetch` API in Listing 2, GHUNTER runs Deno's test suite to collect a list of undefined properties that includes `method` (line 3) and `signal` (line 9). This leads us to our first challenge of automatically identifying undefined property accesses driven by the test suite of runtime APIs, which we discuss in Section 4.1.

Step 2: Identifying source-to-sink flows GHUNTER uses the list of undefined property accesses from the previous step as sources for further analysis. To determine if a property is used for a purpose that is exploitable, GHUNTER implements a lightweight taint analysis that identifies the reachability of values of polluted properties into dangerous sinks. Driven by the test suite, it pollutes the undefined properties with taint values and checks whether these values affect the native (C++/Rust) code of the runtime environment, which conservatively represents security-relevant sinks.

The function call to `op_fetch` in Listing 2 (line 13) executes Deno's native networking implementation for `fetch`. To determine if a polluted value can reach `op_fetch`, GHUNTER simulates prototype pollution and detects the polluted property value in the call to `op_fetch`. For the property `method`, GHUNTER pollutes the property with a taint value and runs the corresponding test case, while intercepting every call to `op_fetch` and checking all arguments for the presence of the taint value used for pollution. Indeed, given the list of properties for `fetch`, GHUNTER finds that the property `method` reaches the sink `op_fetch` on line 13. This leads us to our second challenge of automatically identifying flows from undefined properties to sinks, which we discuss in Section 4.2.

Step 3: Unexpected termination If normal usage of a runtime API (as represented by the test suite) does not result in a crash but the pollution of an undefined property does cause the API to crash, it implies that an attacker can use the API to cause Denial of Service (DoS) attacks. Similarly to Step 2, GHUNTER leverages the runtime's test suite to detect DoS attacks pertaining to prototype pollution. When polluting the property `signal` on line 9, GHUNTER causes the `fetch` API to crash due to a type error on line 14. This leads us to our third challenge of automatically identifying fatal crashes that cause DoS attacks on applications that use the APIs under pollution, which we discuss in Section 4.3.

```
1  class Request {
2    constructor(input, init = {}) {
3      this.method = init.method || "GET";
4      // ...
5    }
6  }
7  function fetch(input, init = {}) {
8    const request = new Request(input, init);
9    const promise = mainFetch(request, false,
         request.signal);
10   //...
11 }
12 async function mainFetch(req, recursive,
       terminator) {
13   const res = op_fetch(req.method, /*...*/);
14   terminator[abortSignal.add]();
15   //...
16 }
```

Listing 2: Simplified Deno fetch implementation.

Step 4: Manual validation The previous automated steps yield a list of potential sinks and unexpected program crashes pertaining to pollution of undefined properties. These results do not necessarily imply that a runtime API is exploitable, but require manual validation. To aid the security analyst, GHUNTER supports processing (e.g., removal of duplicates from different test cases) and representation of results in SARIF format for visualization within an IDE.

In our example, the SARIF file contains two results, called gadget candidates, for the `fetch` API: One for property `method` reaching the sink `op_fetch` and one for property `signal` resulting in a program crash. The manual analysis of `method` reveals that an attacker can override the default HTTP method from `GET` at wish, revealing a true gadget. For instance, they can pollute `method` with value `DELETE`, thus causing the command `fetch("https://192.168.3.14/users/"+uid)` to delete user records (in Section 5 we extend this attack to full Server Side Request Forgery). The analysis of the program crash due to `signal` reveals an attacker can perform a DoS attack, thus denying users of access to data. In Section 4.4 we discuss this final challenge of effectively validating gadget candidates.

4 System Design and Implementation

We design GHUNTER to overcome the challenges outlined in Section 3. In line with the architecture and workflow of Figure 1, this section describes and motivates our design and explains how it supports comprehensive analysis of JavaScript runtime environments for finding gadgets. First, we discuss source properties and detail our approach to capturing them exhaustively. Second, we show how to achieve comprehensive coverage for sinks into native runtime code and how to identify source-to-sink flows by our lightweight taint analysis. Third, we discuss unexpected termination and how to detect fatal terminations leading to DoS attacks. Finally, we discuss the process of preprocessing and manually validating results, as well as the current limitations of GHUNTER.

Along with the discussion of the design we also describe the implementation of GHUNTER, which we implement against Node.js `v21.0.0` and Deno `v1.37.2`. These are the most recent versions of the respective runtimes that share a common V8 engine version, namely `v11.8.172.17`.

4.1 Source Properties

In this work we consider undefined property accesses as *sources*. At a high level, an undefined property access happens when code tries to read a property that is not one of the object's own properties. There are many ways in which this can happen in JavaScript, including `obj.prop` as seen on line 3 of Listing 2, computed property names such as `obj[str_var]`, array-indexed properties such as `obj[1]`, for-in loops, and various syntactic sugar forms such as destructuring assignment. These features pose significant challenges for static analysis approaches [43], leading to both false positives (due to conservatively computing undefined properties) and false negatives (due to computed property names).

To ensure we comprehensively capture all undefined property accesses we modify the V8 runtime to trap on property accesses that are looked up but not present in the root object's prototype object. This conservatively covers all property accesses that may be influenced by prototype pollution, excluding pollutions with other side effects (i.e. existing prototype properties) and circumstantial pollutions of specific types.

Because gadgets are pre-existing runtime API function calls in application code, we are interested in undefined property accesses that happen as a result of normal API usage. Thus, we leverage the runtime's test suite as a proxy of real API usage and capture all undefined property access that occur during test execution. We store the observed property names on a per-test basis for use in the next steps.

For our example of Section 3 this step yields 95 properties for `fetch` from the `fetch_test.ts` test suite in Deno.

Implementation To intercept all property accesses, we modify the code of `Runtime::GetObjectProperty` and `LoadIC::Load` methods, which look up the property name in an object's prototype chain to read a property value. If the property is not found in the chain we log the access attempt.

However, V8 implements optimizations to avoid slow calls to these methods when the property name can be easily determined, as in `obj.prop`. Thus, we deoptimize the inline caches [12] and remove the bytecode handlers in the methods `AccessorAssembler::LoadIC_NoFeedback` for named properties and `AccessorAssembler::KeyedLoadIC` for array-indexed properties. This allows us to trap on every property access, albeit with some performance degradation.

We also implement a separate file logger to dump the results of our tests and extend the globals object with the `log` function. This enables our modifications in the test suite to use the same logs for dumping call stacks as described later in this section. The changes to V8 are limited to 8 files and modify 233 lines of code in total.

4.1.1 Simulating Pollution

Given the names of undefined properties that are accessed for a test, we want to simulate pollution of these properties to observe how it affects the behavior of the runtime. To this end we extend the test runners to automatically modify test files by injecting a code snippet that simulates prototype pollution.

To maximize effectiveness, the polluting snippet is injected at the top of the test file. This ensures the entire test execution is affected by the pollution. In comparison to injection using preloaded modules (e.g. through `--require` or `--module` in Node.js) this avoids affecting irrelevant accesses that happen before the test is started.

We use this prototype pollution simulation in the next two steps. In particular, if N unique undefined property accesses were detected for a test, we run for both the second and third stage of GHUNTER with N different instances of that test, each with a different property polluted.

For our example this means the `fetch_test.ts` test file in Deno is dynamically updated on the fly with a snippet that pollutes one of the 95 detected properties at a time.

Implementation We use two types for the injected values: strings and objects. To assign the property we use `Object.defineProperty` to add gettable (and settable) value. This allows us to output a stack trace for all accesses to that property. Additionally, we utilize this getter to return a unique identifier (incremental number) for every access so that we can match sources and sinks by the tainted value. Listing 9 shows the injected snippet for string values, while the snippet for object values is similar [14].

One of the values we use is a hexadecimal string so that it can be converted into a number, if needed. To support code that expects `Object` as the type for polluted values, we inject objects built based on JavaScript `Proxy`. These tainted values emulate the reading of arbitrary properties via `ProxyHandler`, access to an iterator to support for-of loop against this object,

and conversion to primitive types. Each of these access methods also produces a tainted value to propagate the taint mark.

4.2 Source-to-Sink Flows

We consider function calls where JavaScript executions flow into the runtime's native code as *sinks*. To be able to exhaustively cover such sinks we study the ECMAScript standard [7] to determine function calls that flow into V8 as well as the runtime's development documentation to understand where such flows occur for the runtime's native modules.

For V8, we find that functions such as `eval` and `new Function()` are the sinks that create a function at runtime from their string arguments. In particular, both functions create and subsequently execute JavaScript code. Thus, if a polluted value is used as (part of the) input to these functions, an attacker can potentially execute arbitrary code.

For Node.js, based on its contributor documentation [1] and source code, we identified internal APIs that interoperate with the C++ implementation from JavaScript: *linked bindings* and *internal bindings*. After conducting tests, we confirmed that *linked bindings* are intended for developers to extend Node.js with additional C++ bindings, and this method is not used for Node.js runtime APIs. Consequently, we determined that *internal bindings* comprehensively cover all data flows from JavaScript to the C++ part of Node.js and are implemented in a single JavaScript file: `lib/internal/bootstrap/realm.js`.

For Deno, similar to Node.js, we identify *bindings* as the only bridge between JavaScript and Rust. This is based on the contributor documentation for `#[op]` and `#[op2]` Rust attributes used throughout the Deno code base. As a result we identify a single template file written in JavaScript in the `deno_core` codebase that comprehensively covers all flows from JavaScript to Rust: `core/runtime/bindings.js`.

When the sink receives a tainted value as one of its arguments, it logs information about the sink being reached. This includes the sink name, call stack, tainted value with an identifier for source matching, and the access path if the tainted value is detected in a nested property of the argument.

For the running example of Section 3 this step yields only one result in Deno, namely that of pollution of the `method` property into the `op_fetch` binding.

