Silent Spring: Prototype Pollution Leads to Remote Code Execution in Node.js

Mikhail Shcherbakov and Musard Balliu, *KTH Royal Institute of Technology;*
Cristian-Alexandru Staicu, *CISPA Helmholtz Center for Information Security*

https://www.usenix.org/conference/usenixsecurity23/presentation/shcherbakov

This paper is included in the Proceedings of the
32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Silent Spring: Prototype Pollution Leads to Remote Code Execution in Node.js

Mikhail Shcherbakov
KTH Royal Institute of Technology

Musard Balliu
KTH Royal Institute of Technology

Cristian-Alexandru Staicu
CISPA Helmholtz Center for Information Security

Abstract

Prototype pollution is a dangerous vulnerability affecting prototype-based languages like JavaScript and the Node.js platform. It refers to the ability of an attacker to inject properties into an object's root prototype at runtime and subsequently trigger the execution of legitimate code gadgets that access these properties on the object's prototype, leading to attacks such as Denial of Service (DoS), privilege escalation, and Remote Code Execution (RCE). While there is anecdotal evidence that prototype pollution leads to RCE, current research does not tackle the challenge of gadget detection, thus only showing feasibility of DoS attacks, mainly against Node.js libraries.

In this paper, we set out to study the problem in a holistic way, from the detection of prototype pollution to detection of gadgets, with the ambitious goal of finding end-to-end exploits beyond DoS, in full-fledged Node.js applications. We build the first multi-staged framework that uses *multi-label* static taint analysis to identify prototype pollution in Node.js libraries and applications, as well as a hybrid approach to detect *universal gadgets*, notably, by analyzing the Node.js source code. We implement our framework on top of GitHub's static analysis framework CodeQL to find 11 universal gadgets in core Node.js APIs, leading to code execution. Furthermore, we use our methodology in a study of 15 popular Node.js applications to identify prototype pollutions and gadgets. We manually exploit eight RCE vulnerabilities in three high-profile applications such as NPM CLI, Parse Server, and Rocket.Chat. Our results provide alarming evidence that prototype pollution in combination with powerful universal gadgets lead to RCE in Node.js.

1 Introduction

In recent years we have seen a growing interest in running JavaScript outside of the browser. A prime example is Node.js, a popular server-side runtime that enables the creation of full-stack web applications. Its package management system, NPM, is the world's largest software repository with millions of packages. Researchers have studied this ecosystem extensively to discover several security risks [14, 20, 31, 44–47, 51], showing that these risks are further exacerbated by the interconnected nature of the ecosystem [52]. While most prior work focuses on libraries, the problem of automatically detecting vulnerabilities in Node.js applications is still open.

Prototype pollution is a JavaScript-driven vulnerability that manifests itself powerfully in the Node.js ecosystem. The vulnerability is rooted in the permissive nature of the language, which allows the mutation of an important built-in object in the global scope – `Object.prototype` – called the root prototype. JavaScript's prototype-based inheritance enables accessing this important object through the prototype chain. Thus, attackers can instruct vulnerable code to mutate the root prototype by providing well-crafted property names to be accessed at runtime. As a consequence, every object that inherits from the root prototype, i.e., the vast majority of objects in the runtime, inherits the mutation on the root prototype, e.g, an attacker-controlled property. This vulnerability was first introduced by Arteau [12], showing that it is a widespread problem in Node.js libraries. Recently, Li et al. [31,32] explore static analysis to detect prototype pollution vulnerabilities using object property graphs.

The few prior works [25, 27, 31, 32, 51] on prototype pollution consider a successful attack any mutation of the root prototype. An immediate consequence of such mutations is Denial of Service (DoS) due to the overwriting of important built-in APIs, e.g., `toString`. By contrast, our work studies the implications of prototype pollution beyond DoS. In particular, we propose a semi-automated approach for detecting Remote Code Execution (RCE) vulnerabilities pertaining to prototype pollution. While there is anecdotal evidence about the possibility of such attacks [5, 12], we are the first to propose a principled and systematic approach to detect them. Our key focus is on gadget identification and end-to-end exploitation which no prior work has addressed thoroughly.

Moreover, we note the important similarities between object injection vulnerabilities (OIVs) [17, 41] and RCEs based

---

on prototype pollution. These attacks work in two stages: (1) there is an untrusted flow from an application's untrusted entry points to an *injection sink*, e.g., the property of an object; (2) there is a gadget that further propagates the attacker-controlled data from the injection sink to a security-relevant *attack sink*. In analogy, the attacker loads the gun in stage one (by placing the payload into the injection sink), while letting someone else (a gadget) pull the trigger in stage two and carry out the attack (through an attack sink). We propose calling the class of OIVs pertaining to prototype pollution, *prototype-based object injection* vulnerabilities (POIV).

In statically-typed languages, OIVs are enabled by insecure deserialization, which allows instantiating objects of an unexpected type, thus triggering otherwise unused methods. Similarly, in a duck-typed language like JavaScript, if an attacker mutates the root prototype, they change the dynamic type of multiple objects in the runtime. This in turn activates otherwise unused code paths that correspond to the new type, e.g., object `foo` having a property `bar` defined. Thus, code reuse is done at a finer granularity and in a less localized manner in dynamically typed languages. We also note that attackers can mutate several properties at once, hence chaining gadgets in the fashion of property-oriented programming [17].

Our technical contribution is a multi-staged framework that uses multi-label static analysis for detecting prototype pollution, and a hybrid solution, i.e., combining dynamic and static analysis, for detecting gadgets. We observe that code patterns that lead to prototype pollution, i.e., injection sinks, are rather rare in real-world code. Thus, different from prior work, we propose tuning the static analysis for improved recall, rather than precision. Additionally, to emphasize the feasibility of the attack, we detect *universal gadgets* in Node.js' source code, which can be exploited in a wide-range of applications as they come packaged with the Node.js runtime.

Drawing on security advisories [10], we aggregate a set of 100 vulnerable packages, which we use to design and validate our pollution detection analysis. In comparison with the state-of-the-art tool ODGen [32], we empirically show that one can significantly increase recall and scalability, while only paying a modest decrease in precision.

We then design and evaluate our novel gadget detection analysis against four widely-used APIs for handling code or command execution in Node.js. We find a total of 11 gadgets that can be triggered during typical execution of these APIs. While some gadgets enable code injection directly, others allow attackers to load arbitrary files from the disk into the runtime, by confusing the module resolution mechanism. We also conduct a quantitative study on packages to estimate the prevalence of these gadgets in the Node.js ecosystem. We believe that we are the first to show evidence that control flow can be hijacked in this way in Node.js, further emphasizing the dangers of shipping unused code with applications [28].

Finally, we analyze 15 popular Node.js applications, reporting on the effort to finding RCE with our methodology.

We identify eight exploitable RCE vulnerabilities in highly-popular applications such as NPM CLI, Parse Server and Rocket.Chat. We have responsibly disclosed these critical vulnerabilities to developers and they are now fixed, acknowledging our contributions with a high-severity advisory (e.g., CVE-2022-24760) and bug bounties.

Contrary to established recommendations, this work embraces false positives. We show that a motivated attacker can sieve through the manageable amount of false positives to find critical zero-day exploits against well-tested, mature applications. We believe that vulnerability detection tools tuned for offensive security can afford this luxury due to the high return on investment provided by a single true positive.

In summary, the paper offers the following contributions:

- We are the first to study the impact of prototype pollution vulnerabilities in full-fledged Node.js applications, beyond denial-of-service attacks.
- We present a principled approach for detecting RCE vulnerabilities that are enabled by prototype pollution.
- We show that our pipeline is directly applicable to real-world code: we find 11 universal gadgets in Node.js' source code and eight RCEs in popular applications.
- We provide initial evidence that unused code shipped with the application, e.g., third-party dependencies, can be leveraged as part of code reuse attacks in Node.js.

2 Context and Technical Background

This section provides background information and discusses the targeted threat model.

2.1 Prototype-based OIV

Prototypes are a key feature to implement inheritance of JavaScript properties and methods to form a *prototype chain*. When creating an empty object, e.g., `const obj = {}`, it already contains many built-in properties and functions, for instance, the `toString` function. When invoking `toString` on an object, the runtime engine will first check if the function is explicitly defined for the given object. If not, it will recursively look for its definition on the object's prototype chain. Unfortunately, most objects share the same root prototype. For example, all objects created via the literal `{}` or the `new Object()` constructor share the same prototype unless it is explicitly overridden. The following code snippet illustrates the problem:

```
1  const o1 = {};
2  const o2 = new Object();
3  o1.__proto__.x = 42;
4  console.log(o2.x);
```

Although objects `o1` and `o2` are unrelated, their prototype properties `__proto__` point to the same object. In fact, if we add the new property `x` to the prototype of object `o1` it will

also affect object `o2`, resulting in a print of value `42` to the console. Therefore, if we modify the root prototype shared by different objects, all these objects will reflect the modification.

We now explain the two stages needed to carry out a prototype-based attack that leads to code execution.

Stage 1: Polluting the prototype. Listing 1 shows a contrived example to illustrate key ingredients defining an *injection sink* in a POIV. We assume that the attacker controls all three arguments of function `entryPoint`. The first ingredient is an object that inherits a prototype that the attacker wants to pollute, as shown by the object in line 2, which inherits `Object.prototype`.

```
1  function entryPoint(arg1, arg2, arg3) {
2    const obj = {};
3    const p = obj[arg1];
4    p[arg2] = arg3;
5    return p; }
```

Listing 1: Prototype pollution example

The second ingredient is the attacker-controlled access to the prototype property, as shown in line 3 via the bracket notation. The attacker can pass `__proto__` to `arg1` to store `Object.prototype` in variable `p`. The last two ingredients require creating a target property in the prototype and assigning an attacker-controlled value. In fact, line 4 assigns an attacker-controlled value to a property of `Object.prototype`. Since the attacker controls `arg2` and `arg3`, they can write any value to any property. The JavaScript engine will create a new property, if such property does not exist. In general cases, the attacker cannot fully control all the ingredients, e.g., the property in `arg2` or the value in `arg3`.

An immediate effect of this vulnerable pattern is the attacker's ability to perform a DoS attack, e.g, by executing the function `entryPoint("__proto__", "toString", 1);` to alter the state to an unexpected integer value, i.e., `Object.prototype.toString = 1`, thus, forcing an application that calls `toString()` to crash.

