Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informazione e Bioingegneria
Master of Science in Computer Science and Engineering

Detecting prototype pollution vulnerabilities in JavaScript using static analysis

Advisor: Prof. Stefano ZANERO

Thesis by:
Francesco Dotti Matr. 945232

Academic Year 2020–2021

Abstract

Javascript is the most used programming language for the development of dynamic client-side web pages. With the increase in popularity of Node.js, a run-time environment which executes JavaScript code outside a web browser, it's now a valid choice also as a back-end language for web servers and non-browser applications. This study focuses on prototype pollution vulnerability, a new type of security vulnerability, first discovered in 2018, that has not been studied in depth. The vulnerability exploits the prototype oriented design of JavaScript. By modifying the prototype of native objects such as Object, from which most other objects inherit properties and methods, it's possible, depending on the logic of the specific application, to escalate to almost any other web vulnerability.

The purpose of this thesis is to evaluate static analysis techniques that identify the vulnerability, analyzing existing static analysis tools and testing them on known vulnerable functions extracted from real-world projects. In particular, we explain why a certain tool cannot identify a vulnerability case, if it's due a specific pattern implementation, engine limit, known limitations of static analysis or exclusive to prototype pollution. We conduct some case studies on relevant examples that provide interesting results.

Starting from the analysis of false negatives, we present some detection avoidance techniques. We also describe the root causes for false positives. As the goal is to improve the detection techniques, our findings are reported to the maintainers of each project.

Sommario

JavaScript è il linguaggio di programmazione più usato per lo sviluppo di pagine web dinamiche lato client. Negli ultimi anni viene spesso scelto anche per il lato server e programmi desktop utilizzando Node.js che consente l'esecuzione di codice JavaScript al di fuori del browser. Prototype Pollution è una vulnerabilità, su cui c'è ancora poca ricerca, che affligge esclusivamente JavaScript per del via del paradigma prototipale che sta alla base del suo design. Questa nuova vulnerabilità, scoperta solo nel 2018, si basa sulla manipolazione del prototipo degli oggetti nativi come Object, da cui ereditano quasi tutti gli altri oggetti, che avranno quindi accesso alle proprietà modificate. Di conseguenza, a seconda alla logica della specifica applicazione, è possibile sfruttarla per effettuare attacchi riguardanti la maggior parte delle vulnerabilità web.

Questa tesi mira ad analizzare e valutare le tecniche di analisi statica per individuare questa vulnerabilità. Si presentano alcuni software esistenti che offrono questa funzionalità e si testano su esempi di funzioni vulnerabili estratte da varie librerie JavaScript. Per verificare che gli esempi collezionati siano effettivamente affetti da prototype pollution scriviamo semplici script che simulano l'attacco. Analizziamo per quale motivo un determinato tool non riesca a individuare una vulnerabilità. Spiegando se dovuto a un limite del pattern, dell'engine specifico, per limitazioni classiche dell'analisi statica o esclusive alla prototype pollution. Analizziamo più nel dettaglio alcuni casi di studio che permettono di ottenere risultati interessanti.

Partendo dall'analisi dei falsi negativi riscontrati, si mostra come mettere in difficoltà i tool, creando alcuni esempi di vulnerabilità che non vengono individuati. Successivamente vengono discusse anche le principali cause che portano a falsi positivi. Poiché l'obiettivo di questo lavoro è migliorare gli attuali approcci di analisi statica, comunichiamo i risultati degni di nota agli sviluppatori di ogni tool considerato.

Contents

List of Listings

Chapter 1

Introduction

JavaScript was originally designed to be embedded in web browsers as a scripting language for writing simple scripts that enhance client-side web pages. It has quickly become the dominant programming language for dynamic web app, it is now used to create complex client-side web applications supported by all modern web browsers. With the increase of popularity of Node.js, which is a run-time environment that allows JS applications to be executed outside a web browser, JS has become popular also as a back-end language for web servers and for non-browser applications.

Prototype Pollution is a vulnerability affecting JavaScript due to its prototype based nature. Objects inherit properties and methods from their prototype. The prototype is also an object, and therefore it has a prototype and so on, this sequence is called prototype chain. Properties in `Object.prototype` are available to almost all the objects in an application since it is the first non-null object in the prototype chain. If `Object.prototype` is modified by an attacker then all the objects will have access to the tampered property.

Prototype Pollution is a relatively new vulnerability (first discovered in 2018 by Arteau [1]) and it's unique to JS, however it can be considered a subtype of an "object injection attack" (e.g. deserialization of untrusted data in PHP applications). The vulnerability also has its own Common Weakness Enumeration ID (CWE-1321) and it's defined as: ”Improperly Controlled Modification of Object Prototype Attributes" [2]. There are two main code patterns that allow this attack. The first is a recursive merge operation that copies all properties of a source object to another target object. The second is an operation that defines properties of an object based on a given path. Prototype pollution vulnerability is still unknown to most developers and also massively underrated. Once a prototype pollution is found, depending on the logic of the specific application, it's possible to escalate to almost any web vulnerability (RCE, injections, XSS, Denial-of-service, etc.). There are over 150 confirmed publicly known prototype pollution vulnerabilities with an assigned Common Vulnerabilities and Exposures

(CVE) ID. [3]
This work aims to improve the ability of existing tools to detect the vulnerability. Most existing approaches to detect this vulnerability are usually based on dynamic analysis (scanner/fuzzer), which don't perform advanced code analysis but only try to pollute prototypes using known payloads. [4] [5]
These approaches are not very interesting from a detection perspective, therefore we limit the research on static analysis only (i.e. examining the source code without executing the program). We present the current state-of-the-art static analysis tools and the approach adopted to identify prototype pollution bugs. Specifically we consider Semgrep, [6] an open source tool that performs abstract syntax tree (AST) based static analysis, which doesn't offer complex data flow analysis but prioritize localized analysis and execution times. CodeQL, [7] a powerful proprietary semantic engine that performs many advanced types of analysis, it creates a database from the codebase and then finds vulnerabilities by running query against it. ODGen [8] and ObjLupAnsys [9] are experimental static taint analysis tools based on a new kind of graph called Object Dependence Graph (ODG). This approach improve the detection of vulnerabilities influenced by the prototype chain.
In order to find vulnerable code, we gather JS code from various sources: GitHub Advisory Database, Snyk Vulnerability Database, existing prototype pollution examples and CTFs. We write simple proof of concept exploits for each function in order to be 100% certain that the collected examples are actually vulnerable. We run the static analysis tools considered against the vulnerable applications. Furthermore, we show and explain different code patterns that lead to the vulnerability, performing a case study on the most interesting examples from our dataset. We analyze the results gathered with a stronger focus on false negatives: we show there are code patterns that introduce the vulnerability which no tool is able to identify and present also some detection avoidance techniques. Since the goal is to improve the current static analysis tools, the meaningful results of this work regarding each tool are reported to the maintainers creating issues on the respective GitHub repositories.

The rest of the thesis is organized as follows:

- Chapter 2: In this chapter is described the technical background required to understand this thesis work: first we describe the features of JavaScript that make prototype pollution attacks possible, then we describe in detail the vulnerability. We also give an introduction on dynamic and static analysis. Finally, the state-of-the-art of the researches relevant for this work are briefly presented.

- Chapter 3: In this chapter are analyzed the static analysis tools considered (Semgrep, CodeQL, ODGen and ObjLupAnsys) and their approaches to detect prototype pollution.

- Chapter 4: In this chapter we explain our approach to this work. In particular how we collected vulnerability examples.
- Chapter 5: In this chapter are presented case studies extracted from our dataset. For each case we describe in detail why the vulnerability is present and why a tool can or cannot identify it.
- Chapter 6: In this chapter is described the evaluation of the results. Starting from the analysis of false negatives some detection avoidance techniques are presented. Finally, false positives cases are discussed.

Chapter 2

Background and Related Work

2.1 JavaScript

JavaScript (JS) is a prototype-based, multi-paradigm, single-threaded, dynamic language, supporting object-oriented, imperative, and declarative (e.g. functional programming) styles [10]. JavaScript code is executed by JavaScript engines, modern engines use just-in-time compilation for performance reasons instead of interpreting the code. The most popular engines are Google's V8 engine (used in Chromium-based browsers) and Mozilla's SpiderMonkey (used in Firefox browser). JS is the dominant scripting language for web pages: over 97% of websites use it client-side [11]. With the creation of Node.js, which is a run-time platform for JS applications that relies on the V8 engine and executes programs outside a web browser, JS has become popular also for server-side website development and non-browser applications.

Prototype Chain

The prototype pollution vulnerability is unique to the JavaScript language. Therefore, before analyzing the actual vulnerability, we need to understand the features that make it possible.