Implementation To capture flows involved in creating functions at runtime, we modified the method `Compiler::GetFunctionFromEval()`. This method generates a function from a string passed into its first argument. Public APIs such as `eval` and `new Function()` use this method. We test the value of the first argument, and if it contains our tainted mark as a substring, we log the argument's value along with a record that this sink was triggered.

To capture the flows via binding code we implement a wrapping layer that we apply to all bindings for both runtimes. This wrapper recursively replaces all functions on a JavaScript object with a new function that inspects the arguments for tainted values, calls the original function, and returns its result. If a tainted value is detected we log the sink name, the argument index, the current stack trace, and (if applicable) the path to the tainted value for objects (e.g. `x` if the value of property `o.x` was tainted). This wrapper consists of approximately 380 lines of JavaScript code and is used in both `realm.js` and `bindings.js` for Node.js and Deno respectively.

4.3 Unexpected Termination

Besides dangerous sinks we are also interested in pollutions that result in unexpected or non-termination of the program, indicating potential DoS attack. We focus on fatal crashes that JavaScript code cannot catch and thus terminates the application immediately. Because crashes may happen with no tainted value reaching a sink, we perform this evaluation separately. GHUNTER can also detect non-fatal crashes (catchable in JavaScript), which we do not include in our results.

To comprehensively cover unexpected termination as a result of pollution, we monitor all test executions and look for processes that exit with a non-zero exit code. If a non-zero exit code is detected we evaluate the stdout and stderr of the process to filter out expected failures such as test failures in order to report only unexpected errors such as segfaults/panics, Out Of Memory (OOM), and timeouts.

To avoid reporting crashes that may happen as a result of our runtime modifications, we perform this analysis on the original runtimes. This works because this stage relies exclusively on externally available information, namely the previously-obtained list of undefined property accesses.

For the running example of Section 3 this step yields only one result in Deno, namely that of pollution of the `signal` property leading to an unexpected `TypeError`.

Implementation To perform this part of the analysis, we re-use the test runner that modifies test files with prototype pollution and instruct it to use the unmodified version of the runtime. We extend the test runner to examine the exit code and output (`stdout` and `stderr`) for each test it runs. In particular, if the exit code is nonzero, it will check if the output matches an expected error (e.g. a test failed) and if it does not, log the polluted property name and process output.

4.4 Manual Validation

To effectively validate and create proof-of-concept exploits from the results of Section 4.2 and Section 4.3, we produce a SARIF file with all necessary information for manual validation. The SARIF file format, in combination with a SARIF file viewer, provides a convenient way for an analyst to interactively view results and browse relevant code locations.

We preprocess the output of stages 2 and 3 to obtain a *gadget candidate* for each unique detected sink or unexpected termination. For a reached sink, this is determined by the

property name and the stack trace for the sink call or the stack trace for the polluted property access. For unexpected termination, this is determined by the termination output.

For each gadget candidate, we include all relevant information for validation and creation of a proof of concept. For detected sinks the gadget candidate is presented as a triple consisting of the polluted property name as well as the API and sink represented by the stack trace for the source and sink (SARIF viewers allow for interactively browsing the stack). We also provide the value observed at the sink which helps the analyst understand if the runtime manipulates the polluted value. For unexpected terminations, we are limited to providing the program output after the crash, but additionally we provide the name of the polluted property as well as the test file that crashed.

While each result represents only a single polluted property, if multiple properties affect the same API and sink these results will be co-located in the generated SARIF file. This allows the analyst to combine multiple properties in a proof of concept. Thus, in contrast to a gadget candidate, a *gadget* is a triple consisting of the set of properties, API and sink. We remark that GHUNTER only detects that a value reaches the sink but not the intended type or structure of that value. The analyst has to analyze the API documentation and code to understand what values to use in the proof-of-concept exploit.

For the running example of Section 3, the SARIF file contains two entries, one for the detected flow from the property `method` to the sink `op_fetch` and one for the unexpected error as a result of polluting the property `signal`.

Implementation We generate the SARIF file from the logs of the second and third stages. For the second stage we look for sinks where a tainted value was observed and the corresponding source (property access for that exact value). As a result any source that does not reach a sink is automatically discarded. If no source can be found for a taint value at a sink (e.g. due to modifications to the value), it is reported to the analyst separately. For the third stage we report any test run resulting in a non-zero exit code with a stderr message other than a test failure, excluding tests that failed in the initial run.

4.5 Limitations

Full-fledged taint tracking Our lightweight taint analysis favours performance. This can be seen as a limitation with respect to manual validation because the complete flow from source to sink is not readily available. In practice, we find that the runtime code is relatively simple for most cases, and the flow from source to sink can be identified quickly. Secondly, our lightweight taint tracking may miss flows from sources to sinks in the event that the taint value is removed in certain operation (e.g. splice). Again, we observe that most runtime code does not perform modifications on values beyond simple transformations such as converting a string to uppercase.

Polluted types The pollution simulation only pollutes using strings and objects. We could additionally cover numbers and arrays for pollutions (booleans cannot be taint tracked with our approach). This would only find flows where an explicit type check prevents the tainted value from reaching a sink. Besides polluting with different types, techniques such as concolic execution [31, 47] could be used to improve coverage too.

Gadget chains In contrast to works on gadget detection in libraries and frameworks [31, 44], GHUNTER cannot find gadget chains where one pollution enables another. This is because GHUNTER pollutes only a single property at the time. Running an analysis where multiple properties are polluted at the same time is possible in theory, but infeasible in practice due to the number of possible combinations of properties.

Binding coverage For Node.js we are unable to cover 25 bindings because they exist at a property that is not configurable or not writable, thus preventing us from wrapping them. We evaluated these functions and find them to have little security relevance. For Deno we were unable to wrap 4 bindings, all async, because they do not take any arguments. Such sinks are not interesting for our analysis so we consider this a non-issue.

Test suite limitations Our approach relies on the comprehensiveness of the runtime's test suite. We are thus limited in our analysis by the coverage of the source code by the test suite. We evaluate the coverage statistics and find 95.8% and 91.4% function coverage in Node.js and the Deno standard library respectively. These percentages give confidence in the comprehensiveness of our analysis.

5 Evaluation

This section describes the results of our comprehensive evaluation on Node.js and Deno, answering the research questions:

- **RQ1:** How can we effectively identify exploitable universal gadgets in the Node.js and Deno runtimes?
- **RQ2:** How does GHUNTER compare to Silent Spring?
- **RQ3:** What is the performance overhead of our taint-enhanced runtimes as compared to the original runtimes? How to empirically validate transparency of our taint-enhanced runtime with respect to the original runtimes?

Experimental setup We conduct our experiments on an AMD EPYC 7742 64-Core 2.25 GHz server with 512 GB of RAM. To optimize server resource utilization, we execute tests in parallel. We utilize a modified test runner script that runs test files in parallel with a 20 second timeout per test file. For Node.js we adopt the existing `test.py` runner, for Deno we write a custom runner that invokes `deno test`.

5.1 Universal Gadgets in Node.js and Deno

We demonstrate the effectiveness of GHUNTER through the number of detected gadgets in light of the number of outputs for intermediate analysis steps.

Analysis of Node.js The target of our analysis of Node.js is the standard library built into the Node.js binary. The first step of our analysis produced 509,481 unique test-property combinations for 3,782 test files. The second and third steps of our analysis found 22,860,092 sinks reached, 9,743 segfaults, and 6 tests that timeout. Preprocessing of results reduced the number of sink-source pairs to 13,029 unique pairs and segfaults to 13 (no reduction in test timeouts). Furthermore, we excluded source-sink pairs that could only lead to Denial of Service: 11,730 sinks related to infrastructure code such as type checking, internal utils, asynchronous call wrappers, exception and error message builders; 120 in `buffer.byteLengthUtf8`; 258 in `messaging.postMessage`, which sends messages between workers; and 101 in the `buffer` parameter in `fs.read` which is used for output of the sink call. After filtering, there are 820 gadget candidates out of which we confirmed 56 to be exploitable. The manual verification process required 31 person hours.

Analysis of Deno Our analysis of the Deno runtime covers the core API (accessible by `Deno`), the Node.js compatibility module, and the Deno standard library. We ran our pipeline on each separately, but accounted for duplicates when aggregating the results, which we report here.

The first step of our analysis produced 21,786 unique test-property combinations for 596 test files. The second and third steps of our analysis found 13,519 sinks reached, 1 panic, and 139 tests that timeout. Preprocessing of results reduced the number sink-source pairs to 399 unique pairs, 18 tests that timeout, and no reduction in panics. As a result, we obtained 418 gadget candidates out of which we confirmed 67 to be exploitable. The manual validation took 15 person hours.

Node.js vs Deno We observe quite a large difference in numbers when comparing Node.js to Deno. First, Node.js produces significantly more results. One reason for this is that Node.js has a larger test suite (both in terms of test files and test cases). Despite Deno's security focus, we find similar number of exploitable gadgets. One reason for this is that Deno has a larger API surface. Another is that prior work on gadgets has resulted in some protections being implemented in Node.js, in fact some of the gadgets we find in Deno were previously identified and addressed in Node.js.

Result classification We categorize our universal gadgets by the strongest exploit they can be used for. If multiple properties can be combined to achieve a stronger exploit, we consider only the combination and not the weaker exploits pertaining to a subset of properties. Table 1 shows the aggregate number of gadgets per exploit category.

We omit gadgets without a security impact or that only cause a JavaScript exception (they have limited impact since applications can catch such exceptions). We include gadgets that presume an existing vulnerability (e.g. to write a file on the systems) and call these *second order* gadgets.

New detected gadgets We highlight 4 gadgets here and refer to Table 5 and Table 6 in Appendix, and code artifact [14] for the complete list of gadgets.

Attack Type	Node.js	Deno
Arbitrary Code/Command Execution	14	5
Server Side Request Forgery	6	3
Privilege Escalation	7	24
Cryptographic Downgrade	2	0
Path Traversal	3	10
Unauthorized Modifications	0	10
Log Pollution	0	1
Panic/Segfault	12	1
Out of Memory	0	3
Infinite Loop	0	2
Second Order	12	8
Total	56	67

Table 1: Number of gadgets found by type per runtime.