Stage 2: Executing the gadget. This stage requires identifying gadgets that contain insecure flows from injection sinks to *attack sinks* that perform security-sensitive actions.

```
1  const { execSync } = require("child_process");
2  function gadget(args, options) {
3    const cmd = options.cmd || "cmd.exe /k";
4    return execSync('${cmd} ${args}');
5  }
6  const args = ...;
7  gadget(args, {});
```

Listing 2: Gadget example

Consider the benign example in Listing 2, where a list of arguments `args` and a command object `options` is passed to a function `gadget` with the intention to execute command `options` with arguments `args`. The intended use of function `gadget` is to either execute the command that is specified via the property `cmd` of the `options` object or execute the default command `cmd.exe`. However, since the developer passed an empty object to function `gadget` (line 7), the program is expected to execute the default command, because `options.cmd` is undefined (line 3).

Consider now an execution of the program in Listing 1 such that `entryPoint("__proto__", "cmd", "calc.exe&");` The attacker manipulates the `cmd` property of the root prototype, causing the undefined property `options.cmd` to fall back to the value in the prototype chain. Hence, the attacker can control the command passed to `execSync`, which leads to code execution, launching a calculator via `calc.exe&`.

2.2 Threat Model

Our threat model targets an attacker that controls the *untrusted entry points* of a Node.js application with the goal of exploiting prototype-based OIVs to perform arbitrary code execution on the application. These untrusted entry points are application-specific, however, candidates include HTTP connections, untrusted database reads, and the like. We also consider a weaker threat model targeting only *universal* gadgets that occur in the source code of Node.js. Because these gadgets appear in code that executes with the Node.js runtime, they are available for exploitation in any Node.js application. For this threat model, we assume that the attacker controls the injection sinks pertaining to the execution of a gadget.

3 Overview

This section provides an overview of our multi-staged analysis framework, illuminating on the key challenges in detecting and exploiting prototype-based object injection vulnerabilities. We use our newly-detected vulnerability in NPM CLI to illustrate the complexity and challenges of such an endeavor. NPM CLI [9] is the command line client that allows developers to install and publish packages in NPM registries. It comes bundled with the Node.js runtime and consists of 713,648 lines of code.

Detecting prototype pollution. Figure 1 shows the simplified code fragment of the function `diffApply` from NPM CLI's codebase, which is subject to prototype pollution.

The function takes the array `path` from the attacker-controlled parameter `diff` and calls the built-in function `shift()` that returns the first element of the array. The data flow then goes through the loop storing a property value to the variable `obj` (red line). Because the attacker indirectly controls the property name `thisProp` in line 8, the property read allows them to access the object's root prototype by setting `thisProp` to `__proto__`. Subsequently, the attacker can assign any value to any property of the root prototype as illustrated by the assignment in line 11. As a result, the attacker has full control of the injection sink denoted by the blue dotted lines. For instance, the function call

```
1  function diffApply(obj, diff) {
2    var lastProp = diff.path.pop();
3    var thisProp;
4    while ((thisProp = diff.path.shift()) != null) {
5      if (!(thisProp in obj)) {
6        obj[thisProp] = {};
7      }
8      obj = obj[thisProp];
9    }
10   if (diff.op === REPLACE || diff.op === ADD) {
11     obj[lastProp] = diff.value;
12   }
13 }
```

Figure 1: Injection sink in NPM CLI

```
1  const {ArrayPrototypePush} = primordials;
2  const {Process} = internalBinding('process_wrap');
3  function spawn(file, args, opts) {
4    opts = normalizeSpawnArgs(file, args, opts);
5    this._handle = new Process();
6    this._handle.spawn(opts);
7  }
8
9  function normalizeSpawnArgs(file, args, opts) {
10   let envKeys = [], envPairs = [];
11   const env = opts.env || process.env;
12   /* ... */
13   for (const key in env)
14     ArrayPrototypePush(envKeys, key);
15
16   for (const key of envKeys) {
17     const v = env[key];
18     ArrayPrototypePush(envPairs, `${key}=${v}`);
19   }
20
21   return { /* ... ,*/ envPairs /*, ... */ };
22 }
```

Figure 2: Universal gadget in Node.js standard library

diffApply({}, {path: ['__proto__', 'env'], value: 'payload', op: ADD}) injects into Object.prototype the environment property env with payload payload.

This code fragment illustrates the challenges that a static analysis should overcome. First, in contrast to standard taint analysis, injection sinks cannot be identified syntactically as they require specialized data flow analysis that record accesses to object properties, as illustrated by the blue dotted line. The analysis should identify attacker-controlled inputs that allow to control the prototype object, followed by uses of this prototype object as a receiver in a property assignment [31]. Second, the analysis should handle language constructs such as loops and model the JavaScript built-in functions, e.g., shift() to correctly propagate data flows. Third, given the size of the targeted codebases, the analysis should be scalable, seeking the sweet spot between precision and recall. While prior work achieves high precision, it reports low recall, thus increasing the possibility to miss flaws in real applications [31, 32]. These requirements lead us to our first research question: *How to design and implement a scalable static analysis that effectively identifies prototype pollution in real-world libraries and applications?* To answer this question we develop a multi-label static taint analysis, which we discuss in Section 4.1 and evaluate in Section 6.1.

Detecting code gadgets. Recall that our threat model requires identifying code gadgets that read the attacker payloads from the injection sink and pass it into an attack sink. Figure 2 shows a universal gadget we identified, stemming from the popular spawn function of the Node.js standard library. This function first calls normalizeSpawnArgs and reads the value of property opts.env in line 11. This optional parameter contains key-value pairs of the environment variables of a new process. If a developer passes an object without property env, the JavaScript runtime will look up the property in the prototype chain. Alternatively, attacker can inject the environment variable directly using the for..in loop in line 13 to subsequently read it either from the opts.env or process.env object in line 11.

The reader may at this point wonder about our second research question: *How to identify universal properties reads such as* env? In fact, a prerequisite for gadget detection is to identify property reads that delegate the lookup of the property to the prototype chain, while filtering out cases where the property is defined in the object itself. This is a complicated task for a static analysis, hence we use dynamic analysis instead. We discuss the details in Section 4.2.

Further, the gadget contains intricate data flows from the property read in line 11 to the attack sink in line 6 as denoted by the red arrows. Specifically, the for..in loop enumerates the property names of the read object and passes them to an array through the ArrayPrototypePush call. This is an internal function that implements the semantics of Array.prototype.push and subsequently enumerates the envKeys array, storing key-value pairs by the template literal (line 18) and returning a new object with the property envPairs. Therefore, an analysis should model the semantics of internal functions, template literals, the for..in and for..of statements to propagate the attacked-controlled values properly. Moreover, function spawn (line 3) passes the modified object opts to method spawn of the internal wrapper Process (line 6) that is implemented in the C++ component of Node.js. This method corresponds to the actual attack sink. Specifically, if an attacker uses {GIT_SSH_COMMAND: 'calc&'} as payload for function diffApply, they can simply wait for an invocation of the attack sink spawn from the git command. The latter uses the specified command from the environment variable GIT_SSH_COMMAND when connecting to a remote system. This leads us to our third research question: *How to identify the attack sinks and data flows from universal property reads to these attack sinks in Node.js?* Gadget detection is a new challenge with no prior research, except for some evidence provided by the practitioners' community [5, 12]. To address this question, we develop a taint-based static analysis that tracks flows from property reads to attack sinks, which we discuss in Section 4.2 and evaluate in Section 6.2.3.

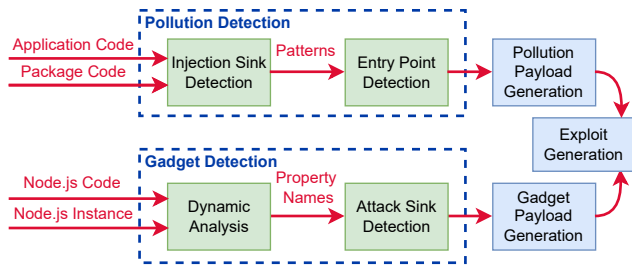**Putting it all together.** The presence of prototype pollution

Figure 3: High-level workflow: automated steps (green) and manual steps (blue).

and gadgets is not sufficient to carry out an end-to-end RCE attack. The attacker needs to identify application-specific untrusted entry points that enable the payload to reach the injection sinks, and to subsequently propagate this payload to an attack sink via the gadget. This step requires us to combine data flow analysis with the call flow analysis, starting at untrusted entry points, while driving the payload to reach an attack sink. This leads to our final research question: *How to identify public entry points and payloads to demonstrate the feasibility of RCE attacks?* We use a combination of manual and automated analysis to drive the exploit towards success, as detailed in Section 4.3 and evaluated in Section 6.3.

4 Methodology

We present a semi-automated analysis framework for detecting and exploiting prototype-based vulnerabilities. The framework is divided into three major steps: (*i*) automated prototype pollution detection; (*ii*) automated gadget detection; and (*iii*) manual exploit generation for end-to-end attacks. Figure 3 illustrates the sequence of steps and their dependencies.

The prototype pollution detection step takes as input the code of an application or NPM package and performs a *multi-label* taint-based static analysis. Subsequently, the analysis reconstructs the call graph of the application to find entry points that reach the prototype pollution, thus facilitating the task of identifying attacked-controlled entry points. The gadget detection step implements a hybrid solution. A dynamic analysis first detects which properties can be actually polluted by executing Node.js APIs of interest in a testing environment that logs property accesses, ultimately returning a list of accessed property names. These property names, together with the source code of Node.js, are used as input to our second static analysis to identify (universal) gadgets in Node.js. Each gadget includes an entry point that reaches a targeted property read and an attack sink that is called with values read from the target property. The last step of the approach is the end-to-end exploit generation. This is a manual step that requires an investigation of the target application's workflow to validate the exploitability of the detected prototype pollution and gadget to achieve code execution on the system.

4.1 Prototype Pollution Detection

Multi-label taint analysis. The detection of prototype pollution requires specialized data flow analysis that identifies injection sinks boiling down to the pattern `obj[prototype][property] = value`. We find these patterns by means of a flow- and context-sensitive *multi-label* taint analysis. Specifically, we use two labels *input* and *proto* to capture the temporal relationship between (attacker-controlled) property accesses in an object. We use label *input* to mark parameters that are directly controlled by the attacker and label *proto* to record that the attacker already controls the prototype of the labeled object.