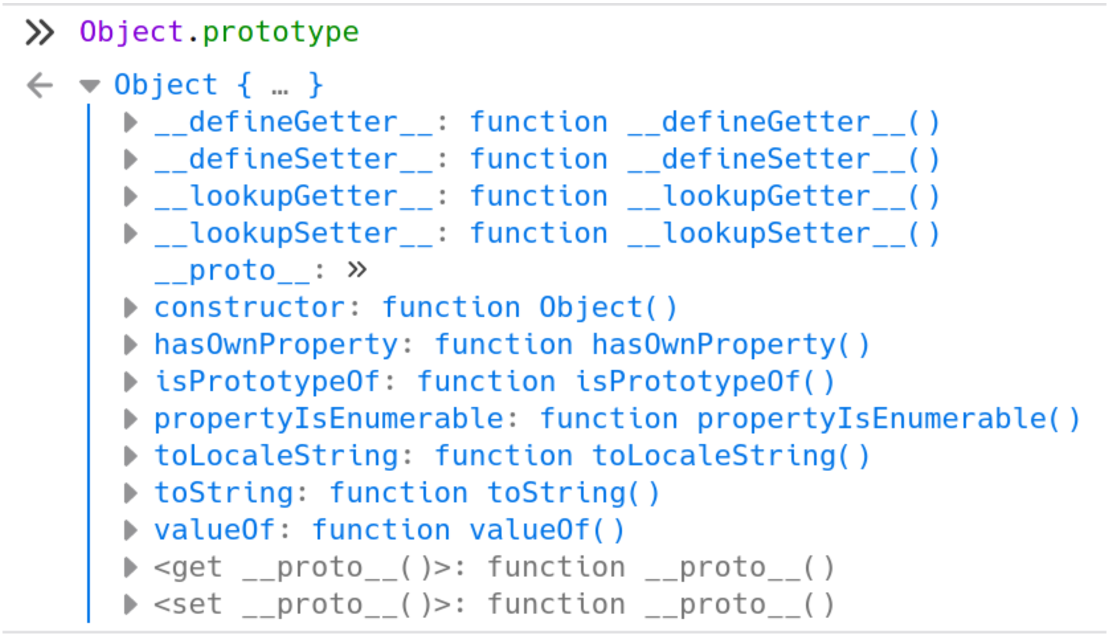
JavaScript is a class-free, object-oriented language, it uses prototypal inheritance instead of classical inheritance. Almost any entity in JS is an object: arrays, functions, class defintion. This can be confusing to programmers trained in class-based object-oriented languages like C++ or Java.

In order to understand how it works, in listing 2.1.1, we declare object `o` with properties `color` and `length`.

As expected when we access a non-existent property we get the value `undefined`, yet the `o.tostring()` method, which we have not defined, is available.
If property is not found on the object it's searched in the **prototype chain** [12]. Every object has a private property which holds a link to another object called its

```
var o = {color: 'blue', length: 12}
//access properties
o.color         <- 'blue'
o["length"]     <- 12
o.size          <- undefined
o.toString()    <- '[object Object]'
```

Listing 2.1.1: Declaring an object and accessing its properties



```
» Object.prototype
← ▾ Object { … }
    ▸ __defineGetter__: function __defineGetter__()
    ▸ __defineSetter__: function __defineSetter__()
    ▸ __lookupGetter__: function __lookupGetter__()
    ▸ __lookupSetter__: function __lookupSetter__()
      __proto__: »
    ▸ constructor: function Object()
    ▸ hasOwnProperty: function hasOwnProperty()
    ▸ isPrototypeOf: function isPrototypeOf()
    ▸ propertyIsEnumerable: function propertyIsEnumerable()
    ▸ toLocaleString: function toLocaleString()
    ▸ toString: function toString()
    ▸ valueOf: function valueOf()
    ▸ <get __proto__()>: function __proto__()
    ▸ <set __proto__()>: function __proto__()
```

Figure 2.1: Object.prototype in DevTools

prototype. The prototype itself is an object, therefore, it has a prototype, and so on, until we find an object with a null prototype. By definition, `null` has no prototype, so it's the terminal link in the chain.
When an object is created, its prototype is assigned to `Object.prototype`. In fact, almost all objects in JS are instances of `Object` which comes just after `null` on the top of a prototype chain. As we can clearly see in figure 2.1, the `toString()` method is actually defined in `Object.prototype`.

When reasoning about prototypes in JavaScript we may refer to different entities:

- `[[Prototype]]` internal property: It is used to designate the prototype of an object, we can access it through special functions (as showed in listing 2.1.2) like `Object.setPrototypeOf()`, `Object.getPrototypeOf()` or with `__proto__` which is non-standard but de-facto used by most engines.

- `prototype` property of functions: like `function.prototype`, `Object.prototype`, which instead specifies the `[[Prototype]]` to be assigned to all instances of objects created by the given function when used

```
o.__proto__ === Object.Prototype             <- true
o.__proto__ === o.constructor.prototype      <- true
Object.getPrototypeOf(o) === o.__proto__     <- true
//For the object o the prototype chain is short
o.__proto__.__proto__                        <- null
```

Listing 2.1.2: Different ways of accessing the prototype

```
//add height property to o
o.height = 4
//now object o is:
Object { color: "blue", length: 12, height: 4 }
```

Listing 2.1.3: Modify an object

as a constructor. The `Object.prototype` property represents the `Object` prototype object

- `__proto__` property: it is a getter/setter and only gets a reference to the prototype of the object and returns it.

Objects in JavaScript are mutable, it's possible to modify them, giving them new fields and methods as showed in listing 2.1.3.
What happens if we modify `Object.prototype` as in listing 2.1.4?
Changes to the `Object.prototype` object are seen by all objects unless:

1. The properties and methods subject to those changes are overridden further along the prototype chain

2. The object doesn't inherit from `Object`

In the static object-oriented languages, if you want an object which is slightly different from another object, you need to define a new class. In JavaScript, you can add methods to individual objects without the need for additional classes. For instance, we can create an animal object with the fields and methods of a generic animal. Then, if we want to define a particular species, e.g. a racoon, the new racoon object is "cloned" from the animal object, properties specific to racoons are appended or overridden. Then individual racoons objects are created/cloned from the generic racoon object.
Note that a constructor in JavaScript is just a function that happens to be called with the `new` operator, `constructor` is a property which returns the function used to create the object. The prototype object has a constructor that points to the function itself.
ECMAScript 2015 introduced new keywords like: `class`, `extends`, `super`, but it's just syntactical sugar, JS remains prototype-based. [13]

```
Object.prototype.weight = 100
//now all objects that inherit from Object
//will have the weight field set to 100
o.weight            <- 100
let y = {}
y.weight            <- 100
let z = {weight: 50}
z.weight            <- 50
[1,2,3].weight      <- 100
z.__proto__.weight  <- 100
```

Listing 2.1.4: Changing Object prototype

2.2 Prototype Pollution

Prototype pollution is a term that was used to describe how libraries like Prototype.js [14] work. That is by adding many convenience methods to the prototype of native objects such as `Object`, `Array`, `Function`. This approach introduced bugs in applications and was abandoned because considered a bad practice. These libraries are not used any more.

What if `Object.prototype` is tampered by a malicious user?

If there is a bug that make it possible then a prototype pollution vulnerability occurs. Olivier Arteau in 2018 [1] was the first who discovered and analyzed this vulnerability.

General problem

Let's consider an expression in this form: `obj[a][b] = val`. If an attacker can control `a` and `val`, then she can set the value of `a` to `__proto__`. Therefore, the property `b` will be defined for all existing objects of the application with the value `val`.

Similarly, in an expression like `obj[a][b][c] = val`, the attacker can obtain the same result by assigning `constructor` to `a` and `prototype` to `b`. If `obj` is not an instance of `Object` it's possible to achieve the same result by going up the prototype chain. (e.g. `__proto__.__proto__`).

While it is very rare to find code that looks textually like the examples, there are some utility functions that can lead to a similar situation. Arteau identifies three main types of API that can be affected by the vulnerability:

1. Object recursive merge
2. Property definition by path
3. Object clone

```
function merge(target, source) {
    for (var attr in source) {
        if (isObject(target[attr]) && isObject(source[attr])) {
            merge(target[attr], source[attr]);
        } else {
            target[attr] = source[attr];
        }
    }
    return target
}
```

Listing 2.2.1: Example of object recursive merge implementation

```
var a = {};
var b = JSON.parse('{"__proto__":{"admin":true}}');
merge(a,b);
Object.prototype.admin //true
```

Listing 2.2.2: Merge Attack example

Object recursive merge/clone

The merge operation copies all properties of a source object to another target object. Listing 2.2.1 contains a vulnerable implementation.
For each attribute of the source object:
If the attribute exists, and it is an object on both target and source then recursively merge these objects.
Otherwise copy the attribute to the target object.

If the source object contains a property named `__proto__` defined with `Object.defineProperty()` (usually this happens when the input is parsed with `JSON.parse()`) the condition that checks if the property exists and is an object on both the target and the source will be true and the merge will continue with the target now being `Object.prototype` and the source an object defined by the attacker. Therefore, attributes defined by the attacker will be copied on `Object.prototype`, listing 2.2.2 shows how to perform the attack. As mentioned before this works because `JSON.parse` treats `__proto__` like any other property, overriding the standard prototype accessor functionality (examples are included in listing 2.2.3).

This merge implementation bug was found in some popular JS libraries such as lodash, Hoek, jQuery. A clone API is vulnerable as well when implemented as a recursive merge on an empty object (listing 2.2.4).

```
var a = {__proto__: 1};
var b = JSON.parse('{"__proto__": 1}');
const c = {};
Object.defineProperty(c, '__proto__', {value: 42});
a.__proto__ //Object.prototype
b.__proto__ //1
c.__proto__ //42
```

Listing 2.2.3: The `__proto__` property can be overridden only by special functions

```
function clone(obj) {
  return merge({}, obj);
}
```

Listing 2.2.4: Vulnerable object clone

Property definition by path

There are some JS libraries that use an API to define property values on an object based on a given path. The vulnerable functions have generally this signature: `func (obj, path, val)`. If the attacker can control the value of `path`, she can set this value to `__proto__.x`. The property `x` is then added to the prototype of the class of `obj` with value `val`.
As an example, listing 2.2.5) reports a simplified version of the vulnerable set function in the package dot-object [9]. At Line 9, `key` equals to `__proto__`, `k` to `toString` and `val[k]` to `exploit`. Therefore, `Object.prototype.toString` is polluted. This bug was found in many libraries such as lodash, pathval and js-data.