Listing 3 shows a proof of concept (PoC) of the `fetch` gadget from Section 3. In addition to the property `method`, polluting the properties `body` and `headers` allows attackers to control all aspects of the request to the application-specific URL. Moreover, due to the way Deno's `fetch` implementation stores request URLs internally, the pollution of property `0` allows the attacker to override the URL and achieve SSRF. This gadget transforms a simple benign-looking request like `fetch("http://example.com")` into a completely unrelated HTTP request.

```
1  // send a POST request to http://fake.com
2  ////////////////////////////////////////////////
3  // PROTOTYPE POLLUTION:
4  Object.prototype[0] = 'http://fake.com'
5  Object.prototype.method = 'POST'
6  Object.prototype.body = '{"pwned":"yes"}'
7  Object.prototype.headers = {"content-type":"
       application/json"}
8  ////////////////////////////////////////////////
9  // GADGET:
10 fetch('http://example.com')
```

Listing 3: PoC of *fetch* gadget (Deno).

Similarly, we found that the `fetch` API of Node.js can also exploited to achieve SSRF attacks. In addition to controlling `method` and `body`, an attacker is able to pollute `socketPath` to redirect HTTP requests to a local socket rather than the specified URL. This gadget can be exploited to target local daemons, such as Docker.

Another universal gadget in Deno allows for path traversal on temporary files. Polluting `dir` allows an attacker to control where `Deno.makeTempDir` and `Deno.makeTempFile` create temporary file system entries. Even if `dir` is specified by the application, `prefix` still allows for path traversal by using a string like `../` as a prefix (prior to Deno v1.41.1). Depending on how the temporary file is used, this gadget can be a setup for a stronger attack.

We also identify two new Arbitrary Code Execution (ACE) gadgets in Node.js, located in the commonly used `require` and `import` functions. The gadget in `require` has been fixed

API	GT	Silent Spring			GHunter		
		GC	TP/FP	FN	GC	TP/FP	FN
cp.exec	2	20	1/19	1	3	2/1	0
cp.execFile	1	16	0/16	1	2	1/1	0
cp.execFileSync	4	21	3/18	1	7	4/3	0
cp.execSync	4	13	3/10	1	7	4/3	0
cp.fork	2	25	1/24	1	6	2/4	0
cp.spawn	3	14	2/12	1	5	3/2	0
cp.spawnSync	4	11	3/8	1	7	4/3	0
import	1	0	0/0	1	5	1/4	0
require	3	19	2/17	1	4	1/3	2
vm.compileFunction	1	4	1/3	0	5	0/5	1
Total	25	143	16/127	9	51	22/29	3

Table 2: Silent Spring vs GHunter on Node.js v16.13.1 with properties used in Silent Spring gadgets as ground truth.

API	GT	Silent Spring			GHunter		
		GC	TP/FP	FN	GC	TP/FP	FN
cp.exec	1	9	0/9	1	2	1/1	0
cp.execFile	1	9	0/9	1	2	1/1	0
cp.execFileSync	4	11	3/8	1	7	4/3	0
cp.execSync	2	3	1/2	1	3	2/1	0
cp.fork	1	5	0/5	1	1	1/0	0
cp.spawn	3	9	2/7	1	5	3/2	0
cp.spawnSync	4	6	3/3	1	7	4/3	0
import	1	0	0/0	1	1	1/0	0
vm.SyntheticModule	3	3	1/2	2	1	1/0	2
Total	20	55	10/45	10	29	18/11	2

Table 3: Silent Spring vs GHunter on Node.js v21.0.0 with properties used in GHunter ACE gadgets as ground truth.

as of Node.js v18.19.0. We detail this gadget and its fix in Section 6.3. The gadget associated with `import`, shown in Listing 4, can be exploited by polluting the `source` property with JavaScript code and invoking the `import` function on any `.mjs` file. This causes the code from the property to be evaluated.

```
////////////////////////////////////////////////
// PROTOTYPE POLLUTION:
Object.prototype.source ='console.log("PWNED")'
////////////////////////////////////////////////
// GADGET:
import('./any_file.mjs')
```

Listing 4: PoC of *import* gadget (Node.js).

5.2 GHunter vs Silent Spring

We compare the effectivess of GHunter and Silent Spring [43] in finding universal gadgets. Silent Spring can detect prototype pollution statically and also universal gadgets in Node.js using a mix of dynamic and static taint analysis. The two approaches differ in non-trivial ways. GHunter uses dynamic analysis to detect pollutable properties at runtime and it is driven by the test suite of a runtime environment. In contrast, Silent Spring syntactically identifies any property reads and uses them in a dynamic analysis to check if they are pollutable. This causes challenges with properties that are not identifiable statically, for example computed properties. Moreover, GHunter analyzes all APIs systematically (subject to coverage by the test suite), while Silent Spring analyzes only 3 APIs.

Because of these differences and the fact that some of the gadgets from Silent Spring have since been fixed, we perform the following comparison: we use the gadgets identified by both toolchains as a basis for ground truth and evaluate whether or not each tool finds a gadget candidate (GC) for each *property* used in the gadgets for a given API. This is because both toolchains can only taint/pollute one property at a time and report one GC per property. We focus only on ACE gadgets as was the case in Silent Spring.

Our first experiment uses the gadgets of Silent Spring as a ground truth on Node.js v16.13.1. We recreated PoCs for all its gadgets to determine the affected APIs and necessary properties. Based on this we created new test cases in the style of Silent Spring's dynamic analysis. We reran both Silent Spring and GHunter on Node.js v16.13.1 using these new test cases to obtain the results shown in Table 2. Ground truth (GT) is the number of GCs required to identify all gadgets of an API. False negatives (FN) represent the number of GCs that were identified manually (and not by a tool), but are in the GT of a gadget. We see that GHunter is more precise (0.43 compared to 0.11) and has better recall (0.88 compared to 0.64). This is due to the underlying dynamic analysis, which guarantees that a polluted property reaches a sink. GHunter has three FNs because it lacks features necessary to detect the sink (the `require` gadget requires a chain of pollution; the `vm` gadget requires array support). For Silent Spring we find nine FNs. The FNs for child process (`cp`) are due to the lack of support for `for-in` analysis, causing it to miss one variant of the gadgets. For `import` it fails to detect the gadget API and for `require` it fails to detect one property; in these cases the true and false positives would have allowed the analyst to extrapolate the properties reported as FNs here.

Our second experiment uses the gadgets of GHunter as a ground truth on Node.js v21.0.0. For a fair comparison, we created test cases for ACE gadgets from Table 5 in the style of Silent Spring's dynamic analysis. We reran both GHunter and Silent Spring on Node.js v21.0.0 using these new test cases to obtain the results shown in Table 3. For this selection of gadgets, GHunter finds more gadgets while reporting fewer gadget candidates, again showing better precision (0.62 compared to 0.18) and recall (0.90 compared to 0.50), requiring less manual work. Silent Spring again exhibits FNs for all child process APIs because it lacks support for `for-in` construct. For the `import` gadget, Silent Spring fails to detect the API that triggers the gadget.

In summary, these experiments show that GHunter is more precise, resulting in less manual work required and higher accuracy. We believe this is primarily due to the fully dynamic approach used by GHunter, which guarantees every GC reaches a sink and provides support for dynamic language features. The shortcomings of GHunter are due to the limitations discussed in Section 4.5.

5.3 Performance Overhead and Transparency

We evaluated the performance overhead incurred by GHUNTER in comparison with the unmodified JavaScript runtimes. To evaluate the effect of the customized runtimes and the customized V8 engines on the behavior of runtime APIs, referred to as transparency, we use the test suites as oracles to identify behavioral changes.

Node.js Running the full Node.js test suite, which contains 3,810 tests, using our modifications increased runtime by 111.72% (from 252s to 542s). The success rate decreased from 3,782 to 3,669 cases, marking a 2.99% reduction. The number of tests failing due to timeout increased from 2 to 44 cases.

Deno Running the three different test suites using our modifications increased runtime by 4.46% (from 157s to 164s) for Deno core, by 43.85% (from 130s to 187s) for Deno's Node.js compatibility module, and by 5.93% (from 253s to 268s) for Deno std. In total that is 14.63% (from 540s to 619s). The success rate decreased by by 0.17% (from 1,145 to 1,143 out of 1,340) for Deno core, by nothing for Deno's Node.js compatibility module, and by 0.27% (from 2,207 to 2,201 out of 2,258) for Deno std. In total that is 0.15% (from 5,364 to 5,356 out of 5,648). The number of tests failing due to timeout increased from 1 to 2 cases.

Evaluation The main reason for the decreased performance and higher failure rate is the code responsible for checking tainted values in internal sinks. This code recursively traverses received values of each argument of the sink. Unexpected exceptions in the traversed objects' code, such as in property getters, lead to failures. Additionally, the modified version extends `globalThis` with `log`, causing some tests to fail.

6 Defense Best Practices

While previous works provide convincing evidence on the dangers of prototype pollution, as of today, there is no comprehensive defense against this vulnerability. In this section, we systematize the current proposals and mitigations and outline directions for future work. Since our universal gadgets require the existence of prototype pollution, a reasonable question to ask is whether we should mitigate the impact of the vulnerability by fixing the gadgets. Given the lack of comprehensive defenses against prototype pollution, we think that gadgets should be treated similarly to memory corruption vulnerabilities such as return-oriented programming (ROP) and jump-oriented programming (JOP), due to their high impact. Developers of runtimes or libraries are unaware of the presence of prototype pollution in the applications using their code. Therefore, it stands to reason to assume the presence of vulnerabilities and treat the prototype objects as untrusted data, thus guaranteeing security by fixing gadgets in their code. Similarly, application developers are unaware of prototype pollution in third-party libraries or runtimes of their application, hence they should mitigate gadgets.