The analysis works as follows: initially, it marks the parameters of the analyzed function with the *input* label. Then, it performs (standard) taint analysis propagating this label according to the JavaScript semantics until it reaches a property read with a tainted value in the property name, e.g., `obj[prototype]` with `prototype` having label *input*. This indicates that the attacker may control the property name and get access to the root prototype. At this point, the label of the resulting property read, e.g., `obj[prototype]`, is changed to the label *proto* to record that the attacker can now control the prototype. Subsequently, the analysis continues the taint propagation until it reaches a property assignment, e.g., `obj[prototype][property] = value`, where the object of the property assignment, i.e., `obj[prototype]`, is marked with the *proto* label, thus identifying the injection sink. We note that this a general characterization of injection sinks, where the attacker does not necessarily control the accessed property (`property`) and the assigned value (`value`), so long as they control the root prototype (`prototoype`). Because this setting is more difficult to exploit, our analysis supports a *priority* mode to identify attacker-controlled property names and values in a property assignment. Specifically, it performs two additional operations to check that the property read (`property`) and the value (`value`) are marked with label *input*, indicating that they may be controlled by the attacker. As expected, these priority injection sinks are an easier target for exploitation in practice.

Figure 1 illustrates the multi-label taint analysis for the prototype pollution vulnerability in NPM CLI. We consider the function `diffApply` as target function and mark the parameters with label *input*. The red arrows depict the propagation of label *input*. The parameter `diff` is an object and the taint analysis passes the tainted label to all its properties. The method `shift` is a built-in method that returns the first element of the array. The static analysis models JavaScript standard built-in objects, and thereby, propagates the *input* label to `thisProp` in line 4. The next node of the data flow is the property read in line 8, hence the analysis changes its label to *proto*. The blue dotted lines then visualize the *proto* label propagation. The tainted value reaches the property assignment, and the algorithm reports this expression as the injection sink. This

is also a priority sink because the parameters `lastProp` and `diff.value` in line 11 have label *input*.

Methodology We define the (attacker-controlled) target functions in two ways: (*i*) a package's exported functions (dubbed Exported Functions) or (*ii*) any function of the analyzed codebase (dubbed Any Functions). We use the first option for the package analysis only, assuming that the attacker controls any exported function and class of a package. The second option allows us to analyze real-world applications with no knowledge of the application's entry points, which usually depend on the specific threat model. We find this option useful in practice to overcome inherent limitations of static analysis for JavaScript, which does not always support the correct label propagation, e.g., due to callbacks or dynamically-generated code. In this case, the analysis allows us to detect injection sinks by propagating the *input* label from the nearest function on the call graph. Yet, the semantic modeling of built-ins is key to increasing the true positive rate.

Ideally, a taint analysis should provide precise and complete models of JavaScript constructs. CodeQL features many person-hour contributions into the modeling of built-in functions. Nonetheless, we observe that in practice these models are still insufficient. Our approach relies on the ground truth provided by known vulnerabilities to improve the tool in modeling features that pertain to these vulnerabilities, thus reducing the number of false negatives. Concretely, we review the CodeQL standard library to identify and fix language features, e.g., Arrays and reflection calls (see Section 5) that affect the taint semantics for the considered packages. We applied this process iteratively to achieve high recall.

Entry point detection. We propose a lightweight analysis to detect application-level entry points that may trigger the injection sinks. This helps with applications that receive tainted data from external storage to find the external action that triggers the data acquisition from the storage. The static analysis first reconstructs a call graph where the functions with no callers are represented by nodes with outgoing edges only. The algorithm considers such nodes as potential application entry points and reports the code paths to the injection sink.

Summary This step provides information about the pollution patterns and application's entry points for future manual validation and exploit generation. We contribute five analysis variants: one analysis for entry point detection; two *priority* analyses (for each type of target function) that report injection sinks with all tainted ingredients; and two *general* analyses (for each type of target function) that report injection sinks with a tainted receiver only.

4.2 Gadget Detection

Dynamic analysis. We first parse the Node.js' source code and syntactically extract all directly-accessed properties. The dynamic analysis defines a custom handler with a property getter in `Object.prototype` for each extracted property name. We systematically analyze the Node.js API documentation to identify functions that potentially run processes or evaluate arbitrary code in the runtime. We then invoke these APIs to log their attempt of property reads from `Object.prototype`, which result in reading uninitialized properties and getting the value `undefined`. This means that the values of these properties can be tampered via prototype pollution. The dynamic analysis passes the collected property names to the next step.

Static analysis. The analysis takes the Node.js' source code and the property names as input. The algorithm first performs the call flow analysis of Node.js API functions, including information about aliases, ultimately allowing us to reconstruct a precise call graph of the analyzed functions. We then use the call flow analysis to identify paths from any exported function to polluted property reads (identified by the dynamic analysis) and subsequently combine it with context-sensitive taint tracking to identify paths from these property reads to attack sinks, represented as tainted arguments to internal function calls. Specifically, the analysis propagates the taints on return values only for functions that are reached by the Node.js API on the analyzed call flow. Additionally, the analysis identifies affected exported functions that were not analyzed dynamically. For instance, the analysis of function `spawn` reports a possible pollution of property `env`. The static analysis shows the attack sinks that are affected by `env` include additional Node.js API functions such as `spawnSync`, `exec` and `fork`.

The taint analysis considers internal functions, i.e., functions for which the analyzer cannot resolve the function body, as candidate attack sinks. We conservatively cover all functions with no implementation in the codebase. The taint analysis also uses multi-labels. For property assignments, the algorithm propagates the taint label *polluted* of the property and applies the new label *receiver* to the receiver recursively. For instance, if `value` in the assignment `obj.prop = value` has label *polluted*, then the analysis applies the *receiver* label to `obj` and the *polluted* label to its property `prop`. This is needed because we cannot enumerate all properties of an object when this object is used as parameter to an attack sink. Finally, the static analysis reports internal functions with no arguments and either *polluted* or *receiver* labels as attack sinks.

Figure 2 shows the analysis in action for property `env`. The blue dotted arrows illustrate the call flow analysis from the exported function `spawn` to the first function call. The `normalizeSpawnArgs` contains the property read `env` which is the starting node of the taint analysis (red arrows). Initially, the taint analysis propagates the label *polluted* through the data flows. When the tainted value reaches the object creation statement in line 21, the analysis keeps the taint label for the property `envPairs` and assigns the label *receiver* to the created object. This object is further propagated to the caller function and passed to the internal function `_handler.spawn` in line 6, thus reporting `_handler.spawn` as a candidate sink.

4.3 Exploit Generation

Our approach relies on the human-in-the-loop model for exploit generation. For gadget exploits, the information about attack sinks allows us to evaluate the impact of a polluted property and filter out non-malicious sinks. The call flow and taint analysis help to explore the code slice that reaches the attack sink. We use this information to generate a payload and test it on the detected Node.js APIs. We validate the detected sinks and report new gadgets for Node.js in Section 6.2.3.

A security analyst first analyzes the prototype pollution patterns to filter out false positives and non-executable cases in the regular application workflow, e.g., patterns in testing code and development tools. For suspicious cases, the analyst uses the automatically-detected entry points to generate the first version of a payload and validates it on the application. If an exploit fails, the analyst investigates the cause using other tools (e.g., a debugger) and modifies the payload.

If the validation of the prototype pollution succeeds, then the next step is to search for gadget triggers. We extend the universal gadget entry points (e.g., `spawn`) with functions that evaluate JavaScript code represented as strings (`eval()`, `new Function()`, `new vm.Script`) and provide a call graph analysis for these calls. The analyst may use the call graph analysis to detect calls to these functions as well as the application's entry points that reach these calls.

If the analyst detects a gadget trigger, they need to validate that it is executed after the injection sink and then generate a payload that pollutes the required properties. If code evaluation function is detected, the analyst investigates the preconditions for invoking it with attacker-controlled data. The input data can be read from the polluted property, or the function's execution may be dependent on specific conditionals that use the polluted property. These steps lead to arbitrary code execution inside the Node.js instance. We estimate the effort of using such exploitation model in a study in Section 6.3.

5 Implementation

CodeQL [4] is a production-scale analysis engine to perform semantics-based search on a target codebase, essentially by treating code as data. The analysis first extracts a full hierarchical representation of code (e.g., the AST) into a relational database. It then runs analysis *queries* against the database to compute result tuples, for instance, pairs of source locations and error messages for bug finding. CodeQL queries are written in a declarative, object-oriented logic programming language called QL, which uses Datalog as underlying semantic model [13]. It also provides a *standard library* of queries that implement control-flow and data-flow analyses, as well as support for mainstream languages including JavaScript. The JavaScript model and the analyses are part of the open-source QL standard library, making them amenable to extensions.

A key feature that we use in our analyses are *path queries* that describe the data flow between a source and a sink in the codebase. They support expandable taint tracking with the possibility of using multiple flow labels. This is essential to implement our analysis algorithms described in Section 4. Specifically, we develop the custom path queries for pollution and gadget detection. We extend the taint tracking configuration to combine the call-flow and data-flow analyses, thus propagating tainted values through call flows in a context-sensitive way. This feature is essential for some of our analyses, e.g., to analyze entry points that receive tainted data from a database and not propagate the taint labels through code that is reachable from other entry points. We also model the array built-in functions `reduce`, `filter` and more, to correctly propagate tainted values via callback functions passed as arguments. This allows us to detect vulnerabilities that use `reduce` in the injection sink. We also resolve new functions created by `bind` call to propagate taints from the provided values of the `bind` arguments to the bound function parameters. Other changes include support for parameter passing via `apply()` and `call()` function calls, as well as the rest parameter syntax and the `arguments` object. We also improve the detection of exported functions of Node.js packages. Our analysis queries for pollution and gadget detection follow the methodology described in Section 6.1 and are publicly available as complementary material [43].

6 Evaluation

This section presents our experiments to validate the usefulness of our approach to detect and exploit POIVs. We perform the experiments on an Intel Core i7-8850H CPU 2.60GHz, 16 GB of memory. The tool, the analysis results and data are available in the GitHub repository [43].