Impact

Prototype pollution is an interesting vulnerability, either it is server-side or client-side. Depending on the logic of the specific application, once a prototype pollution is found it can lead to practically all web vulnerabilities.
Denial-of-service (DoS) attacks can be very easy to perform. `Object` holds generic functions that are implicitly called for various operations (e.g. `toString` and `valueOf`), the attacker could pollute `Object.prototype.toString`, redefining it as a number, if the method is called later, which is very likely, the application won't work properly.
Remote code execution (RCE) on NodeJS applications has been the main goal server-side: for instance Arteau discovered it in Ghost CMS [1], Bentkowksi in kibana. [15]
Soncino managed to perform a SQL injection in the TypeORM package. [16]
Basically every time a property that is not defined on the same object it is ac-

```
function set (path, val, obj){
    var i,k,keys,key;
    keys = parsePath(path, '.');
    for (i = 0; i < keys.length; i++){
        key = keys[i];
        if (i === keys.length - 1){
            for (k in val){
                if (hasOwnProperty.call(val, k)){
                    obj[key][k] = val[k];
                }
            }
        }
        ...
    }
    return obj;
}
var path = "__proto__";
var val = {toString : "exploit"};
set(path, val, {});
```

Listing 2.2.5: Dot-object set function

cessed, the attacker can change the evaluation of the check by prototype pollution. This can happen with For-loop pollution: every time a loop in the form `for (var key in obj) { ... }` is used, it will also iterate over polluted properties.

Another case is property injection: the attacker pollutes properties used for security like cookies, tokens or privilege flags. For example the NodeJS http module supports multiple headers sharing the same name, which, when parsed, are concatenated together and comma separated. If the attacker pollutes the key `cookie`, the value of `request.headers.cookie` will always start with the polluted value, leading to a variant of a session fixation attack where everyone querying the server will share the same session.

On the client-side this vulnerability is usually escalated to reflected XSS. The JavaScript code that can be used to escalate prototype pollution to other vulnerability is called a gadget. Bentkowksi proves that prototype pollution can lead to bypass a lot of popular HTML sanitizers. [17] These libraries take an untrusted HTML input and delete all tags or attributes that could introduce an XSS attack. They're based on a whitelist approach: they have a list of tags and attributes that are allowed. If this list is stored in an object (as in listing 2.2.6), then it is easily exploitable by polluting the prototype: `Object.prototype.SCRIPT = true;`. Therefore `ALLOWED_ELEMENTS["SCRIPT"]` returns true, and the script tag can be used by the attacker to perform XSS.

```
const ALLOWED_ELEMENTS = {
 "h1": true,
 "i": true,
 "b": true,
 "div" :true
}
```

Listing 2.2.6: Whitelist stored in an object

Also, a lot of nested parameter parsers (e.g. functions that deserialize a query string into an object) are vulnerable to prototype pollution.

Mitigation

There are different ways to mitigate this vulnerability, some applications are vulnerable by design, so the only way to fix it is to validate the user input.

- The most used approach is to blacklist the following keywords: `__proto__`, `prototype`, `constructor`.
  The deny list check must be done just before the property is added to the object to avoid workarounds as it happened with Jira: the devs fixed the issue by blacklisting, but a function removed the square bracket characters from the key after the check, therefore a key like `__pro[]to__` would bypass the blacklist. [18] This method is easy to implement and usually enough to fix the vulnerability, however it does not eliminate the problem because there is still the possibility of changing prototypes.

- Another possibility is to require schema validation of JSON input.

- It's possible to create objects that don't have any prototype using `Object.create(null)`. However, for example, calling `toString()` on this object won't work unless the function is re-defined for the specific object, this may break some apps as these methods are often used.

- When a key/value structure is needed, it is a best practice to use `Map` instead of `Object`, it works like a hash-map but without all the security problems of `Object`. [19]

- It's also possible to freeze the prototype using the `Object.freeze()` function. After calling `Object.freeze(Object.prototype)`, the `Object.prototype` cannot be modified. However, dependencies that rely on modifying the prototype may introduce bugs which are hard to identify. Also, the prototype of other objects like `Array.prototype` should be frozen too.

- In Node.js, it's possible to use the flag `--disable-proto` which disables the `Object.prototype.__proto__` property.

2.3 Code Analysis

Dynamic analysis is the analysis of the properties of an application at run-time. Static analysis, instead, is the process of analyzing a program by examining its source code without executing it. Static and dynamic approaches have different strengths and weaknesses: usually the best way to detect vulnerabilities is a hybrid approach.

Static Analysis

There are various techniques to analyze source code statically: data flow analysis, abstract interpretation, symbolic analysis [20].
Most of these techniques were first applied to design compilers, but they are used a lot in the cybersecurity space: malware analysis, reverse engineering and of course to find vulnerabilities.

- Data flow analysis is used to collect run-time (dynamic) information about data in software while it is in a static state [20].
- A control flow graph (CFG) is a directed graph which models all paths that can be traversed during the execution of a program. Edges represent control flow paths and the nodes basic blocks (i.e.: sequence of instructions without branches).
- An abstract syntax tree (AST) is a representation of the abstract syntactic structure of source code. Each node represents a construct occurring in the source code.
- Taint analysis attempts to identify the flow of user input through the application. Variables that have been "tainted" (i.e. subject to modifications by the user) are traced until they are used in potential vulnerable functions (sinks).

Static Application Security Testing (SAST) Tools, can help analyze source code or compiled versions of code to help find security flaws [21].
A typical static tool analysis is based on the following steps:

1. Translate source code into an intermediate representation (IR): (e.g. AST)
2. Perform multiple types of analysis to identify the tainted sources and dangerous sinks

3. Output results to be reviewed, specifying the exact locations of the vulnerabilities in the source code. SARIF (Static Analysis Results Interchange Format) is an open source format which was developed to standardize the output of these tools.

The main strengths of static analysis are:

- Scalable: Can be run on lots of software and it's easy to automate.
- Make it easier for devs to fix the vulnerability as it provide the exact line of code of the vulnerability. The feedback is typically fast and early in the Software Development Life Cycle (SDLC).
- Excellent code coverage

The main weaknesses of static analysis are:

- Some types of security vulnerabilities are very difficult to find automatically: for instance it's impossible to identify vulnerabilities which exist only in the deployment environment.
- High numbers of false positives: it may fail to understand that some paths from sources to sinks are not exploitable.

Dynamic Analysis

With dynamic analysis we can see what the code actually does. We can more accurately simulate how a malicious user would try to attack an application.
A common type of dynamic tools are fuzzers: programs which inject automatically data (could be carefully designed by the tester like a list of known payloads or just random data) into a program/stack and detect bugs. Many JavaScript tools also make use a headless browser (e.g. Burp suite). Tools that scan web applications, normally from the outside, to look for security vulnerabilities are also known as web application scanners [22].
The main strengths of dynamic analysis are:

- Can identify timing and environment-related issues.
- In JavaScript, dynamic analysis, has proven to be effective due to the many dynamic language features [23]
- Less prone to false positives: it actually observes the data being propagated from source to sink during execution

The main weaknesses of dynamic analysis are:

- Cannot generate and test all possible inputs in reasonable time
- Limited code coverage
- More prone to false negatives: malicious data may not reach a sink due to the current state of the program. It's hard to consider all possible cases.

2.4 Related work

In the following section, the state of the art of the researches that are relevant for this work are briefly presented.
As prototype pollution is a new vulnerability there are just a handful academic papers and few blog posts by security researchers about it.
The first ever article about prototype pollution is from Oliver Arteau. [1]
He discovered the prototype pollution vulnerability in some node.js packages, including one in lodash packages (CVE-2018–3721). This package is used in a lot of popular packages including Ghost CMS, in which Arteau exploited the vulnerability to gain unauthenticated Remote Code Execution (RCE).
Arteau also created a dynamic vulnerability analysis tool that could detect a prototype pollution vulnerability in Node.js modules. This tool performs fuzz testing on all the functions with pre-defined input. It first executes the function with malicious payloads and then checks if the prototype is polluted.
There are some other **dynamic tools** (scanner/fuzzer) that focus on client-side prototype pollution. These tools only exploit known gadgets (usually taken from a collection hosted on GitHub [24]), they don't perform code analysis or any advanced prototype pollution exploitation.

- **PPmap** [4] Scanner which automatically exploits known and existing gadgets (checks for specific variables in the global context) to perform XSS via Prototype Pollution. (written in go)
- **PPFuzz** [5] Tool that attempts to check for prototype-pollution vulnerabilities by adding an object & pointer queries. It fingerprints the script gadgets used and then displays additional payload info that could potentially escalate its impact to XSS, bypass or cookie injection. (written in rust)
- **ProtoScan** [25] Another Prototype Pollution Scanner, similar to the previous tools. (written in go)
- **PPScan** [26] Chrome extension that runs in background while the user visits the webpage. It also searches JS resources with known patterns that lead to prototype pollution using regular expressions.