6.1 Gadget Mitigations

Gadget can be mitigated by avoiding the use of potentially polluted properties in the code. A solution is to ensure that any access to the properties of an object does not fall back to the object's prototype chain. We distinguish different mitigations depending on where in the code an object with a polluted prototype may be *created*. This can be either the developer's own code (e.g., a library or module) or third-party code (e.g. dependencies or application code that use APIs provided by the developer). This leads us to the first guideline.

G1: Explicit access to own properties

If the code accesses a property in only a few instances, developers should verify each access explicitly.

Developers should check if an object defines an own property before accessing it. This can be achieved with built-in methods such as `Object.hasOwn(obj, 'prop')`. We encountered this pattern regularly during our analysis of for-in loops to prevent reading unexpected properties. These checks should be added every time a potentially undefined property is accessed, thus preventing access to a polluted property. This guideline can be applied regardless of where the object being checked was created. However, overuse of these checks increases the codebase's complexity. Therefore, developers should follow other recommendations whenever their code makes use of many property accesses. We also recommend using the method `Object.keys`, which returns the object's own enumerable properties rather than for-in loops, which additionally iterate over properties in the prototype chain.

G2: Safe object creation

When creating an object, developers should use either `null` prototypes or built-in objects `Map` and `Set`.

The method call `Object.create(null)` and the object literal `{__proto__:null}` allow to create objects that do not inherit from the prototype hierarchy. In this case, any property access `obj.prop` returns `undefined` unless `prop` is an own property of object `obj`. On the downside, this solution can lead to unexpected exceptions. For example, code patterns like `obj + "str"` will throw an exception because no `toString` method is available without the prototype.

When the created object is returned by the underlying function or it is passed as an argument to a third-party function, developers should copy the object to a new object that includes `Object.prototype` to ensure backward compatibility. We recommend assigning default values to unused properties to prevent pollution with attacker-controlled values in third-party code. This operation can be facilitated by, e.g., using

the method `Object.assign({}, defaultObj, obj)`. We remark that the prototypes of nested objects require cloning the object by means of a deep copy algorithm, for example, using the global method `structuredClone()`.

An alternative solution is to use built-in objects that provide safe access to properties. For instance, the `Map` object holds key-value pairs and provides methods such as `Map.get` that do not use the prototype chain to look up the stored values. Hence, `map.get('prop')` can serve as a replacement for accesses to objects.

G3: Safe copy of input data

Whenever an object is received as input data, developers should copy the object's properties to a safe object.

If a developer uses an object as a function argument (for example, `options` in Listing 5), or an object originating from a deserialization function (for example, `JSON.parse` in Listing 7), they should assume that the object's prototype can be polluted. A safe solution is to copy the expected properties to a new object with `null` prototype. This can be achieved by creating a copy with only own properties, using the expression `{__proto__:null,...obj}`. If the code returns the received object back, the developers should use the original value instead of the copied one to avoid compatibility issues.

The guidelines G1 and G3 may be backward incompatible when an object relies on a prototype chain to define properties within nested prototypes. We expect this design pattern to be used for functions rather than data-type properties, which are subject to prototype pollution. An empirical evaluation is necessary to validate this claim.

As we can see, systematic mitigation of gadgets is an open problem. Developers are expected to identify all gadgets to universally apply mitigation techniques to any potentially undefined property, which is infeasible in practice. Moreover, gadget mitigation can be hard to apply to existing code bases since it requires identifying every access to undefined properties. These considerations motivate the need for solutions like the one proposed in this paper but we believe the guidelines can be automated as suggestions for quick fixes in IDEs or similar tooling. Detection may require inter-procedural analysis, yet we expect that G1 and G2 can be implemented based on quick intra-procedural analysis.

6.2 Prototype Pollution Mitigations

Prototype pollution is the root cause for exploitation of gadgets, hence a comprehensive mitigation technique would solve the problem altogether. As with gadget mitigations, this requires striking a balance between security and usability, which makes it a challenging task. Here we discuss recommendations for developers and opportunities for researchers.

Guidelines for developers A general solution is to prevent any accesses to the prototypes of objects, which can be achieved by the above-mentioned guidelines for gadget mitigation. Following guideline G1, developers should avoid accesses to object prototypes through property reading expressions. This is because properties such as `__proto__` and `constructor.prototype`, which give accesses to the prototype chain, are not defined in the object itself. Alternatively, this can also be achieved by explicitly checking accesses to properties `__proto__`, `constructor`, and `prototype`. Similar to own property checks for gadget mitigation, this mitigation introduces additional verbosity. Following guideline G2, one can instead use data structures with either `null` prototypes or safe `get` and `set` functions.

Another solution is to prevent unintended modification to the prototype object itself, which can be achieved with built-in functions such as `freeze`, `preventExtension`, and `seal` [5]. These functions offer a mechanism to prevent the creation of new properties on an object. The `freeze` function additionally prevents overwriting. Node.js provides the experimental command-line feature, `--frozen-intrinsics`, which freezes the prototypes of built-in objects like `Array` and `Object`. Similarly, Deno removes `__proto__` from `Object.prototype` by default.

While mitigating prototype pollution, these solutions can be problematic for third-party packages that rely on changing the prototype to implement, e.g., polyfills. Also, they require coverage of all prototype object, including user-defined classes which makes it verbose and hard to maintain for large projects. We recommend these solution for the development of a new project while existing project should perform regression testing to ensure that no functionalities are disrupted.

Research opportunities Mitigation of prototype pollution and gadgets remains an open problem. A recent proposal driven by Google aims to prevent prototype pollution at the language- and runtime-level [6]. It proposes an opt-in *secure mode*, which, if enabled, prevents accesses to prototypes with dynamic string keys. It allows prototype access through reflection APIs instead of strings, thus only requiring changes to `__proto__` and `constructor`, whenever they are accessed purposefully. While an important step in the right direction, this solution poses challenges of backward compatibility for server- and client-side applications.

6.3 Case Studies

We evaluate fixes of known server-side prototype pollution vulnerabilities and their gadgets to identify common issues in mitigations that permit attackers to bypass the fixes. We conducted our search through public vulnerability reports on HackerOne, blog posts, and publications related to open-source applications over the past 5 years, summarizing our findings in Table 4. Our results contain 12 exploitable cases leading to Remote Code Execution (RCE) in 4 popular applications. The root cause of their exploitability, namely code patterns that allow to pollute prototypes, has been addressed

Application	Version	Vulnerability Report	PP Fix	Gadget	Gadget Fix	App Mitigations
Kibana	6.6.0	CVE-2019-7609	✔	child_process.spawn	✖	✔ G2, G3*
	7.6.2	HackerOne #852613	✔	lodash.template	✖	✖
	7.7.0	HackerOne #861744	✔	lodash.template	✖	✔ G3
	8.7.0	CVE-2023-31415	✔	nodemailer	✖	✖
npm-cli	8.1.0	Reported by [43]	✔	child_process.spawn	✔ G2	✖
Parse Server	4.10.6	CVE-2022-24760	✔	bson	✖	✔ Denylisting
	5.3.1	CVE-2022-39396	✔	bson	✖	✔ Denylisting
	5.3.1	CVE-2022-41878	✔	bson	✖	✔ Denylisting
	5.3.1	CVE-2022-41879	✔	bson	✖	✔ Denylisting
	5.3.1	Reported by [43]	✔	require	✔ G2*, G3	✖
	6.2.1	CVE-2023-36475	✔	bson	✔	–
Rocket.Chat	5.1.5	CVE-2023-23917	✔	bson	✔	–

Table 4: A summary of the RCEs exploited via prototype pollution. For each application, we list the vulnerable version, a reference to the report, and the exploited gadget. *PP Fix* shows whether the prototype pollution was fixed; *Gadget Fix* shows whether the gadget was fixed, including any applied guidelines; *App Mitigations* details if mitigations against the attack were implemented in the application. ✖ indicates that no fix has been applied; ✔ indicates that a fix was applied but later bypassed; ✔ indicates that a fix was applied and effectively protects against similar attacks. (*) denotes a guideline that might be bypassed.

in all cases. These vulnerabilities involve 5 unique gadgets to achieve RCEs. For 4 of these gadgets, developers proposed either fixes or mitigations for the attacks.

We identify 6 vulnerabilities that exploit a gadget in the `bson` package. The Parse Server developers fixed 5 vulnerabilities that use this gadget with input data validation through denylisting. However, these mitigations were bypassed several times through unexpected means, e.g. with files metadata. Ultimately, the dangerous feature was removed from `bson`, thereby fixing the gadget. Both Parse Server and Rocket.Chat fixed their vulnerabilities through this method. This highlights the need to fix gadgets because mitigation is difficult and often leaves room for exploitation by other means.

The gadgets in `lodash.template` and `nodemailer` remain unaddressed and could be exploited given new prototype pollutions. The maintainers of Kibana banned the use of `lodash.template` in their code and mitigated it by intercepting `template` calls and validating the polluted property when the package is included as a transitive dependency.

However, as illustrated, it can be dangerous to leave gadgets unfixed. Next, we detail two interesting gadgets and highlight issues in their fixes to demonstrate the risk.

child_process.spawn The first mention of the `spawn` gadget appears in the report CVE-2019-7609 by Michał Bentkowski, outlining a prototype pollution vulnerability in Kibana. Kibana spawns a `node` process, and the security researcher discovered a method to execute arbitrary code through crafted environment variables of the new process.