6.1 Evaluation of Prototype Pollution

This section evaluates the effectiveness of our tool to detect injection sinks, reporting on precision and recall. While recent approaches already target this problem [27, 31, 32] for Node.js libraries, our key contribution is scalability with low-to-moderate precision loss, while achieving high recall. In contrast to prior work on libraries, we find that injection sinks are rare in real-world applications, motivating the need for high recall to identify exploitable vulnerabilities.

Benchmark. We compile an open-source dataset of 100 vulnerable Node.js packages, collected from the Snyk database [10]. By studying the proof-of-concept exploit provided in the vulnerability report, we manually identify code locations (file name and line number) of injection sinks pertaining to the assignment of an attacker-controlled value to the polluted property. We observe that some packages contain multiple exploitable injection sinks, which we also add to our benchmark. This new dataset serves as ground truth to

evaluate the detection capabilities of static analyses. For comparison, we also consider the dataset of 19 packages provided by the state-of-the-art work ODGen [32].

Setup. We use our benchmark to calculate the rate of true positives (TP), false positives (FP), and false negatives (FN) in an effort to identify the sweet spot between the precision and recall of the analysis. The precision metric describes how well the tool identifies exploitable injection sinks, while recall represents the fraction of real vulnerabilities reported by a tool. Following the methodology in Section 4.1, we run our tool in four different modes with the goal of identifying the most effective approach for detecting injection sinks in real-world applications. Our benchmark shows that attackers can have different levels of control over the injection sinks. While in general it can be sufficient to control the injection of the root prototype only, we notice that most exploits target injection sinks with attackers controlling both the name and value of a polluted property. Therefore, our tool distinguishes between the two cases, respectively, denoted as *General queries* and *Priority queries*. Moreover, since our analysis considers transitive dependencies, we distinguish between target functions considering *Exported Functions* and *Any Functions*, with the goal of identifying the best mode to analyze applications.

We also compare our results with three analysis queries which CodeQL recently made available publicly. We consider these CodeQL queries as baseline queries and run them on our benchmarks. Moreover, we conduct a direct comparison with ODGen [32] on the dataset of 119 libraries.

Results. We report the evaluation results in Table 3 in Appendix and here discuss only the precision and recall metrics in comparison with CodeQL's baseline queries and ODGen.

CodeQL provides three queries to detect prototype pollution, one of which yields no results, hence we discard it. The remaining two queries detect vulnerabilities in 57 packages, with 47% and 67% precision and 42% and 21% recall, respectively. While our analysis queries have been developed independently, our main goal is to achieve high recall with good precision. A fair comparison with the CodeQL baseline corresponds to our *General* queries with *Exported Functions*, which yields 35% precision and 88% recall. The improved recall is due to better support for exported functions, array built-in functions, and complete semantic modeling of reflective invocations through `apply()`, `call()` and `build()` functions. These results confirm the challenge of statically analyzing data flows in JavaScript without precise models of the language semantics and built-in functions.

Our second experiment is an evaluation of *General* queries with *Any Functions* as entry points. The analysis achieves 31% precision and 97% recall, producing 5 false negatives. This false negatives are in packages such as `Templ8` and `total_js` with injection sinks into code that is generated dynamically via `new Function()`, which CodeQL does not support. The high recall shows that injection sinks appear in a few adjacent functions, which reduces the risk of losing the taint marks

because of missing models of built-in functions. However, precision deteriorates because some detected patterns are not actually reachable from the library API with attacker-controlled arguments. We also notice the precision loss is much less than one would expect from an analysis with the strong assumption that any function's arguments are attacker-controlled. We believe this is due to the shape of injection sinks requiring patterns that are not very common in real-world code (see Section 4.1). While 31% precision in aggregate results is not ideal, our analysis produces less than 10 false positives for 90% of the benchmarks.

Our third experiment is the evaluation of *Priority* queries with *Any Functions* as entry points. In this setting, the attacker controls the name and value of the polluted property, thus it can leverage any existing gadget. The analysis achieves 40% precision and 93% recall. The additional restrictions on arguments increase the precision metric and keep high recall. Because the analysis starts from any function and does not require specifying the entry points, we can easily apply it to real-word application analysis. We identify this analysis query as the sweet spot between precision and recall, and use it to detect vulnerabilities in real applications (Section 6.3).

Our final experiment is a direct comparison with ODGen [32]. ODGen's analysis corresponds to our *General* queries with *Exported Functions*. ODGen is tailored towards high precision, while the authors recognize the need for high recall. In fact, our experiment shows that ODGen achieves 100% precision and 50% recall on the dataset of 19 libraries, while our analysis achieves 95% precision and 95% recall (see the evaluation results in [43]). Nonetheless, ODGen detects vulnerabilities in 17 out of the 19 libraries, but fails to detect some variants of these vulnerabilities. We further evaluate ODGen on our dataset of 100 packages to find that it achieves 87% precision and 33% recall.

6.2 Gadget Detection

We evaluate the feasibility of our universal gadget detection analysis and discuss the most important gadgets. We run our analysis on Node.js version 16.13.1 and exploit each gadget both on Linux and on Windows operating systems.

6.2.1 Dynamic Analysis

We download the source code of Node.js and parse it to extract all directly-accessed properties. We obtain a total of 18,741 property names for the analyzed codebase [8]. For each name, we install a getter on `Object.prototoype` to detect any potential access to that property by Node.js' internals.

Subsequently, we exercise the APIs under test with typical inputs from the Node.js documentation, e.g., execute the `ls` command with `spawn` [7], and log any potential accesses observed by the getter. In total, we analyze three APIs, i.e., `child_process.spawnSync`, `require`, and

`vm.runInNewContext`, and obtain 10, 11, and 16 candidate properties, respectively. The usage of these properties is further analyzed in the Node.js' codebase, using static analysis.

We note that the inputs used for driving the dynamic analysis are by no means exhaustive. We probably cover only a small part of the target APIs in our tests, potentially missing property accesses that only happen when the API is invoked with certain arguments. Nonetheless, for such cases, the resulting gadgets would be of limited use, as they would require the target application to pass those exact arguments to trigger the gadget. Instead of being comprehensive in our test case, we focus on the typical usages of the target APIs, which we believe yields easy-to-trigger gadgets.

Given the low number of properties detected in this step, one could directly fuzz these properties and build proof-of-concept exploits. However, we further trace their usage inside the Node.js codebase to understand if they are exploitable.

6.2.2 Static Analysis

As discussed in Section 4.2, our approach takes the JavaScript source code of Node.js and the property names from the dynamic analysis phase as input, and reports a call chain to reach a property read and a data flow from the property read to an internal function invocation. We only analyze the JavaScript code from the folder *lib* of the repository [8]. The analyzed codebase contains 70,493 lines of code (LOC).

In total, we identify 778 exported functions that reach the property reads (sources), and 342 in which values read from these properties flow into internal functions (sinks). We note that while inspecting all these code locations rigorously requires a significant amount of manual effort, we opt for pragmatic exploration: we first analyze the sink and decide if the invoked API, usually a native binding to the C/C++ code, is a relevant injection sink. If so, we continue with inspecting the sources to see which JavaScript APIs we can use to reach a particular code location.

Let us consider the case of `shell`, a universal property identified by our dynamic analysis. The static analysis identifies 8 sources, meaning that the reads of `shell` are reached from eight Node.js exported functions, mostly from the file `lib/child_process.js`. By propagating taints from all detected property reads, we identify 11 function invocations in which the tainted value leaves the JavaScript world. One of them is located in the file `lib/internal/child_process.js` and is a call to the native `spawnSync` in the C++ bindings. By studying the bindings and the way they are invoked, we conclude that the `shell` universal property is a candidate for developing a gadget.

We thus proceed to further study the operations performed on the value stored in the universal property inside the Node.js codebase. CodeQL provides great support in this step, allowing us to jump at the relevant code locations where this value is read and then manipulated. We already know from the dy-namic analysis step that the Node.js core performs a read from this universal property when the function `spawnSync` is invoked, but by running a call graph reachability analysis we identify four other APIs that reach one of the sources.

We build a simple test case to first pollute the `shell` property with the value `touch` and then invoke one of the affected JavaScript API, i.e., `spawnSync`. By observing the side-effect of this test case, i.e., the file creation in the current directory, we conclude that if an attacker can pollute `shell`, the API under test uses its value as command, instead of the argument passed by developers. We next discuss this gadget and others.

6.2.3 Universal Gadgets

We open source all the detected gadgets for Node.js in a GitHub repository [42]. Table 1 overviews the gadgets for the target Node.js version. Some of the gadgets are OS-specific, while most of them run on both considered OSs. We emphasize the diverse set of universal properties involved, showing that gadgets are not isolated buggy cases, but they are common place. These gadgets correspond to a handful of target APIs inside the Node.js core, but that a motivated attacker can probably find many more inside the codebase of a target application. Finally, as we discuss below, some gadgets allow arbitrary code execution with a relatively strong precondition, while others allow hijacking the control flow with a weaker precondition. More importantly, an attacker can combine two such gadgets to get the best of both worlds.

We now discuss some of our most important gadgets and their assumptions to be fulfilled. Let us consider an application that invokes the `execSync` API with a string literal:

```
1  const { execSync } = require('child_process');
2  console.log(execSync('echo "hi"').toString());
```

This benign looking code prints the string `hi` in the console. Staicu et al. [45] report that such API calls are prevalent in the NPM ecosystem, but they consider safe all call sites with constants as arguments, like the one above. That is because they assume an attacker cannot manipulate the command's value as it is set to a fixed value by developers. We find that this assumption does not hold in the presence of prototype pollutions. If attackers can pollute arbitrary properties in the runtime, they can hijack both the command to be executed and its environment variables. Consider the polluted properties:

```
1  Object.prototype.shell = "node";
2  Object.prototype.env = {};
3  Object.prototype.env.NODE_OPTIONS =
       "--inspect-brk=0.0.0.0:1337";
```

They trick the benign code above into spawning a new Node.js process with the debug port open, acting as a reverse shell. This is because the polluted property `shell` overwrites the command given by developers and `env.NODE_OPTIONS` is set as environment variable of the current process and subsequently copied to all children processes.