Kim et Al. [27] propose patterns that can identify prototype pollution vulnerabilities. They created **DAPP**, a tool to detect the vulnerability in Node.js packages using static analysis based on abstract syntax tree and control flow graph.

Li et Al. present a novel graph structure, called Object Dependence Graph (ODG). ODG represents JavaScript objects as nodes and their relations with Abstract Syntax Tree (AST) as edges, and accepts graph queries (especially on object lookups and definitions) for detecting Node.js vulnerabilities (not only prototype pollution)[8]. This work extends **ObjLupAnsys** [9] from the same authors which is a flow-, context-, and branch-sensitive static taint analysis tool to detect prototype pollution vulnerabilities [28]. The implementation is called **ODGen** [29].

A group of security researchers [18] explained their methodology to detect a prototype pollution vulnerability, identify the vulnerable functions and then find script gadgets to achieve XSS to report in bug bounty programs. Some of the tools they created [30] are:

- Match rules for Burp Software Version Reporter extension that passively detect vulnerable libraries even in minified JS code.
- Prototype Checker: JS script that highlights custom fields in prototypes and constructors that can be useful in exploiting Prototype Pollution.
- Burp + pollute.js: script that highlights access to uninitialized properties using code instrumentation. It can be used to replace all Burp Proxy HTTP responses with modified code.

They also show that prototype pollution can be searched with CodeQL queries. Existing CodeQL queries do not cover all variants though.

For what concern static analysis tools there are three official Semgrep rules to detect prototype pollution [31], also there are three official CodeQL queries [32]. These tools are analyzed in detail in the next chapter.

Chapter 3

Overview of Analyzed Tools

3.1 CodeQL

CodeQL [7] is a proprietary semantic code analysis engine used to detect patterns in the code, typically used to find security vulnerabilities, by querying code as if it were data. These patterns are modelled as queries written in the QL [33] language that are executed against databases extracted from the codebase to analyze. There are open source query codebases made available by the community and Semmle/GitHub (the company behind the project).

3.1.1 CodeQL overview

The CodeQL CLI includes all the commands needed to create CodeQL databases, develop queries and to analyze databases.
CodeQL consists in three main components:

- **CodeQL engine**: it is the core of the tool and can be divided in three modules:
  - The extractor parses the code to perform lexical analysis, to generate an Abstract Syntax Tree (AST) and also creates Control Flow Graphs (CFG) and Data Flow Graphs (DFG).
  - The compiler receives queries written in QL as input and outputs compiled and optimized query in terms of relational algebra (RA) or the intermediate DIL (Datalog Intermediary Language) format.
  - The evaluator runs the compiled queries against the database and generates the results.
- **Database**: A CodeQL database is a directory which stores all the data about the source code needed to perform the analysis. This is queryable

data, extracted from the code: ASTs, CFGs, DFGs, references to source code (for displaying query results directly in the code). It also contains query results and log files generated during operations such as database creation and query execution. A QL database schema (different for each language) describe column types and extensional relations of the dataset. There is a table for every language construct, for instance an expressions table containing a row for every single expression in the source code.

- **Queries**: The patterns of the code to identify are defined in CodeQL queries written in the QL language. CodeQL queries can be divided in two main categories:
  - Alert queries that search for issues in specific locations in your code.
  - Path queries which describe the flow of information from source to sink.

  The syntax of QL is similar to SQL, the semantics are based on Datalog. The main properties of the QL language are that it is logical (all operations are logical), declarative, object-oriented and read-only.

CodeQL analysis consists of three steps (figure 3.1):

1. **Creating a CodeQL database**: For compiled languages, extraction must observe the build process. For interpreted languages like JavaScript, the extractor runs directly on the source code. For multi-language codebases, databases are generated one language at a time. After extraction, all the data required for analysis is imported into a single directory which is the database.

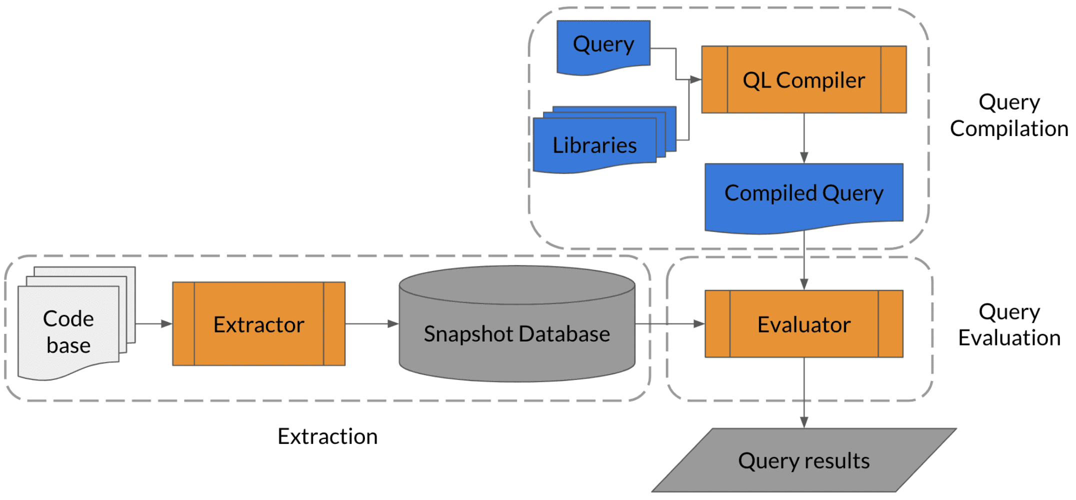


Figure 3.1: CodeQL Analysis overview diagram

2. **Executing queries**: One or more queries are compiled and then executed against the database.

3. **Interpreting the query results**: A binary query result set (BQRS) file is the binary representation of the raw result of a query. The results are exported and converted in one of the supported formats such as csv or sarif. The output can be simply the location in the code of the identified problem or more complex messages that shows multiple steps in a data/control flow path.

CodeQL supports both local and global taint tracking. CodeQL documentation make a distinction between "normal" data flow and taint tracking. [34] The normal data flow libraries are used to analyze the information flow in which data values are preserved at each step. Taint tracking enhances data flow analysis by including also steps in which the values may change, but the potentially insecure object is still propagated. CodeQL can also perform other types of analysis such as type tracking and range analysis.

3.1.2 CodeQL Queries

CodeQL libraries for each language define classes which provide a layer of abstraction over the database tables. This way the user has an object-oriented view of the data which makes it easier to write queries.

A query file has the extension .ql and contains a query clause composed by: a from clause where variables are declared, a where clause in which the actual logic of the analysis is written and a select clause where the CodeQL expressions are written. A query file may include other QL constructs like predicates, classes, and modules.

A query library has the extension .qll and does not contain a query clause, but may contain modules, classes, and predicates. Using import statements the classes and predicates defined in a module can be used in another one.

Listing 3.1.1 includes QL code for a template to create a taint tracking path query [35]. The idea is to create a taint tracking configuration class in which we define a set of relevant sources and sinks; then we "filter" or add additional set of flow edges defining functions like isAdditionalTaintStep or isSanitizer.

The query results are flow paths which:

1. Start in a node matched by isSource.

2. Step through variables, function calls, properties, strings, arrays, promises, exceptions, and steps added by isAdditionalTaintStep.

3. End at a node matched by isSink.

```
import javascript
import DataFlow
import DataFlow::PathGraph

class MyConfig extends TaintTracking::Configuration {
  MyConfig() { this = "MyConfig" }
  override predicate isSource(Node node) { ... }
  override predicate isSink(Node node) { ... }
  override predicate isAdditionalTaintStep(Node pred, Node succ) { ... }
}

from MyConfig cfg, PathNode source, PathNode sink
where cfg.hasFlowPath(source, sink)
select sink.getNode(), source, sink, "taint from $@.", source.getNode(), "here"
```

Listing 3.1.1: CodeQL taint tracking path query template

CodeQL queries are open source, so you can build on the work already done by the community. The QL language however has a steep learning curve: even simple things may not be easy to code.

Prototype Pollution Queries

There are three queries among the CodeQL JavaScript official repository that target prototype pollution vulnerability. [32] The query files and the linked library files are too long (hundreds of lines) to be included here. These queries are also among the most complex available in the official JavaScript queryset. [36]

- **Prototype Polluting Assignment Query**:
  This taint-tracking query is about finding potentially vulnerable assignments. It targets simple direct assignment (e.g. `obj[a][b]=val` and more complex pattern which are usually included in "property definition by path" methods (e.g. listing 2.2.5). The query performs filtering to prevent false positives cases such as paths that start with user inputs and end with a write to a fixed property or if the polluting keywords are blacklisted e.g. `x !== "__proto__"` or `x.includes("__proto__")`.

- **Prototype Polluting Function Query**:
  The rule targets functions that recursively assign properties on objects which are typically recursive deep merge or deep assign implementations (listing 2.2.1). In particular it tracks three separate data flow paths originating from the same property enumeration, all leading to the same dynamic property write (i.e. an assignment in the form of `target[key] = source[key]`). Like the previous query, it also filters out cases in which prototype pollution mitigation techniques are used e.g. creating an object without a prototype using `Object.create(null)`.