Listing 5 presents the necessary code of the `spawn` function to understand the attack. If an application invokes `spawn` with two arguments, `file` and `args`, then the third argument `options` is undefined. Line 3 creates a new object that inherits `Object.prototype`, making it susceptible to prototype pollution. Line 4 makes a shallow copy of `options` to prevent changing the user's options object if passed. In our scenarios, this copy operation is inconsequential because `options` is an empty object created within the function itself. Line 5 retrieves the value of the `env` property. If the value is undefined,

```
1  function spawn(file, args, options) {
2    if (options === undefined)
3      options = {}
4    options = Object.assign({}, options)
5    options.env = options.env || process.env
6    options.file = options.shell || file
7    //...
8    internalSpawn({
9      file: options.file,
10     env: options.env,
11     //...
12   })
13 }
```

Listing 5: Simplified Node.js *spawn* implementation.

the code defaults to `process.env`, assigning this to the `env` property of options. Line 6 similarly handles the `shell` property from options and the `file` parameter. Subsequently, the code passes the aggregated options to the internal implementation of the `spawn` function, which initiates a new process. If an attacker pollutes the `env` property in `Object.prototype`, line 5 will read the attacker-controlled value instead of system environment variables. It allows the attacker to execute arbitrary code, leading to RCE in Kibana.

The Kibana team fixed the prototype pollution vulnerability and mitigated the gadget in PR #55697 to prevent similar attacks in later versions. Because the gadget is part of Node.js' source code, application developers are limited to intercepting `spawn` calls and altering the arguments. Listing 6 provides a simplified version of this mitigation. The code uses a JavaScript Proxy to invoke the `patch` function, thereby securing the options. It evaluates passed arguments from the zero-based array `args`. If the argument at position 1 is an array, line 5 simply advances the position. If the subsequent argument at position 2 is an object, it is treated as the options, and the `prototypeless` function then copies the options' own properties to new objects with null prototypes.

This mitigation follows our guidelines G2 and G3. Lines

```
1  cp.spawn = new Proxy(cp.spawn, {apply: patch})
2  function patch(target, thisArg, args) {
3    var pos = 1;
4    if (Array.isArray(args[pos]))
5      pos++ // fn(file, args, ...)
6    if (typeof args[pos] === 'object') {
7      // fn(file, options, ...)
8      // fn(file, args, options, ...)
9      args[pos] = prototypeless(args[pos])
10   }
11   //...
12   return target.apply(thisArg, args)
13 }
14 function prototypeless(obj) {
15   var newObj = Object.assign(
16     Object.create(null), obj)
17   newObj.env = Object.assign(
18     Object.create(null), newObj.env)
19   return newObj
20 }
```

Listing 6: Simplified *spawn* gadget mitigation in Kibana.

16 and 18 create new objects with null prototypes in accordance with G2, ensuring that care is also taken for nested objects to prevent pollution of `env` when the value is read from `process.env`. The use of `Object.assign` in lines 15 and 17 copies only own properties from the original objects to the new objects with null prototypes, following G3.

However, this mitigation has two critical weaknesses that allow the attacker to bypass it. Developers are constrained to validating arguments and lack control over modifications to arguments after passing them to Node.js functions. As observed in line 5 of Listing 5, the `spawn` function makes a copy of the received options into a common empty object that shares its prototype with others. Consequently, any properties of the options might be polluted again. Fortunately, `spawn` does not copy the `env` property, so environment variables are not affected. The other weakness is more dangerous and allows for bypassing all mitigations and even security fixes in Node.js, as we will see later. Lines 6 and 9 of Listing 6 are also exploitable by prototype pollution. The array `args`, like any array, has `Object.prototype` in its prototype chain and looks up an undefined property. Therefore, polluting the property `2` allows the attacker to control the options. For this exploit, a gadget trigger might look as follows:

```
1 Object.prototype[2] = { env:
2   {NODE_OPTIONS: '--inspect-brk=0.0.0.0:1337'}
3 }
4 spawn('node', ['any_file.js'])
```

Thus, the `spawn` gadget is still exploitable in Kibana after mitigations. This case highlights the importance for developers to exercise caution with security-critical code, such as gadget mitigations, and to test it against other gadgets using tools like GHUNTER to avoid introducing new exploitation flows into the code.

Shcherbakov et al. [43] introduce a variation of the `spawn` gadget. They find that the name of a running process can be manipulated through the polluted property `shell`, as shown in line 6 of Listing 5. Additionally, they disclose new payloads for the exploit that operate without controlling environment variables and controlling only one variable. They identify a vulnerability in the JavaScript package manager npm-cli, and exploit it to demonstrate the practical feasibility of using this gadget. Although npm-cli contributors addressed the reported prototype pollution, they did not mitigate the gadget.

In June 2022, the Node.js team attempted to fix this gadget in PR #43159. In terms of our terminology, they implemented guideline G2 by assigning the value `ObjectFreeze(ObjectCreate(null))` to options in line 3 of Listing 5 and eliminated `Object.assign()` in line 4 to maintain the usage of options with a null prototype. As discussed in Section 6.1, G2 alone is insufficient to prevent all forms of gadget exploitation, and G2 should be used in conjunction with G3. GHUNTER reports a gadget for `spawn` when a user supplies their own options object to `spawn`:

```
1 Object.prototype.shell = 'node'
2 Object.prototype.env =
3   {NODE_OPTIONS: '--inspect-brk=0.0.0.0:1337'}
4 spawn('app', ['file.log'], {cwd: '/tmp'})
```

This case illustrates the importance of a consistent approach in implementing gadget fixes. When applying guideline G2, it is crucial to carefully handle input data and copy it safely, while also applying G3. Relying on validating security-critical parameters outside the gadget proves to be insecure.

```
1  // lib\internal\modules\cjs\loader.js
2  function readPackage(dir) {
3    const jsonPath = resolve(dir, 'package.json')
4    const json = packageJsonReader.read(jsonPath)
5    if (json === undefined)
6      return false
7    return JSON.parse(json)
8  }
9  function tryPackage(requestPath) {
10   const pkg = readPackage(requestPath)?.main
11   if (!pkg) {
12     const js = resolve(requestPath, 'index.js')
13     return loadFile(js)
14   }
15   loadFile(pkg)
16 }
```

Listing 7: Simplified Node.js *require* implementation.

require Shcherbakov et al. [43] report a gadget in `require`, a built-in function in Node.js for including external modules from separate files as well as Node.js modules, and utilize this gadget in one of the Parse Server exploits. Listing 7 illustrates a gadget based on simplified Node.js code. The function `tryPackage` receives a directory path for a module and invokes `readPackage()` in line 10. The code in line 4 attempts to read `package.json` from the given directory. If the read operation is successful, `readPackage()` parses the

content of the file as JSON and returns the parsed object in line 7. `tryPackage` then accesses the `main` property in line 10, loads a file based on the path specified in the `main` property, and evaluates its JavaScript code in line 15. Consequently, if `package.json` lacks the main property, line 10 looks up the property in the prototype chain of the returned object, allowing a polluted property from `Object.prototype` to be assigned to `pkg`. This leads to the evaluation of JavaScript code from an attacker-controlled file in line 15.

The Node.js team attempted to fix this gadget by applying guidelines G2 and G3 to `readPackage` function. They correctly make a safe copy of the parsed object in line 7 to an object with a null prototype. However, GHUNTER detects a variation of the gadget in v18.13.0. If `packageJsonReader` can not find the `package.json` file, the function returns `false` in line 6. Since Boolean is a primitive type and all primitive types in JavaScript inherit from `Object.prototype`, the expression `(false)?.main` in line 10 accesses the polluted value in `Object.prototype` and assigns it to `pkg`, achieving the same attack. This makes the `require` function exploitable, albeit through a different gadget.

End-to-end exploit To demonstrate the impact of this gadget, we analyze Kibana version 8.7.0 for end-to-end exploits. We initially utilized the Silent Spring [43] toolchain to detect prototype pollution vulnerabilities. The analysis reports 44 cases in the server-side code, with 6 being potentially exploitable. The simplified code of one of the cases is presented in Listing 8. Kibana loads a config file, parses it into an object, and expands the properties from dot notation into nested objects (e.g., `{a.b:0}` to `{a:{b:0}}`) with the `ensureDeepObject` function. This code is vulnerable to prototype pollution. On line 19, the first argument allows an attacker to get a reference to the prototype and then assign a value to any property of the prototype in line 14.

To exploit this prototype pollution, an attacker should upload a configuration file with a payload via the Web UI form and restart Kibana to trigger the parsing of the new configuration file. During the restart process, Kibana crashed at an early stage due to an unexpected polluted property that prevented gadget execution via another web request. However, the application invoked `require` multiple times during loading, allowing us to trigger it and achieve RCE. The investigation process took 8 hours for one author already familiar with Kibana. We reported this vulnerability, and the Kibana team acknowledged the issue, assigning CVE-2023-31414 with a critical CVSS 9.1, and rewarding a substantial bounty. The Node.js team fixed the `require` gadget in version 18.19.0.

Takeaways If developers fix only the prototype pollution vulnerabilities while leaving its associated gadget exploitable, they remain at risk. Our case studies show that many developers are aware of this risk and attempt to mitigate the gadgets and similar attacks. However, this task is far from trivial. We identified numerous gadgets and common coding issues that lead to new gadgets, emphasizing the need for more principled solutions. Our proposed guidelines are a step forward in this direction.

```
1  function ensureDeepObject(obj: any): any {
2    return Object.keys(obj).reduce((res, key)=>{
3      const val = obj[key];
4      if (!key.includes('.'))
5        res[key] = ensureDeepObject(val);
6      else
7        walk(res, key.split('.'), val);
8      return res;
9    }, {} as any);
10 }
11 function walk(obj:any, keys:string[], val:any){
12   const key = keys.shift()!;
13   if (keys.length === 0) {
14     obj[key] = val;
15     return;
16   }
17   if (obj[key] === undefined)
18     obj[key] = {};
19   walk(obj[key], keys, ensureDeepObject(val));
20 }
```

Listing 8: Prototype pollution vulnerability in Kibana.

7 Related Work

We discuss our work in the context of closely-related works that address prototype pollution vulnerabilities and position our contributions in the area of web application security.

Universal gadgets in JavaScript runtimes The problem of identifying universal gadgets in JavaScript runtimes remains largely unexplored. To the best of our knowledge, only the work of Shcherbakov et al. [43] studies universal gadgets in Node.js. Section 5.2 compares their work to GHUNTER.