ID	Universal properties	Trigger	Impact	OS
G_1	shell, env	Call command injection API	Execute an arbitrary command	L+W
G_2	shell, env	Call command injection API	Execute an arbitrary command	L
G_3	shell, input	Call command injection API	Execute an arbitrary command	W
G_4	main	Import a package without a declared "main"	Import an arbitrary file from the disk*	L+W
G_5	main	Require a package without a declared "main"	Require an arbitrary file from the disk*	L+W
G_6	exports, 1	Require a file using a relative path	Require an arbitrary file from the disk*	L+W
G_7	'='C:'	Resolve a file path	Resolve the path to a different file	W
G_8	contextExtensions	Require a file using a relative path	Overwrite global variables of the file	L+W
G_9	contextExtensions	Compile function in a new context	Overwrite function's global variables	L+W
G_{10}	shell, env, main	Require a package without a declared "main"	Execute an arbitrary command	L+W
G_{11}	shell, env, exports, 1	Require a file using a relative path	Execute an arbitrary command	L+W

Table 1: A summary of the identified Node.js universal gadgets. For each gadget, we show the properties that the attacker must pollute beforehand, the action that triggers the gadget, and the produced effect. The last column shows the operating system on which the gadget works: Linux (L), Windows (W), or both (L+W). * denotes gadgets for which we have a Windows variant that achieves arbitrary command execution using the SMB protocol.

The presented gadget affects all the APIs for command execution in Node.js: spawn, spawnSync, exec, execSync, execFileSync. A precondition for this attack is that the target command execution call site should not explicitly set an options argument, e.g., for an execSync call, there should be no second argument passed. The existence of this gadget implies that every Node.js application that is vulnerable to prototype pollution and uses a command execution API after a pollution is vulnerable to remote code execution.

Now consider an application that does not directly use such APIs in user-facing code. An attacker can still leverage code that is present on the machine to trigger a command execution API. We found three gadgets that exploit the require and import methods. Consider the following example:

```
1  Object.prototype.main = "./../../pwned.js"
2  // trigger call
3  require('my-package')
```

A precondition for this gadget is that my-package does not have a main property defined in its package.json. If the main property of the root prototype is polluted, at require time, the value of this property is used for retrieving the code to be executed, instead of the legitimate code of the module. The attacker can thus indicate an arbitrary file on the disk to be loaded in the engine. In particular, they can specify a file that contains calls to command execution APIs. For example, the popular growl package [6] contains a file called test.js that invokes the package with different test values. Considering that growl uses spawn internally, the attacker can successfully trigger such APIs call by setting the main property to point to the growl's test file. Moreover, we identified a file shipped with the NPM command line tool that can be used for the same nefarious purpose: npm/scripts/changelog.js.

To the best of our knowledge, the gadget above is the first evidence ever reported that shows that hijacking control flow through code reuse attacks is possible in Node.js. This moti-

vates the need for debloating techniques like Mininode [28].

In addition to the already alarming findings, an attacker can combine the two gadgets discussed above to obtain a powerful universal gadget:

```
1  // pollutions for the first gadget
2  Object.prototype.main = "/path/to/npm/scripts/
       changelog.js";
3  // pollutions for the second gadget
4  Object.prototype.shell = "node";
5  Object.prototype.env = {};
6  Object.prototype.env.NODE_OPTIONS =
       "--inspect-brk=0.0.0.0:1337";
7  // trigger call
8  require("bytes");
```

When the bytes package is loaded, the first gadget instructs the engine to load the changelog.js file. This file in turn invokes execSync, which triggers the second gadget, starting a Node.js process with a debugging session.

Finally, let us present another gadget that lets attackers load arbitrary files into the engine. By polluting the root prototype's properties 1 and exports, an attacker can execute an arbitrary file from the disk when a relative path is loaded:

```
1  let rootProto = Object.prototype;
2  rootProto["exports"] = {".":"./changelog.js"};
3  rootProto["1"] = "/path/to/npm/scripts/";
4  // trigger call
5  require("./target.js");
```

While performing relative path resolution, the require method checks if the target path points to an ES6 module. During this process, the polluted property 1 is inadvertently read when applying a destructuring operator in the file /internal/modules/cjs/loader.js:

```
1  const { 1: name, 2: expansion = "" } =
       StringPrototypeMatch(...) || [];
```

Thus, the attacker-controlled value is assigned as the target module's name. Thereafter, the require method wrongly

concludes that the relative path `./target.js` resolves to the attacker-controlled location `/path/to/npm/scripts/` and that the path corresponds to an ES6 module. The `exports` property is used to confuse the `require` method further by providing the entry point for this non-existing module. Although at the attacker-controlled target location, there is no `package.json` file present, the `require` method still concludes that this is a valid module path. We note that this gadget is not portable to legacy Node.js versions, e.g., version 14.15.0. Thus, an important precondition for exploitation is that the target system must use a recent Node.js version.

We emphasize once again how dangerous the identified gadgets are. Many fairly-large applications would probably meet the preconditions for an RCE, once a prototype pollution is in place: (i) require a file using a relative path or a package with no `main` entry, and/or (ii) have a dependency that uses a command execution API when loaded.

To further study the impact of our gadgets, we estimate the prevalence of their triggers in an experiment with the 10,000 most dependent-upon NPM packages. We measure that 1,958 have no `main` entry in their package.json (G_4, G_5, G_{10}), 4,420 use relatives paths inside require statements (G_6, G_8, G_{11}), and 355 directly use the command injection API (G_1, G_2, G_3). This indicates that many of our gadgets could be deployed against clients of these packages, once a pollution is in place. However, this is an upper bound on the actual prevalence of the gadgets because: (i) the attacker may have a hard time invoking the trigger's code through the public interface of the package, e.g., the code using the command injection API, (ii) some gadgets may not work out of the box because of side-effects in the target package, i.e., polluting the property `1` may have many unintended side-effects that can prevent the gadget from working, (iii) an attacker may find it difficult to deploy a pollution before the gadget, e.g., for the require gadgets, very often, the pollution needs to happen in the application's initialization phase. Nonetheless, considering the power of these gadgets and their widely-available triggers, prototype pollution should be considered a critical security vulnerability in the current Node.js landscape.

6.3 End-to-End Exploitation

We evaluate our approach on popular Node.js applications from GitHub to validate its usefulness in a practical setting.

Setup. We use the GitHub API to search for JavaScript repositories and order them by the number of stars. We then select for further analysis the top 14 web applications running on Node.js, as well as NPM CLI, the JavaScript package manager, because it is installed on every machine with Node.js as default. NPM CLI is also the largest analyzed application in our dataset. We clone the GitHub repository of each application locally and perform the analysis against it.

Methodology. Following the workflow described in Section 4, we first run our *Priority* query with *Any Functions* as entry

points against a target application. The query reports the potential injection sinks and a list of the functions that pass tainted data to these sinks. The list contains functions that are actual entry points of the application and functions that take data from the environment (e.g., a database) and pass it to the injection sink. For the latter, we perform a call flow analysis to detect the application entry points. Second, we manually classify all reported cases as either false positives or *locally exploitable*. Based on the project structure, we also filter out cases in testing and client-side code. We discard these cases because the code does not execute on the server and cannot lead to RCE. Third, we study the application's threat model to detect conditions for exploiting the remaining (locally exploitable) cases. This is a manual process that requires studying the documentation and code of the application. We match the entry points pertaining to the threat model with the detected entry points leading to the injection sinks. Fourth, we verify the matched entry points dynamically by deploying the application locally and generating a payload to pollute the `toString` property. Whenever the payload fails, we rely on the debugger by examining code transformations and validations along the path, and modifying the payload accordingly. Finally, once the pollution is confirmed, we search for the gadgets that may lead to RCE, as described in Section 6.2. If the gadget can be triggered after the execution of the injection sink, we change the payload to pollute gadget-specific properties.

Results. Table 2 presents the analysis results for 15 widely-used Node.js applications. *Total* provides the number of detected prototype pollutions in the application's codebase and the total time for their manual analysis. The analysis finds cases in 8 applications, which we investigate and classify manually. *False Positives* contains the false positives due to over-approximate analysis; *Client-Side* and *Testing Code* show the cases that do not execute on a server-side directly.

We mark the remaining cases (column *Suspicious*) for further investigation. Suspicious cases are locally exploitable patterns, i.e., they can be exploited if an attacker controls all function parameters. We verified the suspicious cases to find eight prototype pollutions (in NPM CLI, Parse Server and Rocket.Chat) that are exploitable according to the threat model of these applications. We also found the gadgets that lead to RCE as explained below. As a sanity check, we run the original CodeQL baseline queries for NPM CLI and Parse Server applications, however, they do not detect exploitable prototype pollutions.

To estimate the manual effort, we track the time to verify the reported cases by one of the authors. A false positive takes an average of 2.6 minutes because the analysis affects a small code fragment. Similarly, non server-side code and testing code take on average 3.8 minutes and 1.2 minutes, respectively. The analysis of suspicious cases takes more time and depends on the quality of the documentation and application's code. The time in *Suspicious* column includes the study of the

Application's Repository	Stars	Lines of code	Total		Exploitable		Suspicious		Testing Code		Client-Side Code		False Positives	
			Cases	Time	Cases	Time	Cases	Time	Cases	Time	Cases	Time	Cases	Time
typicode/json-server	57,257	2,374	0		-		-		-		-		-	
expressjs/express	54,883	14,450	0		-		-		-		-		-	
meteor/meteor	42,673	202,213	26	255	0		5	210	4	10	8	5	9	30
strapi/strapi	40,724	168,998	3	5	0		0		0		0		3	5
TryGhost/Ghost	38,944	125,696	4	55	0		1	50	0		2	3	1	2
hexojs/hexo	33,666	21,073	1	40	0		1	40	0		0		0	
sahat/hackathon-starter	32,431	2,326	0		-		-		-		-		-	
koajs/koa	31,910	4,596	0		-		-		-		-		-	
RocketChat/Rocket.Chat	31,059	242,949	5	1555	1	1500	3	50	0		1	5	0	
balderdashy/sails	22,085	24,445	0		-		-		-		-		-	
emberjs/ember.js	22,034	113,749	6	60	0		2	40	1	10	0		3	10
fastify/fastify	21,043	37,049	0		-		-		-		-		-	
parse-community/parse-server	19,045	107,909	7	3225	5	3220	0		0		0		2	5
docsifyjs/docsify	18,946	7,603	0		-		-		-		-		-	
npm/cli	5,371	713,648	15	603	2	360	6	230	1	3	0		6	10

Table 2: Evaluation results for the applications' analysis. *Cases* shows the number of detected cases of a certain category; *Time* shows the time in minutes to manually classify and validate these cases.

threat model and the matching of detected entry points. The *Exploitable* column includes the time to set up an application, debugging and verification of prototype pollution, search for gadgets, and combination of all attack ingredients. For example, most time for the Parse Server exploit was spend to find a race condition that triggers the injection and attack sinks in the correct order. For NPM CLI, a time-consuming task was to find a way to store the payload to NPM Registry via a malicious package and subsequently parse it during the package installation. The analysis and exploitation of Rocket.Chat required an LDAP server setup that provides a payload to the injection sink, and the configuration of a custom synchronization with the LDAP server. This process is not fully described in the official documentation and required a lot of manual testing of various options.