- **Prototype Polluting Merge Call Query**:
  It is the least interesting query as it does not perform any advanced code analysis. The query identifies calls to known vulnerable merge or extend functions when they receive as input user controllable objects. The functions' names, including the package name and the version, are defined in the PrototypePollutionCustomizations.qll library file. This query may be useful when a user is using an application which imports Node.js packages using npm, even though the `npm audit` command already warns the user about CVE in the installed libraries.

3.2 Semgrep

Semgrep [6] is a free and open source static analysis tool that works on more than 20 programming languages including JavaScript. The name is a combination of semantic and grep (the program which searches files for lines containing a match to a specified pattern), implying that semgrep is a sort of text search utility able to understand the source code semantics. For instance if we want to search for a method named `foo`, grep cannot understand when it is a function, a variable or just text in a comment, while semgrep can.

3.2.1 Semgrep overview

Semgrep consists of a python CLI (semgrep) built around an engine (semgrep-core) written in OCaml which performs the actual parsing/matching work.
Semgrep analysis can be divided in five steps:

1. **Input**: Semgrep accepts multiple rules that contain the patterns to check against multiple source code files
2. **Parsing**: semgrep-core relies on external modules to parse code into augmented language-specific ASTs. Note that these ASTs contains also information that is usually only present in concrete syntax trees (CSTs).
3. **Converting to generic AST**: All language-specific ASTs are converted to a generic AST which is used by semgrep-core to match patterns
4. **Matching**: The matching functions visit the target AST and try to match the pattern for each rule.
5. **Output results**: The results of the match are formatted (in text or sarif or other formats) and outputted.

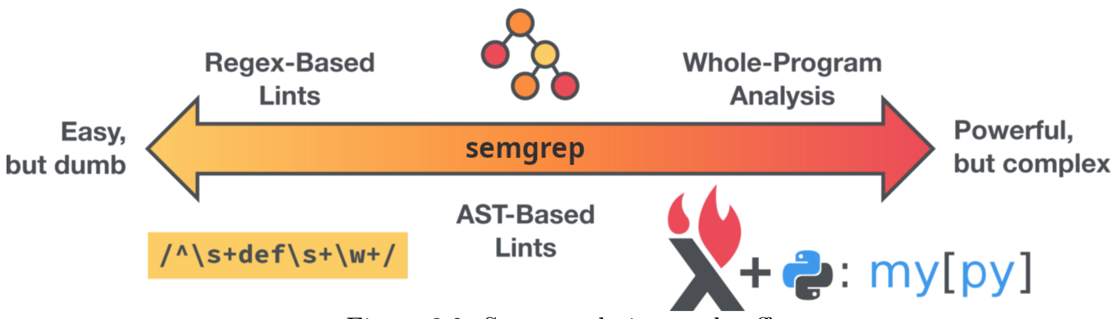


Figure 3.2: Semgrep design trade-off

As showed in figure 3.2, the semgrep engine design is a trade-off between the power of complex static application security testing (SAST) tools and the ease of grep and regex-based linters. [37]

Semgrep analyzes one file at a time, thus it misses bugs related to multiple files such as a constant string defined in another file. It is advertised as a lightweight and fast static analysis tool as files can be processed in parallel, and it doesn't require building the code (for compiled languages). In Semgrep the rules are directly ran against the codebase without the need to create a special database, however it doesn't have some advanced and complex features like interprocedural dataflow analysis that CodeQL does.

Semgrep is more powerful than typical linters which usually focus on a single line of code bugs or formatting/stylistic decisions. Semgrep doesn't care about whitespace or formatting and can detect vulnerabilities that require reasoning over multiple lines. Semgrep has also a limited intra-procedural dataflow analysis engine that adds additional features such as constant propagation and taint tracking.

3.2.2 Semgrep Rules

One of semgrep selling points is that it “lets you reason about your analysis the way you reason about your code”. The rules look like the code of the language you are writing in, which makes it easier to start writing rules. Tools like CodeQL on the other hand have a steeper learning curve: you have to invest a lot of time to learn it. [38]

Trivial one line rules can be passed directly to the semgrep CLI as an argument, more complex patterns are written in YAML files that must follow a syntax schema. In each rule must be specified which programming language syntax to match, which patterns the tool should consider, which to exclude, and also whether the target is inside of a specific code pattern or not. In taint tracking mode is possible to specify the source patterns, the sinks, and sanitizers. Also each rule has an ID, a message that explain what is the problem and suggestions about how to fix, a severity level (info, warning, or error).

The main features of semgrep rules are:

- The ellipsis operator (`...`): represents a sequence of zero or more arguments, statements, characters, etc.
- Metavariables `$X`: used to track values (variables, functions, arguments, classes, etc.) across a specific code scope.
- Smart matching: for instance the pattern `function ...($X) { ... }` matches every function with one parameter, including lambdas.

Users can create their own rules but also use existing rulesets from the open source semgrep registry. There are multiple rule sets for many bugs and vulnerabilities written by the community and r2c (the company behind semgrep).

Prototype pollution rules

For what concerns prototype pollution there are three rules available on the semgrep registry. [31]
In this section we analyze how these rules try to detect the vulnerability.

- **Prototype Pollution Assignment Rule** (listing 3.2.1):
  The idea is to detect "two-step assignment" that may cause prototype pollution. The vulnerability is detected when an assignment (line 4) comes before or after another assignment (line 7). The first step should be something like `obj = tmp["__proto__"]` and the second `obj["prop"] = val`. The rule filters out cases when the assignment key is instead the index an array's element inside a for or forEach loop. It also excludes the cases when the name of the object property to be accessed, in this case the metavariables `$A` and `$B`, are hardcoded strings, template literals (i.e. delimited with backticks) or numbers (lines 14-25).

- **Prototype Pollution Function Rule** (listing 3.2.2):
  It is a taint-mode rule (note that it supports only intra-procedural taint tracking). The idea behind this rule is to detect recursive object merge implementations. It considers a sink any assignment to an object property (line 13) which is inside a function that has a recursive call (lines 25-33). The `pattern-sanitizers` field excludes cases when the polluting keys are checked using an `if` statement. The rule also filters out cases when the property assignment is controlled with the `hasOwnProperty` method. Again the rule does not report results if the assignment key are hardcoded strings, template literals or numbers (lines 11-13).

```
rules:
  - id: prototype-pollution-assignment
    patterns:
      - pattern: $X[$B] = ...
      - pattern-inside: |
          ...
            $X = $SMTH[$A]
          ...
      - pattern-not-inside: if (<...'constructor' ...>) {...}
      - pattern-not-inside: if (<...'__proto__' ...>) {...}
      - pattern-not-inside: for(var $B = $S; ...; ...) {...}
      - pattern-not-inside: for($B = $S; ...; ...) {...}
      - pattern-not-inside: $X.forEach(function $NAME($OBJ, $B,...) {...})
      - metavariable-pattern:
          metavariable: $A
          patterns:
            - pattern-not: '"..."'
            - pattern-not: `...${...}...`
            - pattern-not: ($A: float)
      - metavariable-pattern:
          metavariable: $B
          patterns:
            - pattern-not: '"..."'
            - pattern-not: `...${...}...`
            - pattern-not: ($B: float)
```

Listing 3.2.1: Semgrep prototype pollution assignment rule

```yaml
rules:
  - id: prototype-pollution-function
    pattern-sources:
      - pattern-either:
          - patterns:
              - pattern: $SOURCE[$B]
              - pattern-not: $SOURCE["..."]
              - pattern-not: $SOURCE[`...${...}...`]
              - pattern-not: $SOURCE[($B: float)]
          - pattern: function $X(..., $SOURCE, ...) { ... }
    pattern-sinks:
      - patterns:
          - pattern: $TARGET[$A] = ...
          - pattern-not: $TARGET["..."] = ...
          - pattern-not: $TARGET[($A: float)] = ...
          - pattern-not-inside: |
              if (<... $TARGET.hasOwnProperty($A) ...>) { ... }
          - pattern-either:
              - pattern-inside: |
                  $NAME = function $F(...) { ... $NAME(...) ... }
              - pattern-inside: |
                  function $NAME(...) { ... $NAME(...) ... }
              - pattern-inside: |
                  function $NAME(...) { ... $THIS.$NAME(...) ... }
              - pattern-inside: |
                  function $NAME(...) { ... $NAME.call(...) ... }
    pattern-sanitizers:
      - patterns:
          - pattern: if (<...'constructor' ...>) { ... } ...
          - pattern: if (<...'__proto__' ...>) { ...  } ...
```

Listing 3.2.2: Semgrep prototype pollution function rule

- **Prototype Pollution Loop Rule** (listing 3.2.3):
  The rule idea is to find code patterns that define property values on an object based on a given path. These functions are usually implemented using a loop that iteratively assigns the properties to an object in order to reach the specified path, then a value is assigned to it. The rule, however, does not explicitly flag the line in which the actual pollution occurs but only the iterative assignment (lines 5-8). Cases in which the assignment key `$A` represents the index of an array's element inside a for or forEach loop, are excluded (lines 14-17). Finally, also this rule discard results if the assignment key `$A` is a hardcoded string, template string or number (lines 18-23).

```
1  rules:
2    - id: prototype-pollution-loop
3      patterns:
4        - pattern-either:
5            - pattern: $SMTH = $SMTH[$A]
6            - pattern: $SMTH = $SMTH[$A] = ...
7            - pattern: $SMTH = $SMTH[$A] && $Z
8            - pattern: $SMTH = $SMTH[$A] || $Z
9        - pattern-either:
10           - pattern-inside: for(...) { ... }
11           - pattern-inside: while(...) { ... }
12           - pattern-inside: |
13               $X.forEach(function $NAME(...) { ... })
14       - pattern-not-inside: for(var $A = $S; ...; ...) {...}
15       - pattern-not-inside: for($A = $S; ...; ...) {...}
16       - pattern-not-inside: |
17           $X.forEach(function $NAME($OBJ, $A,...) {...})
18       - metavariable-pattern:
19           metavariable: $A
20           patterns:
21             - pattern-not: '"..."'
22             - pattern-not: `...${...}...`
23             - pattern-not: ($A: float)
```

Listing 3.2.3: Semgrep prototype pollution loop rule

3.3 ODGen and ObjLupAnsys

ODGen [8] and ObjLupAnsys [9] are experimental static analysis tools for JavaScript programs based on a new type of graph called Object Dependence Graph (ODG). The tools are developed by the same authors and share most of the source code.