Recent work by Shcherbakov et al. [44] uses dynamic taint analysis via program instrumentation to find gadgets in NPM packages. This approach cannot be used to identify universal gadgets which require modifications of runtime environments (Node.js and Deno) and the underlying V8 engine. Our universal gadgets are complementary and contribute with additional dangerous sinks for analysis such as [44], thus increasing their attack surface coverage. Kang et al. [24] study prototype pollution on the client-side application by dynamic taint tracking. Their analysis is implemented at the V8 JavaScript engine by adapting the tool of Melicher et al. [32]. Their focus on client-side vulnerabilities is incompatible with server-side runtimes such as Node.js and Deno.

Other work [31, 47] uses concolic execution to find gadgets in client-side JavaScript code. Concolic execution is a promising enhancement of dynamic analysis. Liu et al. [31] focus specifically on finding gadget chains where one gadget unlocks the use of another gadget (e.g. by forcing a branch). It would be interesting to apply these ideas to backend systems.

Prototype pollution In recent years, we have seen increased

attention on prototype pollution vulnerabilities by both academia and practitioners [2,10,21,24,26,29,30,43,50]. The work of Arteau [10] is the first to demonstrate the feasibility of prototype pollution in a number of libraries. On the academic front, the vast majority of research contributions focus on the detection of prototype pollution [26, 29, 30]. These works use static taint analysis to find zero-day vulnerabilities leading to DoS attacks. Our contributions are complementary as they focus on the detection of universal gadgets rather than prototype pollution. The security impact of prototype pollution is discussed in practitioner forums [2,21,50]. Heyes [21] describes how prototype pollution can be exploited in Node.js to find vulnerabilities beyond DoS in black-box scenarios. Their semi-automated approach uses PP-finder [50] to report all undefined properties encountered during the execution and conducts manual inspection of packages for vulnerabilities. This approach is practical for a few specific targets, yet it is neither feasible at scale nor able to identify universal gadgets.

Code reuse attacks for the web Prototype pollution is a new class of code reuse vulnerabilities in web applications and, as such, it shares similarities with object injection vulnerabilities. Several works use static taint analysis to detect code reuse vulnerabilities for a variety of languages including PHP [15, 16, 18, 37], .NET [33, 42], and Java [22, 34]. Xiao et al. [49] study a related type of vulnerability coined hidden property attacks. Lekies et al. [27] and Roth et al. [38] study the implications of script gadgets in bypassing existing XSS and CSP mitigations. While all of these vulnerabilities rely on the reuse of code gadgets, their precise connection is yet to be studied systematically. GHUNTER implements a lightweight form of dynamic taint analysis at the level of JavaScript runtimes and V8 engine. Dynamic taint analysis [39,40] is a popular technique used to identify web-related vulnerabilities, including instrumentations at both program- and runtime-level [8, 13, 19, 23, 25, 28, 35, 41].

8 Conclusion

We have presented a semi-automated pipeline, GHUNTER, able to find exploitable universal gadgets in Node.js and Deno by lightweight dynamic taint analysis. We have used GHUNTER in a comprehensive study of universal gadgets, finding a total 123 exploitable gadgets. In absence of comprehensive defenses, we have systematized existing mitigation for prototype pollution and gadgets in the form of guidelines. We have used these guidelines in a study of existing exploits in real applications to illuminate the current status, finding a high-severity exploit due to the lack of principled mitigations.

Acknowledgments We thank anonymous reviewers for the helpful suggestions and feedback. This work was partially supported by the Swedish Foundation for Strategic Research (SSF) under project CHAINS, the Swedish Research Council (VR) under project WebInspector, and Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation under project ShiftLeft.

References

[1] Adding v8 fast api. https://github.com/nodejs/node/blob/v21.0.0/doc/contributing/adding-v8-fast-api.md.

[2] Client-Side Prototype Pollution and useful Script Gadgets. https://github.com/BlackFan/client-side-prototype-pollution.

[3] Deno, the next-generation JavaScript runtime. https://deno.com/.

[4] Node.js JavaScript runtime. https://nodejs.org/.

[5] Object - JavaScript - MDN. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object.

[6] Prototype pollution mitigation / symbol.proto. https://github.com/tc39/proposal-symbol-proto.

[7] Standard ecma-335 common language infrastructure (cli). https://www.ecma-international.org/publications/standards/Ecma-335.htm.

[8] Marco Abbadini, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, and Stefano Paraboschi. Cage4deno: A fine-grained sandbox for deno subprocesses. 2023.

[9] Mohammad M. Ahmadpanah, Daniel Hedin, Musard Balliu, Lars Eric Olsson, and Andrei Sabelfeld. SandTrap: Securing JavaScript-driven trigger-action platforms. In *USENIX Security Symposium*, 2021.

[10] Olivier Arteau. Prototype pollution attack in NodeJS application. *NorthSec*, 2018.

[11] Fraser Brown, Shravan Narayan, Riad S. Wahby, Dawson R. Engler, Ranjit Jhala, and Deian Stefan. Finding and preventing bugs in JavaScript bindings. In *Symposium on Security and Privacy (S&P)*, 2017.

[12] Mathias Bynens. Javascript engine fundamentals: Shapes and inline caches. https://mathiasbynens.be/notes/shapes-ics.

[13] Darion Cassel, Wai Tuck Wong, and Limin Jia. Nodemedic: End-to-end analysis of node.js vulnerabilities with provenance graphs. In *8th IEEE European Symposium on Security and Privacy, EuroS&P 2023, Delft, Netherlands, July 3-7, 2023*. IEEE, 2023.

[14] Eric Cornelissen, Mikhail Shcherbakov, and Musard Balliu. Ghunter: Universal prototype pollution gadgets in javascript runtimes. https://github.com/KTH-LangSec/ghunter.

[15] Johannes Dahse and Thorsten Holz. Static detection of second-order vulnerabilities in web applications. In *USENIX Security 14*, 2014.

[16] Johannes Dahse, Nikolai Krein, and Thorsten Holz. Code reuse attacks in PHP: automated POP chain generation. In *Conference on Computer and Communications Security (CCS)*, 2014.

[17] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. Towards measuring supply chain attacks on package managers for interpreted languages. In *Network and Distributed System Security Symposium (NDSS)*, 2021.

[18] Stefan Esser. Utilizing Code Reuse/ROP in PHP Application Exploits. *Proceedings of the Black Hat USA*, 2010.

[19] François Gauthier, Behnaz Hassanshahi, and Alexander Jordan. AFFOGATO: runtime detection of injection attacks for node.js. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2018.

[20] Language-Based Security group at KTH Royal Institute of Technology. Server-side prototype pollution gadgets. https://github.com/KTH-LangSec/server-side-prototype-pollution, 2024.

[21] Gareth Heyes. Server-side prototype pollution: Black-box detection without the dos. https://portswigger.net/research/server-side-prototype-pollution.

[22] Philipp Holzinger, Stefan Triller, Alexandre Bartel, and Eric Bodden. An in-depth study of more than ten years of java exploitation. In *Conference on Computer and Communications Security (CCS)*, 2016.

[23] Jordan Jueckstock and Alexandros Kapravelos. VisibleV8: In-browser Monitoring of JavaScript in the Wild. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, October 2019.

[24] Zifeng Kang, Song Li, and Yinzhi Cao. Probe the proto: Measuring client-side prototype pollution vulnerabilities of one million real-world websites. In *Network and Distributed System Security Symposium (NDSS 2022)*, 2022.

[25] Rezwana Karim, Frank Tip, Alena Sochůrková, and Koushik Sen. Platform-independent dynamic taint analysis for javascript. *IEEE Transactions on Software Engineering*, 46(12), 2020.

[26] 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*, 2021.

[27] Sebastian Lekies, Krzysztof Kotowicz, Samuel Groß, Eduardo A. Vela Nava, and Martin Johns. Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets. In *Conference on Computer and Communications Security (CCS)*, 2017.

[28] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: large-scale detection of DOM-based XSS. In *Conference on Computer and Communications Security (CCS)*, 2013.

[29] 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*, ESEC/FSE 2021, 2021.

[30] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. Mining Node.js vulnerabilities via object dependence graph and query. In *USENIX Security Symposium*, 2022.

[31] 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)*. IEEE Computer Society, 2024.

[32] William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. Riding out DOMsday: Toward detecting and preventing DOM cross-site scripting. In *NDSS 2018*, 2018.

[33] Alvaro Muñoz and Oleksandr Mirosh. Friday the 13th json attacks. *Proceedings of the Black Hat USA*, 2017.

[34] Alvaro Muñoz and Christian Schneider. Serial killer: Silently pwning your java endpoints, 2018.

[35] Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. Nodest: feedback-driven static analysis of node.js applications. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, (FSE)*, 2019.

[36] OASIS. Static analysis results interchange format (sarif) version 2.1.0. https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html.

[37] Sunnyeo Park, Daejun Kim, Suman Jana, and Sooel Son. FUGIO: automatic exploit generation for PHP object injection vulnerabilities. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*.

[38] Sebastian Roth, Michael Backes, and Ben Stock. Assessing the impact of script gadgets on CSP at scale. In *Asia Conference on Computer and Communications Security, (ASIA CCS)*, 2020.

[39] D. Schoepe, M. Balliu, B. C. Pierce, and A. Sabelfeld. Explicit secrecy: A policy for taint tracking. In *EuroS&P*, 2016.

[40] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE S&P*, 2010.

[41] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2013.

[42] Mikhail Shcherbakov and Musard Balliu. SerialDetector: Principled and Practical Exploration of Object Injection Vulnerabilities for the Web. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*, 2021.

[43] 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 2023, Anaheim, CA, USA, August 9-11, 2023*. USENIX Association, 2023.

[44] 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 on Web Conference 2024*, WWW '24, New York, NY, USA, 2024. Association for Computing Machinery.