We now describe the RCE exploits for two applications and refer to the extended material for full details [43].

6.3.1 Parse Server RCEs

Parse Server is an open source Backend-as-a-Service (BaaS) framework that provides REST APIs to object and file storage, user authentication, push notifications, dashboard, and uses MongoDB or PostgreSQL as database. The Parse Server has pioneered BaaS systems in 2011 and has brought the serverless, low-touch deployment model to web and mobile backends.

Threat model. The Parse Server can be deployed as a self-hosted solution. In this scenario, an attacker can send any requests to the server, but cannot modify any settings on the server. Therefore, we expect that an application must be secure in the default configuration. In the second scenario, we consider the Parse Server as a part of cloud infrastructure, e.g., Back4App [1]. The attacker can create their own account and become the administrator of that account. This allows the attacker to change some settings, for example, the webhook

triggers. This scenario puts any available configuration at risk for attacks including the default configuration.

Detecting sinks. Our static analysis framework detects 7 unique injection sinks. We marked 5 cases as suspicious by manual validation. One of the suspicious cases is located in the sanitizer of database records as shown in Listing 3.

```
1  function expandResultOnKeyPath(obj, key, res) {
2    if (key.indexOf('.') < 0) {
3      obj[key] = res[key];
4      return obj;
5    }
6    const path = key.split('.');
7    const firstKey = path[0];
8    const nextPath = path.slice(1).join('.');
9    obj[firstKey] = expandResultOnKeyPath(
10     obj[firstKey] || {},
11     nextPath, res[firstKey]);
12   return obj;
13 }
```

Listing 3: Injection sink in Parse Server

This function can be abused to pollute `Object.prototype`. If the attacker controls the input data and passes the value `"obj.__proto__.evalFunctions"` to the parameter `key` and the object `{obj:{__proto__:{evalFunctions: 1}}}` to `result`, then sanitization sets the new property `evalFunctions` to Object's prototype.

Following our methodology, we perform a call flow analysis to detect entry points for the injection sink. A handler of the GET request triggers data reading from the database and then executes the vulnerable sanitizing code. Other detected injection sinks may be triggered via a PUT request by a payload delivered from a third-party webhook application.

In order to detect potential RCE gadgets, we search in Parse Server codebase for universal gadgets and functions that evaluate the code at runtime, e.g., `eval`. The analysis reports a gadget using the `require` function, where an attacker can directly control its argument through a polluted property. The analysis also reports an attack sink in the official MongoDB

BSON parser [2] that deserializes objects from a database, and can evaluate JavaScript code stored in this object. However, the code evaluation is possible only if we set the configuration parameter `evalFunctions`, see Listing 4. This option is not defined by default, but the attacker can pollute the prototype and bypass the if-statement condition in line 5.

```
1  const evalFunctions =
2    options['evalFunctions'] == null
3    ? false
4    : options['evalFunctions'];
5  if (evalFunctions)
6    eval(functionString);
```
Listing 4: Attack sink in Parse Server

Exploitation. The attacker should first pollute the prototype via the injection sink and then trigger the attack sink in a second request. A challenge to exploit prototype pollution is that the polluted property may break the application workflow. In this setting, the web request handler throws an exception whenever `Object.prototype` is polluted. Thereby, the attacker cannot successfully handle the requests in the required order. However, we could bypass it using a *race condition* in the application workflow.

Four of the RCE exploits for Parse Server use the same gadget and attack sink in Listing 4 as follows: First, the attacker sends requests to store payloads in the database. Second, it sends the GET request to trigger the attack sink but delays its execution in the database until the next request. Third, the exploit sends the PUT request to trigger the injection sinks. Because the first request takes longer, a payload triggers the injection sink while another payload reaches the attack sink and executes arbitrary code. The fifth exploit adapts the `require` gadget discussed in Section 6.2.3.

6.3.2 NPM CLI RCEs

NPM CLI [9] is the command line client that allows developers to install and publish packages to NPM registries. During a package installation, NPM CLI puts modules in place so that Node.js can load them, manages dependency conflicts, and may run the pre- and post-install scripts from the package.

Threat model. The public NPM registry can be untrusted, e.g., by storing malicious packages. Since it is a shell tool that is run on a developer's machine, RCE attacks have the highest impact. NPM CLI has the option `--ignore-scripts` to disable running scripts specified in `package.json` files. Therefore, the threat model considers the arbitrary script execution that breaks out of the `--ignore-scripts` flag as unintended RCEs. We have the following constraint: the injection and attack sinks should be available during the execution of the command that installs a malicious package.

Detecting sinks. The static analysis reports 15 unique injection sinks. We marked 8 cases as suspicious. Due to the restricted threat model, we then focus on matching the detected cases to the threat model. When NPM

CLI installs the package, it parses the configuration file `npm-shrinkwrap.json` from the package regardless of the option `--ignore-scripts`. NPM CLI then invokes `diff-apply` and `copyPath` functions from the `parse-conflict-json` package to parse the configuration file. Two of the suspicious cases are located in these functions. Section 3 describes the injection sink in `diff-apply` and the attack sink for the RCE exploitation. We verified manually that the exploitation in both cases leads to RCE.

Exploitation. The NPM CLI invokes the `spawn` function to run the `git` commands for git-located package dependencies. This happens after parsing the configuration files, and therefore, after the injection sink execution. The git supports the command execution via the environment variable `GIT_SSH_COMMAND`. If this environment variable is set, git uses the specified command, instead of ssh, to connect to a remote system. Thereby, the attacker can craft the package configuration file to initiate the call `diffApply({}, {path:['__proto__','env'], value: {GIT_SSH_COMMAND: 'calc &'}, op: ADD})` and wait for the `spawn` invocation of the git command. This payload triggers arbitrary code execution, here launching a calculator.

7 Related Work

This section discusses closely related work targeting object injection vulnerabilities in general and prototype pollution in particular. We also discuss related security analyses for the Node.js ecosystem and client-side JavaScript security.

Prototype pollution vulnerabilities. The security community became aware of prototype pollution vulnerabilities in 2018 in a white paper of Arteau [12] which uses dynamic analysis to showcase feasibility in a number of Node.js libraries as well as an end-to-end exploit in the Ghost CMS platform. The risks and the impact of prototype pollutions has been mainly discussed in security practitioner forums [3], with the exception of a handful of recent research papers [25, 27, 31, 32, 51]. Notably, the work of Li et al. [31, 32] proposes *object dependence graphs* to statically find injection vulnerabilities in Node.js libraries, including prototype pollution. Object dependent graphs allow identifying prototype injection sinks similar to our multi-taint analysis, though with higher precision due to the analysis of branch conditions. By contrast, our approach trades precision for scalability to analyze fully-fledged applications and libraries. In addition, our key focus is on universal gadget identification and end-to-end exploitation which no prior work has addressed systematically so far. Kim et al. [27] develop DAPP, a static analysis tool to detect prototype injection sinks in Node.js libraries by means of pattern analysis. DAPP's lightweight analysis results in low precision and recall, while focusing only on libraries. The recent work by Kang et al. [25] explores prototype pollution on the client-side to exploit a range of vulnerabilities (XSS,

cookie and URL manipulation) by using dynamic taint tracking. Compared with static analysis, dynamic analysis may miss some gadgets because of code coverage limitations, yet it can be helpful to validate the reachability of our injection and attack sinks, which we currently do manually. Xiao et al. [51] study hidden property attacks in Node.js applications, a type of vulnerability which is related to prototype pollution.

Object injection vulnerabilities. We classify POIVs in the general context of object injection vulnerabilities (OIVs). Prior work studies OIVs targeting insecure deserialization by mean of static analysis in a variety of languages including Java [24, 36], PHP [15, 17, 21], .NET [35, 41], and Android [39]. The work of Dahse et al. [16, 17] develops static analysis to systematically detect OIV gadgets in PHP applications. Shcherbakov and Balliu [41] propose a static analysis for detecting object injection patterns for .NET application, including the framework and libraries, and implement a tool called SerialDetector. Arguably, our work faces similar challenges with scaling the static analysis to real-world languages, though in the more intricate context of JavaScript.

Node.js ecosystem security. There is an increasing interest in studying the security of Node.js, both in academia and in industry. Most prior work has concentrated on so-called software supply chain security, i.e., studying security problems that are prevalent in libraries: injections [22, 32, 45], hidden property abuse [51], prototype pollution [31, 32], malicious packages [20, 52], running untrusted code [11, 49, 50], ReDoS [18, 19, 33, 44], code debloating [28]. There is also initial evidence that these problems in libraries affect websites in production [31, 44]. We are the first to show the existence of universal gadgets in Node.js and to study the impact of prototype pollution, beyond denial-of-service attacks.

Static analysis for Node.js. Madsen et al. [34] propose augmenting call graphs with information about event propagation to find bugs in Node.js programs. Staicu et al. [45] advocate using intra-procedural data flow analysis to infer runtime policies for injection sinks. Nielsen et al. [37] introduce feedback-driven abstract interpretation for detecting injection vulnerabilities in Node.js code. More recently, Nielsen et al. [38] show how modular call graphs can be used to reduce false positives alerts in software composition analysis. Li et al. [31, 32] propose using object dependency graphs for finding prototype pollution, injection, and path traversal vulnerabilities. We are the first to propose using static taint analysis for detecting universal gadgets.