3.3.1 Differences

ObjLupAnsys and ODGen generate the same ODG, but they use the generated graph in different ways. ObjLupAnsys uses real-time object lookup analysis by expanding object lookups and propagating taints during abstract interpretation to detect prototype pollution vulnerability. ODGen instead uses graph query. ODGen can also detect other vulnerabilities such as command injection, path traversal, and XSS.
For performance reasons, in the next release, ODGen will embrace ObjLupAnsys approach to detect prototype pollution vulnerability. [39]

3.3.2 Overview

ODG nodes are all JavaScript objects, variables and scopes, generated during abstract interpretation. Relations among objects (e.g. object-level data dependency) and between objects and AST nodes (e.g. object lookup) are the edges. This graph can successfully model object lookups based on prototype chain which are fundamental to detect prototype pollution vulnerabilities.

The analysis starts with the parsing of a target JavaScript program into an AST. Then, the core of tool, written in python, abstractly interprets each AST node edges performing ODG generation, taint and object lookup analysis. The tool also models Node.js built-in objects and functions. The current implementation support those that are used by more than 5% of packages. The tool can still analyze packages with unimplemented features skipping those parts.

The analysis of each AST node can be divided in three phases:

1. ODG generation: The tool abstractly interprets the target AST node and generate the ODG.

2. Taint analysis: Collects the conditions that are attached to object lookups for the target AST node. If on path all the collected constraints can be satisfied, the tool will propagate taints between objects.

3. Object Lookup Analysis: An object lookup (i.e. `obj[prop]`) may be vulnerable when `prop` is marked as tainted (controllable by the adversary). If also `obj` can be controlled by the user then the tool expands the source cluster (i.e. adds a new property node to the target source object). Otherwise, it

expands the sink cluster by adding conditional ODG edges with constraints specifying the adversary-controlled value as the property name and shortening the paths to the system built-in objects. If a system built-in function is redefined, a prototype pollution vulnerability is flagged.

Chapter 4

Approach

In this section is provided a description of the approach used in this thesis work. First we give a general overview of the approach. Second is described more in detail the process of creating a vulnerability database.

4.1 Approach Overview

In the first phase of the work we studied existing work about prototype pollution in order to understand the vulnerability and searched for tools that offer functionalities to detect it. From the start the focus of this work was towards improving the detection of the vulnerability and not on building attacks exploiting it. We then decided to limit the research on static analysis tools only. Existing dynamic approaches (e.g. scanners or fuzzers) don't perform any code analysis but only try to exploit known payloads/gadgets and so are less interesting from a detection perspective.

For the next phase of the work we manually collect a good number of vulnerable applications, testing that the vulnerability is actually present by writing simple proof of concept exploits. Then, we test the considered tools (Semgrep, CodeQL, ODGen, ObjLupAnsys) on our dataset. We select some case studies that show meaningful results (e.g. a tool can't detect the vulnerability while another can). The next phase is to analyze the results to understand why a pattern/tool does not work and if it is due to limitations of the pattern or the tool's engine? Other interesting questions are if detection does not work due to general limitation of static analysis or exclusive to a particular prototype pollution case.

Finally, starting from the false negatives observed, we propose various techniques to hide the vulnerability from the detection. We also analyze which code patterns lead to false positives, for instance when tools cannot distinguish a vulnerable pattern from a version in which the vulnerability has been mitigated.

4.2 Creating a vulnerability dataset

4.2.1 Collecting Vulnerable Functions

In order to evaluate how static analysis tools detect prototype pollution, we first need to build a dataset containing vulnerable functions. We search for JavaScript code containing prototype pollution vulnerabilities leveraging various sources:

- Vulnerability database inclusive of CVEs:
  - GitHub Advisory Database [40]
  - Snyk Vulnerability Database [41]
  - Mitre CVE List [3]
- Collection of vulnerable client-side libraries hosted on GitHub [24]
- Various CTF challenges
- Examples/tests provided by the tools analyzed (Semgrep, CodeQL, ObjLupAnsys, ODGen)

From these sources we select more than 40 examples.
We download the vulnerable package using npm (when it is a Node.js package) or directly from the git repository which is usually provided. We then analyze the code with the help of the descriptions to understand where is the vulnerability. In some cases the vulnerability report include the name of the vulnerable functions and also links to the git commit which fixes the problem.

4.2.2 Refactoring the vulnerable code

Since we want to obtain runnable examples, we need to perform some filtering operations to exclude irrelevant lines of code. Files may include code which is not needed to execute the functions we are interested to analyze: dependencies, functions, variables.
These operations also help speeding up the time of execution of analysis tools.

Some examples need to be modified a little in order to be executed. There could be references to functions which do not exist in the current file when it is a part of a large library with multiple files, and we are only interested in analyzing a small part. In these case we just copy the needed function(s) in the file.

Then there could be issues when working with client-side libraries. For instance when analyzing client-side scripts there could be references to browser global objects such as `document` or `window` which do not exist in Node.js. However, in the cases we considered (e.g. listing 4.2.1) these objects are used only to retrieve the

```
function parseParams (query) {
    ...
    query = query + '';
    if (query === '') query = window.location + '';
    ...
}
```

Listing 4.2.1: Example of window object usage in a function that parses querystrings

```
(function($) {
    ...
    var parseParams = function (query) { ... };
    ...
})(jQuery);
```

Listing 4.2.2: JQuery Closure example

URL, as the functions that parse querystrings may read it from `window.location` if no argument is passed to the function.
Another case regards scripts/plugins that extend jQuery functionalities. In those files (e.g. listing 4.2.2) the code is wrapped in a closure, this way the dollar sign `$` is a reference to the jQuery object. Since the jQuery object is not declared and we do not need it, we can extract the functions we want to analyze removing the closure.

4.2.3 Creating PoC exploits

For each function we provide a simple proof of concept (PoC) exploit to verify that the application is actually vulnerable to prototype pollution.
The scripts are very simple and all of them follow the same structure:

1. Import the vulnerable function
2. Call the vulnerable function passing the right arguments
3. Check if the prototype has been polluted

As example we provide the PoC exploit code for js-data (a Node.js package) in listing 4.2.3. The output of the script is showed in listing 4.2.4.

```javascript
import * as jsdata from "./utils.js";
var obj = {};
var payload = '{"__proto__": {"polluted": "Yes polluted"}}'
console.log("Before : " + {}.polluted);
console.log("Before : " + {}.polluted2);
jsdata.deepFillIn(obj, JSON.parse(payload));
jsdata.set(obj, '__proto__.polluted2', 'Yes it is polluted');
console.log("After : " + {}.polluted);
console.log("After : " + {}.polluted2);
```

Listing 4.2.3: PoC exploit example

```
> node js-data-poc.mjs
Before : undefined
Before : undefined
After : Yes polluted
After : Yes it is polluted
```

Listing 4.2.4: PoC exploit example output

Chapter 5

Case Studies

In this section we conduct case studies on some of the examples we have collected in our dataset. We consider 11 cases from 10 different packages/snippets/libraries, selected in order to have interesting situations (e.g. a tool can detect the vulnerability, others tools cannot). For each case we describe in detail why the vulnerability is present in the program and why the tool cannot detect the vulnerability.

5.1 CanJS deparam

CanJS [42] is a collection of client-side JavaScript architectural libraries. The can-deparam module offers the functionality to deserialize a query string into an object.

In listing 5.1.1 is reported a simplified version of the function, the actual one [43] is more complex as it considers also the case when parameters are arrays. It's possible to pollute `Object.prototype` by calling the function passing a string such as `"constructor[prototype][test]=yes"` or `"__proto__[test]=yes"`.

As in a "property definition by path" case the loop at lines 10-11 iteratively assigns the properties to reconstruct the querystring structure in a native JavaScript object, then the actual pollution occurs with the assignment at line 13.

The loop semgrep rule can detect the assignment at line 11 which iteratively assigns the properties to reconstruct the querystring structure in a native JavaScript object. It does not explicitly flag the assignment where the actual pollution occurs (line 13).