[45] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. SYNODE: understanding and automatically preventing injection attacks on Node.js. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[46] Cristian-Alexandru Staicu, Daniel Schoepe, Musard Balliu, Michael Pradel, and Andrei Sabelfeld. An empirical study of information flows in real-world JavaScript. In *14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security, PLAS*, 2019.

[47] Marius Steffens. Understanding emerging client-side web vulnerabilities using dynamic program analysis. 2021.

[48] Ben Stock, Martin Johns, Marius Steffens, and Michael Backes. How the web tangled itself: Uncovering the history of client-side web (in)security. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. USENIX Association, 2017.

[49] Feng Xiao, Jianwei Huang, Yichang Xiong, Guangliang Yang, Hong Hu, Guofei Gu, and Wenke Lee. Abusing hidden properties to attack the Node.js ecosystem. In *USENIX Security Symposium*, 2021.

[50] YesWeHack. Server side prototype pollution, how to detect and exploit. https://blog.yeswehack.com/talent-development/server-side-prototype-pollution-how-to-detect-and-exploit/.

[51] Markus Zimmermann, Cristian-Alexandru, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *USENIX Security Symposium*, 2019.

A Appendix

```
let __pollutedValue = '0xEFFACED',
    __accessIndex = 0;
Object.defineProperty(Object.prototype, '${prop}', {
 get: function() {
  const returnValue = __pollutedValue +
      __accessIndex;
  __accessIndex += 1;
  try {
   throw new Error();
  } catch(error) {
   globalThis.log(returnValue + ' source stack:
       ' + error.stack);
  }
  return returnValue;
 },
 set: function(newValue) {
  Object.defineProperty(this, '${prop}', {
   value: newValue,
   writable: true,
   enumerable: true,
   configurable: true
  });
 },
 enumerable: ${prop === FORIN_SYMBOL ? "true" :
     "false"},
 configurable: true,
});
```

Listing 9: Injected snippet for polluting with a string value.

Gadget	Properties	Attack Type
cluster.fork	NODE_OPTIONS	ACE
cp.exec	NODE_OPTIONS	ACE
cp.execFile	NODE_OPTIONS	ACE
cp.execFileSync	shell, NODE_OPTIONS	ACE
	shell, input	ACE
	uid	PE
	gid	PE
	cwd	PT
cp.execSync	NODE_OPTIONS	ACE
	input	ACE
cp.fork	NODE_OPTIONS	ACE
cp.spawn	shell, NODE_OPTIONS	ACE
	uid	PE
	gid	PE
	cwd	PT
cp.spawnSync	shell, NODE_OPTIONS	ACE
	shell, input	ACE
	uid	PE
	gid	PE
	cwd	PT
crypto.privateEncrypt	padding	CD
crypto.publicEncrypt	padding	CD
crypto.subtle.encrypt	kty	Segfault
crypto.publicKey.export	kty	Segfault
crypto.privateKey.export	kty	Segfault
crypto.createPrivateKey	type	Segfault
	passphrase	Segfault
crypto.createPublicKey	type	Segfault
	passphrase	Segfault
fetch	socketPath, body, method, referrer	SSRF
fs.createWriteStream	mode	PE
https.get	hostname, headers, method, path, port, NODE_TLS_REJEC...	SSRF
https.request	hostname, headers, method, path, port, NODE_TLS_REJEC...	SSRF
	0	Segfault
http.get	hostname, headers, method, path, port	SSRF
http.request	hostname, headers, method, path, port	SSRF
http.Server.listen	backlog	Segfault
import	source	ACE
require (v18.13.0)	main	ACE
Socket.send	address	SSRF
stream.Duplex	readableObjectMode	Segfault
tls.TLSSocket.connect	path	Segfault
vm.SyntheticModule	sourceText, lineOffset, columnOffset	ACE
zlib.createGzip().write	writableObjectMode	Segfault

Table 5: A summary of the exploitable first-order gadgets in **Node.js**. *Gadget* identifies the public API that triggers a gadget; *Properties* specifies which properties must be polluted; *Attack Type* specifies one of Arbitrary Code/-Command Execution (ACE), Cryptographic Downgrade (CD), Path Traversal (PT), Privilege Escalation (PE), Server Side Request Forgery (SSRF), or Segfault.

Gadget	Properties	Attack Type
fetch	body, headers, method, 0	SSRF
Worker	env	PE
	ffi	PE
	hrtime	PE
	net	PE
	read	PE
	run	PE
	sys	PE
	write	PE
Deno.makeTempDir	dir	PT
	prefix	PT
Deno.makeTempDirSync	dir	PT
	prefix	PT
Deno.makeTempFile	dir	PT
	prefix	PT
Deno.makeTempFileSync	dir	PT
	prefix	PT
Deno.mkdir	mode	PE
Deno.mkdirSync	mode	PE
Deno.open	append	UM
	mode	PE
	truncate	UM
Deno.openSync	append	UM
	mode	PE
	truncate	UM
Deno.writeFile	append	UM
	mode	PE
Deno.writeFileSync	append	UM
	mode	PE
Deno.writeTextFile	append	UM
	mode	PE
Deno.writeTextFileSync	append	UM
	mode	PE
Deno.run	cwd	PT
	gid	PE
	uid	PE
Deno.Command	cwd	PT
	gid	PE
	uid	PE
cp.exec	shell, env	ACE
cp.execFileSync	shell, env	ACE
cp.execSync	shell, env	ACE
cp.spawn	shell, env	ACE
	gid	PE
	uid	PE
cp.spawnSync	shell, env	ACE
fs.appendFile	length	Loop
	offset	OOM
fs.writeFile	length	Loop
	offset	OOM
http.request	hostname, method, path, port	SSRF
https.request	hostname, method, path, port	SSRF
zlib.createBrotliCompress	params	Panic
json.JsonStringifyStream	prefix	UM
	suffix	UM
log.FileHandler	formatter	LP
tar.Tar.append	gid	PE
	uid	PE
yaml.stringify	indent	OOM

Table 6: A summary of the exploitable first-order gadgets in **Deno**. *Gadget* identifies the public API that triggers a gadget; *Properties* specifies which properties must be polluted; *Attack Type* specifies one of Arbitrary Code/-Command Execution (ACE), Log Pollution (LP), Loop, Out of Memory (OOM), Panic, Path Traversal (PT), Privilege Escalation (PE), Server Side Request Forgery (SSRF), or Unauthorized Modifications (UM).

USENIX Security '24 Artifact Appendix: GHUNTER: Universal Prototype Pollution Gadgets in JavaScript Runtimes

Eric Cornelissen
KTH Royal Institute of Technology

Mikhail Shcherbakov
KTH Royal Institute of Technology

Musard Balliu
KTH Royal Institute of Technology

A Artifact Appendix

A.1 Abstract

The artifacts develop lightweight taint analysis on top of the JavaScript runtimes Node.js and Deno with the goal of identifying prototype pollution gadgets. In particular, each artifact modifies the V8 JavaScript engine shared by Node.js and Deno as well as some minor aspects of each runtime itself; these changes are present as `.patch` files in the artifact. Additionally, each builds on top of the project with tooling to run our analysis and generate results. Finally, the last artifact constitutes modifications to Silent Spring used for the comparison between GHUNTER and Silent Spring.

We demonstrate the functionality and reproducibility of the analysis artifacts and evaluate the effectiveness of our analysis against Silent Spring in terms of precision and recall. The results of the former refer to Section 5.1 and Table 1, 5 and 6 while the latter refers to Section 5.2 and Table 2 and 3.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

There are no risks for the users relating to security and privacy of their machines. The artifact has been used to detect gadgets in production-ready software and these vulnerabilities have been responsibly disclosed to the vendors.

A.2.2 How to access

The artifacts are accessible on GitHub at github.com/KTH-LangSec/ghunter/tree/23abc11 which encompasses three sub projects: the Deno analysis artifact, the Node.js analysis artifact, and the Silent Spring comparison artifact.

A.2.3 Hardware dependencies

We performed the experiments described in this appendix on an AMD Ryzen 7 3700x 8-core CPU (3.60GHz) with 32 GB RAM and 50 GB of disk space. No specific hardware features are required for the artifact evaluation.

A.2.4 Software dependencies

We performed the experiments on the Ubuntu 22.04 OS. We used Docker as an OCI container runtime.

A.2.5 Benchmarks

Deno v1.37.2 We run our gadget detection analysis against Deno version 1.37.2. The source code of this benchmark is incorporated as git submodules in the `ghunter4deno` sub project (named `deno`, `deno_core`, and `rusty_v8`). Section 5.1 and Table 1 of the paper reports the aggregate number of gadgets detected and Table 6 of the paper reports all the detected first-order gadgets in detail.

Deno standard library v0.204.0 In addition to Deno v1.37.2, we run our gadget detection analysis against the Deno standard library version 0.204.0. The source code of this benchmark is incorporated as a git submodule in the `ghunter4deno` sub project (named `deno_std`). Section 5.1 and Table 1 and 6 also report on this benchmark.

Node.js v21.0.0 We run our gadget detection analysis against Node.js version 21.0.0. The source code of this benchmark is incorporated as a git submodule in the `ghunter4node` sub project (named `node`). Section 5.1 and Table 1 of the paper reports the aggregate number of gadgets detected and Table 5 of the paper reports all the detected first-order gadgets in detail.
Additionally, we run our gadget detection analysis against Node.js version 21.0.0 for a comparison to Silent Spring. The test cases for this comparison are located `src/ss21`. Section 5.2 and Table 3 of the paper reports on the results of this analysis.

Node.js v16.13.1 We run our gadget detection analysis against Node.js version 16.13.1 for a comparison to Silent Spring. The test cases for this comparison are located `src/ss16`. Section 5.2 and Table 2 of the paper reports on the results of this analysis.