Client-side JavaScript security. Lekies et al. [30] study XSS vulnerabilities on the web using fine-grained dynamic taint analysis. Hedin et al. [23] present JSFlow, a more sophisticated information flow analysis for detecting integrity and confidentiality problems in web applications. Recently, Lekies et al. [29] discuss how script gadgets can be used to bypass existing cross-site scripting mitigation. Roth et al. [40] further study the effect of script gadgets on content security poli-

cies. Steffens and Stock [48] present PMForce, a lightweight dynamic analysis augmented with forced execution for studying post message handlers. Khodayari and Pellegrino [26] propose JAW, a hybrid analysis tool based on code property graph, showing its usefulness by studying client-side CSRF vulnerabilities. None of the work above studies the relation between prototype pollution and injection vulnerabilities.

8 Conclusion

We presented the first principled study on the impact of prototype pollution vulnerabilities in Node.js. We propose a semi-automated approach for detecting end-to-end exploits, consisting of three phases: (i) static analysis for detecting pollutions, (ii) hybrid analysis for detecting gadgets, and (iii) static analysis with human-in-the-loop for developing end-to-end exploits. We apply our approach to large codebases to find eight exploitable RCE vulnerabilities directly enabled by prototype pollution, and eleven universal gadgets [42] that are shipped with the Node.js runtime. Finally, we show that universal gadgets introduce a new threat in the Node.js ecosystem: hijacking the control flow of a program to (ab)use unused code available in the application's dependencies.

References

[1] Back4App. https://www.back4app.com.

[2] BSON Parser for node and browser. https://github.com/mongodb/js-bson.

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

[4] CodeQL. https://codeql.github.com.

[5] Exploiting prototype pollution – RCE in Kibana (CVE-2019-7609). https://research.securitum.com/prototype-pollution-rce-kibana-cve-2019-7609.

[6] Growl - NPM. Growl support for Node.js. https://www.npmjs.com/package/growl.

[7] Node.js documentation. https://nodejs.org/api/child_process.html.

[8] Node.js JavaScript runtime v16.13.1. https://github.com/nodejs/node/tree/v16.13.1/lib.

[9] NPM - a JavaScript package manager. https://github.com/npm/cli.

[10] Snyk. https://snyk.io.

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

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

[13] Pavel Avgustinov, Oege De Moor, Michael Peyton Jones, and Max Schäfer. Ql: Object-oriented queries on relational data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[14] 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.

[15] Johannes Dahse and Thorsten Holz. Static detection of second-order vulnerabilities in web applications. In *USENIX Security 14*, pages 989–1003, 2014.

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

[17] Johannes Dahse, Nikolai Krein, and Thorsten Holz. Code reuse attacks in PHP: automated POP chain generation. In *Conference on Computer and Communications Security (CCS)*, pages 42–53, 2014.

[18] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2018.

[19] James C. Davis, Francisco Servant, and Dongyoon Lee. Using selective memoization to defeat regular expression denial of service (ReDoS). In *Symposium on Security and Privacy (S&P)*, 2021.

[20] 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.

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

[22] 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.

[23] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. JSFlow: tracking information flow in JavaScript and its APIs. In *Symposium on Applied Computing (SAC)*, 2014.

[24] 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)*, pages 779–790, 2016.

[25] 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.

[26] Soheil Khodayari and Giancarlo Pellegrino. JAW: studying client-side CSRF with hybrid property graphs and declarative traversals. In *USENIX Security Symposium*, 2021.

[27] 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*, pages 1–23, 2021.

[28] Igibek Koishybayev and Alexandros Kapravelos. Mininode: Reducing the attack surface of Node.js applications. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[29] 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)*, pages 1709–1723, 2017.

[30] 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)*, pages 1193–1204, 2013.

[31] 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, page 268–279, New York, NY, USA, 2021. Association for Computing Machinery.

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

[33] Yinxi Liu, Mingxue Zhang, and Wei Meng. Revealer: Detecting and exploiting regular expression denial-of-service vulnerabilities. In *Symposium on Security and Privacy (S&P)*, 2021.

[34] Magnus Madsen, Frank Tip, and Ondrej Lhoták. Static analysis of event-driven node.js javascript applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, 2015.

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

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

[37] 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.

[38] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. Modular call graph construction for security scanning of node.js applications. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2021.

[39] Or Peles and Roee Hay. One class to rule them all: 0-day deserialization vulnerabilities in android. In *WOOT'15*, 2015.

[40] 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.

[41] 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.

[42] Mikhail Shcherbakov, Musard Balliu, and Cristian-Alexandru Staicu. Server-Side Prototype Pollution Gadgets. https://github.com/yuske/server-side-prototype-pollution.

[43] Mikhail Shcherbakov, Musard Balliu, and Cristian-Alexandru Staicu. Silent Spring: Prototype Pollution Leads to Remote Code Execution in Node.js - Artifacts. https://github.com/yuske/silent-spring.

[44] Cristian-Alexandru Staicu and Michael Pradel. Freezing the web: A study of redos vulnerabilities in JavaScript-based web servers. In *USENIX Security Symposium*, 2018.

[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, Sazzadur Rahaman, Ágnes Kiss, and Michael Backes. Bilingual problems: Studying the security risks incurred by native extensions in scripting languages. *arXiv preprint arXiv:2111.11169*, 2021.

[47] 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.

[48] Marius Steffens and Ben Stock. PMForce: Systematically analyzing postmessage handlers at scale. In *Conference on Computer and Communications Security (CCS)*, 2020.

[49] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. Breakapp: Automated, flexible application compartmentalization. In *Network and Distributed System Security Symposium, (NDSS)*, 2018.

[50] Nikos Vasilakis, Cristian-Alexandru Staicu, Grigoris Ntousakis, Konstantinos Kallas, Ben Karel, André DeHon, and Michael Pradel. Preventing dynamic library compromise on Node.js via RWX-based privilege reduction. In *Conference on Computer and Communications Security (CCS)*, 2021.

[51] 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.

[52] 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.

Appendix

8.1 Evaluation Results

In Table 3, we present the results of the evaluation of ODGen, the original CodeQL queries (*Baseline queries*) and our custom queries (*Priority queries* and *General queries*) against our benchmark of 100 vulnerable NPM packages.

Package@Version	LoC	Baseline queries				Priority queries				General queries				ODGen	
		Prototype Polluting Assignment		Prototype Polluting Function		Exported Functions		Any Functions		Exported Functions		Any Functions			
		TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP
101@1.6.3	2,366	0/2	0	0/2	0	2/2	0	2/2	0	2/2	2	2/2	2	0/2	0
arr-flatten-unflatten@1.1.4	104	0/2	0	0/2	0	1/1	0	1/1	0	2/2	0	2/2	0	0/2	0
asciitable.js@1.0.2	173	0/1	0	1/1	1	1/1	0	1/1	1	1/1	0	1/1	1	1/1	0
assign-deep@1.0.0	56	0/1	0	1/1	0	1/1	0	1/1	0	1/1	1	1/1	1	0/1	0
bmoor@0.8.11	3,718	4/6	2	1/6	0	4/4	0	4/4	0	6/6	0	6/6	0	3/6	0
bodymen@1.0.0	17,993	1/1	3	0/1	0	1/1	2	1/1	6	1/1	8	1/1	10	0/1	0
changeset@0.1.0	1,427	3/3	1	0/3	0	1/1	0	1/1	0	3/3	0	3/3	0	0/3	0
class-transformer@0.1.1	735	0/2	0	0/2	0	2/2	0	2/2	0	2/2	0	2/2	0	0/2	0
confucious@0.0.12	7,046	7/7	1	0/7	0	4/4	3	4/4	5	7/7	4	7/7	4	1/7	1
connie@0.1.0	13,433	0/3	0	1/3	1	1/1	0	1/1	1	3/3	0	3/3	4	0/3	0
controlled-merge@1.0.0	171	0/3	0	2/3	0	2/2	1	2/2	1	3/3	1	3/3	1	3/3	0
copy-props@2.0.4	348	1/1	1	0/1	0	0/1	0	0/1	0	0/1	0	1/1	1	0/1	0
deap@1.0.0	698	0/2	0	2/2	0	0/2	0	2/2	1	0/2	0	2/2	1	1/2	2
deep-defaults@1.0.5	17,475	0/1	3	1/1	0	1/1	2	1/1	4	1/1	8	1/1	8	0/1	1
deep-override@1.0.0	73	0/1	0	0/1	0	1/1	2	1/1	5	1/1	9	1/1	9	0/1	0
deep-set@1.0.0	41	0/1	0	0/1	0	1/1	0	1/1	0	1/1	1	1/1	1	1/1	0
deephas@1.0.5	351	0/1	0	0/1	0	1/1	0	1/1	0	1/1	1	1/1	1	0/1	0
deeply@3.0.0	238	0/1	0	0/1	0	0/1	0	0/1	0	0/1	0	0/1	0	0/1	0
deepref@1.1.1	136	0/1	0	0/1	0	0/1	0	1/1	0	0/1	0	1/1	0	0/1	0
deeps@1.4.5	231	1/1	1	1/1	0	1/1	0	1/1	0	1/1	2	1/1	2	1/1	0
defaults-deep@0.2.4	89	0/1	0	0/1	0	0/1	0	1/1	0	1/1	0	1/1	0	0/1	0
dot-object@2.1.2	5,500	2/4	5	0/4	0	4/4	2	4/4	6	4/4	10	4/4	20	0/4	0
dot-prop@2.0.0	34	1/1	1	1/1	0	1/1	0	1/1	0	1/1	1	1/1	1	0/1	0
dot-notes@3.2.0	223	1/1	1	0/1	0	1/1	0	1/1	0	1/1	1	1/1	1	1/1	0
dotty@0.0.1	475	1/1	1	0/1	0	1/1	0	1/1	0	1/1	1	1/1	1	1/1	0
dset@1.0.0	18	1/1	1	1/1	1	1/1	1	1/1	1	1/1	1	1/1	1	1/1	0
expand-hash@1.0.1	36	0/1	0	0/1	0	1/1	0	1/1	0	1/1	1	1/1	1	0/1	0
extend@3.0.1	63	0/1	0	1/1	0	1/1	1	1/1	1	1/1	1	1/1	1	1/1	0
field@1.0.1	76	4/4	0	0/4	0	2/2	0	2/2	0	4/4	0	4/4	0	1/4	0
@firebase/util@0.3.2	4,725	0/4	0	4/4	0	4/4	0	4/4	0	4/4	0	4/4	0	0/4	0
flattenizer@0.0.5	436	0/1	0	0/1	0	1/1	0	1/1	1	1/1	1	1/1	3	0/1	0
gammautils@0.0.81	6,919	1/1	3	0/1	1	1/1	1	1/1	1	1/1	4	1/1	4	1/1	0
gedi@1.6.3	7,160	1/1	6	0/1	2	1/1	2	1/1	3	1/1	7	1/1	8	0/1	0
getobject@0.1.0	126	1/1	1	0/1	0	1/1	0	1/1	0	1/1	1	1/1	1	0/1	0
hoek@5.0.0	764	0/1	0	0/1	2	1/1	3	1/1	4	1/1	5	1/1	5	0/1	0
immer@8.0.0	5,136	0/5	0	0/5	0	0/5	1	5/5	2	0/5	1	5/5	2	0/5	0
ini-parser@0.0.2	32	1/1	0	0/1	0	1/1	0	1/1	0	1/1	0	1/1	0	1/1	0
js-data@3.0.8	14,056	0/1	3	1/1	5	1/1	11	1/1	14	1/1	17	1/1	38	0/1	0
js-extend@0.0.1	53	0/1	0	1/1	0	0/1	0	1/1	0	0/1	0	1/1	0	1/1	0
js_ini@1.2.0	537	0/1	0	0/1	0	1/1	0	1/1	0	1/1	0	1/1	0	0/1	0
json-ptr@1.1.0	1,630	1/1	3	0/1	0	1/1	5	1/1	5	1/1	5	1/1	5	0/1	0
json8-merge-patch@1.0.1	635	0/1	0	1/1	0	1/1	0	1/1	0	1/1	0	1/1	0	0/1	0
just-extend@3.0.0	36	0/1	0	0/1	0	0/1	0	0/1	0	0/1	0	0/1	0	1/1	0
keyd@1.3.4	265	0/1	0	0/1	0	0/1	0	1/1	1	0/1	0	1/1	1	1/1	0
keyget@2.2.0	389	1/4	0	0/4	0	2/2	2	2/2	2	4/4	1	4/4	1	2/4	0
libnested@1.5.0	210	1/1	1	0/1	0	1/1	0	1/1	0	1/1	1	1/1	1	1/1	0
linux-cmdline@1.0.0	42	0/1	0	0/1	0	1/1	0	1/1	0	1/1	1	1/1	1	1/1	0
locutus@2.0.11	14,994	1/1	1	0/1	0	1/1	2	1/1	2	1/1	3	1/1	4	0/1	0
lodash@4.17.11	17,302	1/1	3	0/1	0	1/1	1	1/1	3	1/1	7	1/1	7	1/1	0
madlib-object-utils@0.1.6	81	1/1	1	0/1	0	1/1	0	1/1	0	1/1	1	1/1	1	1/1	0
merge@2.1.0	103	0/1	0	1/1	0	1/1	0	1/1	0	1/1	0	1/1	0	0/1	0
merge-deep@3.0.0	483	0/3	0	0/3	0	0/2	0	0/2	1	3/3	0	3/3	0	2/3	0