CodeQL assignment query and ODGen/ObjLupAnsys can detect the vulnerability using taint tracking analysis.

```
function deparam (params) {
    var data = {}, pairs, lastPart;
    if (params && paramTest.test(params)) {
        pairs = params.split('&');
        pairs.forEach(function (pair) {
            var parts = pair.split('='), key = prep(parts.shift()),
                value = prep(parts.join('=')), current = data;
            if (key) {
                parts = key.match(/([^\[\]]+)|(\[\])/g);
                for (var j = 0, l = parts.length - 1; j < l; j++)
                    current = current[parts[j]];
                lastPart = parts.pop();
                current[lastPart] = value;
            }
        });
    }
    return data;
}
```

Listing 5.1.1: CanJS deparam function

5.2 Component querystring

Component was a front-end package manager for JS which is not maintained anymore, however some packages are still maintained and available on npm. Querystring [44] is a simple key/value pair querystring parser and formatter.

Listing 5.2.1 shows the vulnerable `parse` function. Like other querystring parser it creates an empty object and then adds key/value pairs. This is an interesting case as the pollution only occurs if the property is a number due to the regex pattern `/(\w+)\[(\d+)\]/`. A possible payload is `"__proto__[1337]=polluted"`, at line 8 then the assignment `obj.__proto__.1337 = 'polluted'` would take place.

Semgrep does not flag it because the assignment rule can't detect "direct" assignments such as the vulnerable one at line 8. The function rule only works for recursive functions. The loop rule can't detect it as there is not an iterative assignment on the polluting properties like in other implementations of query parsers, e.g. CanJS (listing 5.1.1), the properties are "separated" by the regex at line 6.

Again the CodeQL assignment query and ODGen/ObjLupAnsys can detect the vulnerability using taint tracking analysis.

```
function parse(str){
    ...
  var obj = {}, pairs = str.split('&');
  for (var i = 0; i < pairs.length; i++) {
    var parts = pairs[i].split('='), m, key = decode(parts[0]);
    if (m = /(\w+)\[(\d+)\]/.exec(key)) {
      obj[m[1]] = obj[m[1]] || [];
      obj[m[1]][m[2]] = decode(parts[1]);
      continue;
    }
    obj[parts[0]] = null == parts[1] ? '' : decode(parts[1]);
  }
  return obj;
}
```

Listing 5.2.1: Component Querystring parse function

5.3 Extend2

Extend2 [45] is a port of the classic extend method from jQuery. The extend function merge the contents of two or more objects together into the first object. Listing 5.3.1 includes a simplified version of the extend function, which is an object recursive merge: at line 9 a recursive call is made if the property exists and it is an object if accessed on both objects that are being merged, otherwise the attribute is added to the target object (line 11). A possible poc exploit is included at the last line.

The Semgrep function rule does not detect the vulnerability because it does not recognize the function declaration if it is directly assigned to the exports object. Declaring the function and then exporting it solves the problem.

CodeQL function query and ODGen/ObjLupAnsys can detect the vulnerability using taint tracking analysis.

5.4 Js-data

JS-Data is a framework-agnostic, datastore-agnostic Object-Relational Mapping (ORM) for Node.js and the Browser.

Until Version 3.0.10 of the package the `deepFillIn` and `set` functions were vulnerable to prototype pollution. This is interesting as the previous version (v3.0.9) had another vulnerability in the `deepMixIn` function, but the patch only targeted that function. [46]

```
module.exports = function extend() {
    ...
    for (name in options) {
      src = target[name];
      copy = options[name];
      if (target === copy) continue;
      if (deep && copy && isPlainObject(copy)) {
        clone = src && isPlainObject(src) ? src : {};
        target[name] = extend(deep, clone, copy);
      } else if (typeof copy !== 'undefined') {
        target[name] = copy;
      }
    }
  }
  return target;
};
extend(true, {}, JSON.parse('{"__proto__": {"polluted": "yes"}}'))
```

Listing 5.3.1: Extend2 source code

deepFillIn

The `deepFillIn` function (listing 5.4.1) is yet another recursive object merge implementation. The `forOwn` function iterates the anonymous function over the `source` object own enumerable properties.

The function semgrep rule can detect the vulnerability at line 9 as it considers a sink any assignment to an object property inside a function that has a recursive call.

ODGen/ObjLupAnsys correctly identifies the vulnerability.

Surprisingly CodeQL does not detect it, even if, in theory, the function query should identify this case. It seems that the engine struggle to propagate taints when there is a complicated function calls sequence. In this case, the anonymous function defined at line 3 is passed to the `forOwn` method which invokes it (line 18), then the anonymous functions uses the object `dest` which is an argument of its parent function `deepFillIn`, and it also performs an indirect recursive call, invoking that function (line 6).

set

Listing 5.4.2 shows the `set` function which is a "property definition by path" case. For instance calling the function like did at line 15 would pollute `Object.prototype`. In that case `parts` would be `['__proto__.test', '__proto__', 'test', ...]`. The `mkdirP` function (listing 5.4.3) is the "loop part" of the typical "property def-

```
function deepFillIn (dest, source) {
  if (source) {
    forOwn(source, function (value, key) {
      const existing = dest[key]
      if (isPlainObject(value) && isPlainObject(existing)) {
        deepFillIn(existing, value)
      } else if (!Object.hasOwnProperty.call(dest, key)
                  || dest[key] === undefined)
          dest[key] = value
    })
  }
  return dest
}
function forOwn (obj, fn, thisArg) {
  const keys = Object.keys(obj)
  const len = keys.length
  for (let i = 0; i < len; i++)
    if (fn.call(thisArg, obj[keys[i]], keys[i], obj) === false)
      break
}
```

Listing 5.4.1: Js-data deepFillIn and forOwn functions

inition by path" function: it iteratively assigns the properties to reconstruct the path structure in a native JavaScript object. Therefore, at line 9 of the `set` function the assignment `Object.prototype.test='polluted'` would occur.

Semgrep function rule detects the vulnerability at line 9 in the `set` function, this is an assignment to an object property and the function has a recursive call. Semgrep loop rule detects the iterative assignment loop at line 7 in the `mkdirP` function.

ODGen/ObjLupAnsys does not detect the vulnerability in the `set` function. The problem regards the `isObject` function invoked at line 2. The function checks if the `path` value is an object by comparing the object tag (e.g. `"[object Object]"`) with the result of the `toString()` method which returns `"[object Type]"`, where Type is the object type of the `path` value. The actual implementation of the function, the first in listing 5.4.4, uses the `call()` method to invoke and that cause the false negative. Replacing it with the second function allow the tool to detect the vulnerability. This is a strange behavior as the call function is part of the abstract implementation. Also, the actual vulnerable code should be detected anyway as, like in our poc exploit, when the `path` value is a string only the else block is executed. Therefore, in theory, ODGen as a branch-sensitive static taint analysis tool should be able to detect it.

CodeQL assignment query, instead, correctly identifies the sink at line 9 in the set function (listing 5.4.2).

```
function set (object, path, value) {
    if (isObject(path)) {
        forOwn(path, function (value, _path) {
            set(object, _path, value)
        })
    } else {
        const parts = /^(.+)\.(.+)$/.exec(path)
        if (parts) {
            mkdirP(object, parts[1])[parts[2]] = value
        } else {
            object[path] = value
        }
    }
}
set({}, '__proto__.test', 'polluted');
```

Listing 5.4.2: Js-data set function

```
const mkdirP = function (object, path) {
    if (!path) return object
    const parts = path.split('.')
    parts.forEach(function (key) {
        if (!object[key])
            object[key] = {}
        object = object[key] //vuln
    })
    return object
}
```

Listing 5.4.3: Js-data mkdirP function

```
function isObject(obj){
  return Object.prototype.toString.call(value) === '[object Object]'
}
function isObjectAlt(obj){
  return value.toString() === '[object Object]'
}
```

Listing 5.4.4: Js-data isObject function

5.5 Mutiny

The parse function included in listing 5.5.1 is extracted from a rich library hosted as a minified JavaScript file on mutinycdn.

```
 1 function parse(query) {
 2   return query
 3     .replace("?", "")
 4     .split("&") //separates parameters
 5     .reduce((prevParam, currParam) => {
 6       var n = currParam.split("="),
 7         paramName = n[0],
 8         paramValue = n[1],
 9         paramNameArray = paramName.split(".");
10       return (
11         paramNameArray.reduce((prevNamePart, currNamePart, currIndex) => {
12           return (
13             currIndex === paramNameArray.length - 1 //check last element
14               ? (prevNamePart[currNamePart] = //vuln assign
15                   "string" == typeof paramValue
16                     ? unescape(paramValue.replace("+", " "))
17                     : paramValue || "")
18               : (prevNamePart[currNamePart] = prevNamePart[currNamePart] || {}),
19             prevNamePart[currNamePart]
20           );
21         }, prevParam),
22         prevParam
23       );
24     }, {});
25 }
26 parse('?__proto__.test=polluted');
```

Listing 5.5.1: Mutiny

It's another querystring parser method, the implementation is quite interesting, it uses two nested `Array.prototype.reduce` to loop over the query parameters. The `reduce` method executes a "reducer" callback function on each element of the array, in order, passing in the return value from the calculation on the preceding element. The final result is a single value. [47]
The first reduce (line 5) receive an array of parameters, if we consider our poc (call at line 26) there's only one parameter, and it is executed only one time. The callback function separates the field name and value, then creates the `paramNameArray` array which contains all the strings contained in the parameter name dot-separated. In our case `"__proto__"` and `"test"`. The second reduce function iteratively assigns the arrays elements (line 18) building the path specified, when it finally reaches the last element it assigns the parameter value (lines 14-16), polluting the prototype, in our poc at the second iteration.