Silent Spring We compare our results against those of Silent Spring. We do this on both Node.js v16.13.1 and v21.0.0. Our adaptation of Silent Spring is located in

the `silentspring4ghunter` sub project. This benchmark re-embeds the respective Node.js benchmarks on separate commits (`a6ae944` and `14966b5` resp.). Section 5.2 and Table 2 and 3 (resp.) of the paper report on the results of this analysis.

A.3 Set-up

We provide two modes for testing the Deno and Node.js artifacts (1) a prepared OCI container and (2) instructions on how to set up the environment from scratch. We only provide instructions on how to set up the environment from scratch for the Silent Spring artifact.

(S1): Deno. For the analysis of Deno use either the OCI container image `ghcr.io/kth-langsec/ghunter4deno`[1] by pulling it, launching it, and attaching a shell. Alternatively, build the container image by following the instructions from the README of github.com/KTH-LangSec/ghunter4deno at commit `63a9faa`. In this mode, the users may skip the rest of (**S1**) and (**I1**).
For a local set-up, clone github.com/KTH-LangSec/ghunter4deno with submodules recursively and checkout commit `63a9faa`. Then continue with (**I1**).

(S2): Node.js. For the analysis of Node.js use either the OCI container image by pulling `ghcr.io/kth-langsec/ghunter4node`[2], launching it, and attaching a shell. Alternatively, build the container image by following the instructions from the README of github.com/KTH-LangSec/ghunter4node at commit `86aad7c`. In this mode, the users may skip the rest of (**S2**) and (**I2**).
For a local set-up, clone github.com/KTH-LangSec/ghunter4node with submodules recursively and checkout commit `86aad7c`. Then continue with (**I2**).

(S3): Silent Spring. For the comparison to Silent Spring, clone github.com/KTH-LangSec/silentspring4ghunter with submodules recursively. For the comparison on Node.js v16.13.1 checkout commit a6ae944 and for the comparison on Node.js v21.0.0 checkout commit 14966b5. In either case, continue with (**I3**)-(**I5**).

A.3.1 Installation

(I1): Deno development prerequisites. See github.com/denoland/deno-docs commit `7b4aa84` file `building_from_source.md`.

(I2): Node.js development prerequisites. See github.com/nodejs/node commit `38d0e69` file `BUILDING.md`.

(I3): CodeQL v2.9.2. Download and unzip an asset for your platform of version 2.9.2 from the official repository. Add the path of the codeql folder to PATH environment variable.

(I4): Node.js v16.13.1. Follow the instructions on the official website to install Node.js version 16.13.1 for the comparison on this Node.js version.

(I5): Node.js v21.0.0. Follow the instructions on the official website to install Node.js version 21.0.0 for the comparison on this Node.js version.

A.3.2 Basic Test

(B1): Deno. We recommend running the source-to-sink analysis with a single test case as a basic test. First build using `./make.sh s2s sync`, then run the basic test using `./analyze.sh 2 20 basic-test`. The first command compiles Deno and can take up to an hour, the latter runs a simple analysis that should take about one minute. This is expected to yield about 6 gadget candidates.

(B2): Node.js. We recommend running the source-to-sink analysis with a single test case as a basic test. We provide a script to perform this test, `./nodejs-test-one.sh`. This will build Node.js for the analysis and run the analysis with a single test. This command compiles Node.js, which can take up to an hour, and runs a simple analysis that should take about one minute. This is expected to yield about 10 unique source-to-sink pairs after filtering.

(B3): Silent Spring. Follow the basic test instructions for the original Silent Spring artifact.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Our dynamic analysis tool applied to Deno uncovered 67 universal gadgets. This is evaluated by experiment (**E1**) and described in Section 5.1 and Table 6 of the paper.

(C2): Our dynamic analysis tool applied to Node.js uncovered 56 universal gadgets. This is evaluated by experiment (**E2**) and described in Section 5.1 and Table 5 of the paper.

(C3): Our dynamic analysis tool has higher precision and recall than Silent Spring for finding universal gadgets on two different Node.js versions. This is evaluated by experiment (**E3**)-(**E6**) and described in Section 5.2 of the paper.

A.4.2 Experiments

We describe a total of 6 experiments, 2 related to claims (**C1**) and (**C2**) and 4 related to (**C3**). The former cover the first 3 benchmarks and the latter cover the last 3 benchmarks.

[1] a4c29470545af82a0d8b446e1594ba4e78ad45babdf3af51c72f54fad1c35860
[2] a2b09930d54d652f192d086a91186d5d9d94c14f2deae451b88e563fcb38231a

(E1): Analysis of Deno, 10 human-minutes + <4 compute-hours + 50GB disk: Full analysis of the Deno runtime for universal gadgets.
Set-up: Follow (**S1**).
Preparation: Follow (**B1**).
Execution: Start `./run.sh`, optionally with a number of workers (default 5) and test timeout (default 20s) as `./run.sh <W> <T>`.
Results: The output at the end of the analysis provides a table of which expected gadget candidates from Table 6 of the paper were found as well as the analysis numbers from Section 5.1 (paragraph *Analysis of Deno*); This output can be recomputed by running the `numbers.sh` script *after* the analysis has finished. The exact numbers may differ but are expected to be similar. The folder `_aggregate` will contain the final two SARIF files for manual analysis, which can be compared to the SARIF files in the `results` directory.

(E2): Analysis of Node.js, 10 human-minutes + <4 compute-hours + 50GB disk: Partial analysis of the Node.js runtime for universal gadgets in the `child_process` API.[3]
Set-up: Follow (**S2**).
Preparation: Follow (**B2**).
Execution: Start `./run-child_process-s2s.sh`, optionally with a number of workers (default 5) and test timeout (default 20s) as `./run-child_process-s2s.sh <W> <T>`, to perform the source-to-sink analysis.
After the script has finished, start `./run-child_process-crashes.sh`, optionally with a number of workers (default 5) and test timeout (default 20s) as `./run-child_process-crashes.sh <W> <T>`, to perform the unexpected-termination analysis.
Results: The output at the end of the former script constitutes part of the aggregate analysis numbers from Section 5.1 of the paper except limited to the `child_process` API (expect ~70,000 sinks reached with ~1,500 unique sink-source pairs before filtering, and ~40 unique sink-source pairs after filtering). It produces the SARIF files for manual review in a folder named `node/fuzzing/X-YYYY-MM-DD-HH-MM-SS`.
The output at the end of the latter script constitutes the remaining part of the aggregate analysis numbers from Section 5.1 of the paper except limited to the `child_process` API (expect 2 gadget candidates out of ~111,000 crashes).

(E3): Comparison on Node.js v21.0.0 GHUNTER, 10 human-minutes + <2 compute-hour + 50GB disk: The GHUNTER part of the comparison between GHUNTER and Silent Spring on Node.js v21.0.0.
Set-up: Follow (**S2**).
Preparation: Not applicable.
Execution: Start `./run-compare-ss-21.sh` to run the source-to-sink analysis for the relevant APIs for the comparison.
Results: This will output the results and also store them in the `node/fuzzing.ss21` folder. There will be 9 folders following the `X-YYYY-MM-DD-HH-MM-SS` naming scheme. Each maps to a row from Table 3 of the paper according to the mapping found in the project's README and contains two relevant files: `count.txt` for the number presented as "GC" in Table 3 and `compare.json` with the properties and corresponding sinks of each gadget candidate (validating them is a manual process). False negatives are derived as $FN = GT - TP$.

(E4): Comparison on Node.js v21.0.0 Silent Spring, 10 human-minutes + <5 compute-hours + 2GB disk: The Silent Spring part of the comparison between GHUNTER and Silent Spring on Node.js v21.0.0.
Set-up: Follow (**S3**) and checkout `14966b5`.
Preparation: Run `node --version` and ensure you are using v21.0.0.
Execution: Start `./compare.sh`.
Results: The script writes the raw results for the comparison as a folder per row of Table 3 in the paper in the `raw-data` folder. Each folder contains the raw output from Silent Spring as well as a `ghunter.log` file with the data for comparison. In particular, the last line (starting with `Candidates`) is the "GC" number from Table 3 and the data preceding it (starting from `all props`) contains the true and false positives data (validating them is a manual process). False negatives are derived as $FN = GT - TP$.

(E3): Comparison on Node.js v16.13.1 GHUNTER, 10 human-minutes + <2 compute-hour + 50GB disk: The GHUNTER part of the comparison between GHUNTER and Silent Spring on Node.js v16.13.1.
Set-up: Follow (**S2**).
Preparation: Not applicable.
Execution: Start `./run-compare-ss-16.sh` to run the source-to-sink analysis for the relevant APIs for the comparison.
Results: This will output the results and also store them in the `node/fuzzing.ss16` folder. There will be 11 folders following the `X-YYYY-MM-DD-HH-MM-SS` naming scheme. Each maps to a row from Table 2 of the paper according to the mapping found in the project's README and contains two relevant files: `count.txt` for the number presented as "GC" in Table 2 and `compare.json` with the properties and corresponding sinks of each gadget candidate (validating them is a manual process). False negatives are derived as $FN = GT - TP$.

[3] The full analysis can be performed by substituting "child_process" for "all" in the experiment steps, but this requires hardware similar to that described in the paper rather than hardware similar to that described in this appendix and is expected to take over 96 hours to complete.

(E6): Comparison on Node.js v16.13.1 Silent Spring, 10 human-minutes + <6 compute-hours + 2GB disk: The Silent Spring part of the comparison between GHUNTER and Silent Spring on Node.js v16.13.1.
Set-up: Follow (**S3**) and checkout `a6ae944`.
Preparation: Run `node --version` and ensure you are using v16.13.1.
Execution: Start `./compare.sh`.
Results: The script writes the raw results for the comparison as a folder per row of Table 2 in the paper in the `raw-data` folder. Each folder contains the raw output from Silent Spring as well as a `ghunter.log` file with the data for comparison. In particular, the last line (starting with `Candidates`) is the "GC" number from Table 2 and the data preceding it (starting from `all props`) contains the true and false positives data (validating them is a manual process). False negatives are derived as $FN = GT - TP$.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.