		TP	FP	TP	FP	TP	FP	TP	FP	TP	FP	TP	FP
merge-recursive@0.0.3	58	1/1	1	0/1	0	1/1	0	1/1	0	1/1	1	1/1	1	1/1	0
mixin-deep@2.0.0	29	0/1	0	1/1	0	1/1	0	1/1	0	1/1	0	1/1	0	0/1	0
mout@2.0.0-alpha.1	9,337	0/2	2	0/2	0	2/2	0	2/2	0	2/2	1	2/2	1	0/2	0
mpath@0.4.1	1,839	1/1	2	0/1	0	1/1	2	1/1	2	1/1	2	1/1	2	1/1	2
nconf_toml@0.0.1	4,743	0/1	0	0/1	0	1/1	0	1/1	1	1/1	2	1/1	2	0/1	0
nested-property@0.0.5	97	0/1	0	0/1	0	1/1	0	1/1	0	1/1	1	1/1	1	0/1	0
nestie@1.0.0	66	0/1	0	0/1	0	1/1	0	1/1	0	1/1	1	1/1	1	0/1	0
nis-utils@0.6.10	35,669	2/2	0	1/2	1	1/1	9	1/1	15	2/2	18	2/2	18	2/2	0
node.extend@2.0.0	958	0/1	0	1/1	0	1/1	1	1/1	1	1/1	1	1/1	1	1/1	0
node-forge@0.9.0	17,978	1/1	5	0/1	0	1/1	2	1/1	4	1/1	7	1/1	7	1/1	0
nodee-utils@1.2.2	22,385	2/2	0	1/2	0	1/1	5	1/1	12	2/2	11	2/2	15	2/2	0
object-collider@1.0.3	143	0/2	0	0/2	0	2/2	1	2/2	1	2/2	1	2/2	1	0/2	0
object-path-set@1.0.0	185	2/2	0	0/2	0	1/1	1	1/1	1	2/2	0	2/2	0	2/2	0
objnest@5.0.0	971	0/1	0	0/1	0	1/1	0	1/1	0	1/1	3	1/1	3	0/1	0
objtools@3.0.0	20,693	0/5	5	2/5	0	4/5	14	5/5	16	4/5	24	5/5	24	0/5	0
patchmerge@1.0.0	138	0/1	0	1/1	0	1/1	2	1/1	2	1/1	6	1/1	6	0/1	0
paypal-adaptive@0.4.1	203	0/1	0	1/1	1	1/1	1	1/1	2	1/1	2	1/1	2	0/1	0
phpjs@1.3.2	48,116	1/1	4	0/1	0	1/1	3	1/1	7	1/1	8	1/1	18	0/1	0
predefine@0.1.2	488	0/1	0	0/1	0	1/1	1	1/1	1	1/1	1	1/1	1	0/1	0
promisehelpers@0.0.5	132	1/1	1	0/1	0	1/1	0	1/1	0	1/1	1	1/1	1	1/1	0
properties-reader@2.0.0	1,293	0/1	0	0/1	0	1/1	2	1/1	2	1/1	7	1/1	7	0/1	0
property-expr@2.0.2	196	1/1	0	0/1	0	0/1	0	1/1	0	1/1	0	1/1	0	1/1	0
prototyped.js@2.0.0	7,911	0/1	0	0/1	0	1/1	0	1/1	0	1/1	0	1/1	0	1/1	0
putil-merge@3.0.0	68	0/2	0	0/2	0	2/2	0	2/2	0	2/2	2	2/2	2	0/2	0
querymen@2.1.3	18,205	1/1	3	0/1	0	1/1	2	1/1	6	1/1	8	1/1	10	0/1	1
safe-flat@2.0.0	298	0/1	0	0/1	0	1/1	0	1/1	0	1/1	0	1/1	0	0/1	0
safe-object2@1.0.3	104	0/1	0	0/1	0	1/1	0	1/1	1	1/1	0	1/1	1	0/1	0
safe-obj@1.0.0	242	0/1	0	0/1	0	1/1	1	1/1	1	1/1	2	1/1	2	0/1	0
safetydance@2.0.1	570	0/1	0	0/1	0	0/1	0	1/1	0	0/1	0	1/1	1	1/1	0
set-deep-prop@1.0.0	11	1/1	0	0/1	0	1/1	0	1/1	0	1/1	0	1/1	0	1/1	0
set-getter@0.1.0	179	0/1	0	0/1	0	0/1	0	0/1	0	1/1	1	1/1	1	0/1	0
set-in@2.0.0	172	1/1	0	0/1	0	1/1	0	1/1	0	1/1	0	1/1	0	1/1	0
set-object-value@0.0.5	113	0/2	0	0/2	0	2/2	4	2/2	4	2/2	6	2/2	6	1/2	0
set-or-get@1.2.10	115	1/1	0	0/1	0	1/1	0	1/1	0	1/1	0	1/1	0	1/1	0
set-value@3.0.0	123	2/2	1	1/2	0	1/1	0	1/1	0	2/2	1	2/2	1	2/2	0
shvl@2.0.1	18	0/1	0	0/1	0	1/1	0	1/1	3	1/1	1	1/1	4	0/1	0
smart-extend@1.7.3	8,949	0/1	0	1/1	1	1/1	2	1/1	3	1/1	2	1/1	3	0/1	0
@strikeentco/set@1.0.0	27	1/1	1	0/1	0	1/1	0	1/1	0	1/1	1	1/1	1	0/1	0
supermixer@1.0.3	9,843	0/1	2	0/1	0	0/1	5	0/1	9	0/1	8	0/1	12	0/1	0
Templ8@0.7.0	785	0/1	0	0/1	0	0/1	0	0/1	0	0/1	0	0/1	0	0/1	0
tiny-conf@1.1.0	255	4/4	0	0/4	0	2/2	0	2/2	1	4/4	0	4/4	1	1/4	0
total.js@3.4.6	40,699	0/1	3	0/1	1	0/1	1	0/1	2	0/1	4	0/1	7	0/1	0
undefsafe@2.0.2	544	0/1	0	0/1	0	1/1	0	1/1	0	1/1	0	1/1	0	0/1	0
upmerge@0.1.7	124	0/4	0	3/4	0	3/3	1	3/3	1	4/4	0	4/4	0	2/4	0
utils-extend@1.0.8	239	0/1	0	1/1	0	1/1	0	1/1	0	1/1	2	1/1	2	0/1	0
worksmith@1.0.0	91,294	0/1	4	0/1	0	0/1	7	1/1	13	0/1	19	1/1	33	0/1	1
y18n@3.2.1	129	3/3	0	0/3	0	1/1	0	1/1	1	3/3	0	3/3	0	2/3	0
yargs-parser@6.0.0	677	6/6	2	0/6	0	2/2	4	2/2	5	6/6	3	6/6	3	0/6	0
Total:		42.1	46.6	21.3	67.3	82.2	49.6	93.3	40.1	88.4	35.3	97	30.9	32.9	87.1

Table 3: Evaluation results of our benchmark analysis. The *TP* columns contain the number of detected cases / the total number of true positives for the package. The *FP* columns contain the number of false positive cases for the package. The *Total* row summarizes the data and presents the recall metric (in %) in the *TP* columns and the precision (in %) for the *FP* columns.