This weird code pattern poses a challenge to every tool considered, not even one can detect the vulnerability.
Semgrep does not flag it because the vulnerable assignment it's not a two-step case. The function rule only identify recursive calls. The loop rule can't detect it because doesn't consider the case of implementing an iterative assignment using

a reduce.
CodeQL queries do not work due to engine limitations in propagating taints in presence of complicated nested function calls sequence.
ODGen/ObjLupAnsys does not detect it beacuse the tool fails to build ODG dataflow edges that represent the flow of data from the first reduce to the second.

5.6 Parse MockDB

Parse-mockdb package is a fully mocked in-memory Parse database implementation. [48]
The function `registerHook` (listing 5.6.1) is one of the rare cases of code that looks exactly like the easiest example of prototype pollution. The function registers a hook of type `hookType` on a class denoted by `className`, `hookFn` is the function that will be called. Since there are no checks on any of the parameters it's very easy to pollute `Object.prototype` calling the function like at line 9.
Despite being a very easy case, semgrep does not flag it. As mentioned before, the assignment rule can't detect direct assignments.
CodeQL assignment query and ODGen/ObjLupAnsys can detect the vulnerability.

5.7 pp.js

PP.js (listing 5.7.1) is a sample module provided by ODGen authors.
The function `pp` (which is the input of the module), calls the function `input_value` to get the value of `mid`, then calls `foo` using also `mid` as one of the parameters. Inside the function `foo`, the `tmp` variable is assigned with the `key1` property of the `target` object; then `value` is assigned to the property `key2` of `tmp`. If `key1` is `"__proto__"`, like in the possible poc exploit at line 14, then at line 4 a prototype pollution will occur.

```
1 let hooks = {};
2 function registerHook(className, hookType, hookFn) {
3   if (!hooks[className]) {
4     hooks[className] = {};
5   }
6   hooks[className][hookType] = hookFn;
7 }
8 registerHook("__proto__", "test", "polluted");
```

Listing 5.6.1: Parse-mockdb registerHook function

```javascript
function foo(key1, key2, value) {
  var target = {};
  var tmp = target[key1];
  tmp[key2] = value;
}
function input_value(val) {
  var mid = val + " ";
  return mid;
}
function pp(key1, key2, value) {
  mid = input_value(value);
  foo(key1, key2, mid);
}
pp("__proto__","test","polluted");
```

Listing 5.7.1: pp.js

This is an example which the semgrep assignment rule can identify because there is a “2-step assignment” (lines 3 and 4).
This module is made to showcase ODGen/ObjLupAnsys ability to perform inter-procedural data flow analysis and, as expected, it can detect the vulnerability. CodeQL can identify it as well.

5.8 Prototype pollution explained

This one is different from the other cases as it considers an entire application. It’s an example of a chat server which is vulnerable to prototype pollution due to the use of a vulnerable version of lodash merge. [49]
The application saves the users in an array of objects as showed in listing 5.8.1. The program, before deleting a message, checks if the user who made the request has the `canDelete` property set to `true`. The application also allows any authenticated user to post messages. The request made by the user is merged to a “default” message object (using the vulnerable lodash version), before being saved to the state. Therefore, the user can send a malicious message (e.g. the request

```javascript
const users = [
  {name: 'user', password: 'pwd'},
  {name: 'admin', password: Math.random().toString(32),
   canDelete: true},
];
```

Listing 5.8.1: users array

```
curl --request PUT \
  --url http://localhost:3000/ \
  --header 'content-type: application/json' \
  --data '{"auth": {"name": "user", "password": "pwd"},
   "message": { "text": "m", "__proto__": {"canDelete": true}}}'
```

Listing 5.8.2: Malicious request

of listing 5.8.2) successfully polluting `Object.prototype.canDelete`. Since non-admin users do not override the `canDelete` property, now the user `user` can delete the messages.

We included this example to showcase the functionality of the merge call CodeQL query. The query identifies call to known vulnerable functions when they receive as input user controllable objects. Since the merge lodash function used here is among those, the tool reports the vulnerable call.

Note that both CodeQL and Semgrep, by default, do not analyze the packages in the `node_modules` directory. ODGen/ObjLupAnsys, instead, follows the data flow analyzing also imported packages. However, due to efficiency problem, when analyzing large packages the execution of the tool takes a lot of time. In this case, with the lodash package, the tool did not produce an answer after hours of execution.

5.9 Semgrep loop

The prototype-pollution-loop.js module contains 5 functions (3 vulnerable and 2 non-vulnerable) provided by semgrep to test the homonymous rule. These functions are all "property definition by path" cases: a loop iteratively assigns the properties to an object in order to reach the specified path, then a value is assigned to that property.

In the first function (listing 5.9.1) a `forEach` is used to loop over the properties. Obviously the semgrep loop rule detect the iterative assignment at line 10, however the tool do not explicitly flag the line in which the actual pollution take place (line 6).

Both CodeQL assignment and function queries precisely identify the vulnerability.

ODGen/ObjLupAnsys, instead, does not flag the vulnerability. There are problems generating data flow edges in the ODG when a `forEach` is used to iterate over the polluting properties. The same exact function but with the `forEach` converted to a for loop is marked as vulnerable by the tool.

```javascript
function test1(name, value) {
  let config = {};
  name = name.split('.');
  name.forEach(function (item, index) {
    if (index === name.length - 1) {
      config[item] = value;
    } else {
      if (!isObject(config[item]))
        config[item] = {};
      config = config[item];
    }
  });
  return config;
}
test1("__proto__.test","polluted");
```

Listing 5.9.1: Semgrep loop test1 function

```
{"constructor":{"prototype":{"test":"polluted"}}}
{"prototype":{"test":"polluted"}}
{"test":"polluted"}
"polluted"
```

Listing 5.10.1: Yui3 addition values

5.10 Yui3

Yui3 was a library for building richly interactive web applications which is no longer maintained. The vulnerability is exposed by the `parse` function [50] which returns a native JavaScript object representation of a querystring.

This is another “recursive object merge” case. The actual vulnerability is in the `mergeObjects` internal function, which is called when both `params` and `addition` are objects in the `mergeParams` function.

For instance if the querystring `"constructor[prototype][test]=polluted"` is passed to the `parse` function then `addition` would assume the values showed in listing 5.10.1 on the different recursive calls. Therefore the third and last assignment at line 4 (listing 5.10.2) would succesfully pollute the prototype: `Object.prototype.test='polluted'`.

Semgrep does not detect the vulnerability: the function rule considers only direct recursive calls, at line 4 an indirect recursion takes place (i.e. when a method invokes another method, eventually resulting in the original function being called again).

CodeQL function query and ObjLupAnsys can detect the vulnerability using in-

```
function mergeObjects (params, addition) {
    for (var i in addition)
        if (i && addition.hasOwnProperty(i))
            params[i] = mergeParams(params[i], addition[i]);
    return params;
}
function mergeParams (params, addition) {
return (
    (!params) ? addition
    : (Array.isArray(params)) ? params.concat(addition)
    : (!isObject(params) || !isObject(addition)) ? [params].concat(addition)
    : mergeObjects(params, addition)
  );
}
function parse (qs, sep, eq) {
  return arrayReduce(
            arrayMap(qs.split(sep || "&"), pieceParser(eq || "=")),
            {},
            mergeParams);
}
```

Listing 5.10.2: Yui3 source code

terprocedural taint tracking analysis.
It is interesting to note that while ObjLupAnsys can identify the vulnerability in seconds, ODGen did not produce an answer after hours of execution.

Chapter 6

Results Analysis

In this chapter we explain in detail what could lead to false negatives. Actual false negatives in the code or just environment / tools misconfiguration. Starting from the false negatives found in our dataset, we explain possible ways to bypass the tools' detection. In the second part we describe the root causes for false positives.

	Tool / Pattern							
	Semgrep			CodeQL			ODGen	ObjLupAnsys
Case Study	A	F	L	A	F	M		
CanJS-Deparam			✓	✓			✓	✓
Component-querystring				✓			✓	✓
Extend2					✓		✓	✓
Js-data deepFillIn		✓						✓
Js-data set		✓		✓				
Mutiny								
Parse-mockdb				✓			✓	✓
PP.js	✓			✓			✓	✓
PP explained						✓		
Semgrep loop test1			✓	✓	✓			
Yui3					✓			✓

Table 6.1: Case studies results: vulnerability coverage of the tools/patterns considered
The checkmark means that the tool can detect the vulnerability.
From the left: Assignment Semgrep rule, Function Semgrep rule, Loop Semgrep rule, CodeQL Assignment query, CodeQL Function query, CodeQL Merge call query, ODGen, ObjLupAnsys

```
{ "name" : "foo"
, "version" : "1.2.3"
, "main" : "file.js"
}
```

Listing 6.1.1: Minimal package.json example: this file is used by npm to include project properties such as version, dependencies, etc...

6.1 False negatives analysis

6.1.1 Configuration settings

In this section are presented some configuration settings that must be considered to avoid false negatives because the tools aren't even analyzing the vulnerable functions.

CodeQL and ObjLupAnsys/ODGen only analyze functions which are directly or indirectly controllable by the user. Therefore, we have to be sure that the vulnerable function is exported or is invoked by an exported one. This requirement is guaranteed as for all case studies we created a simple poc exploit to verify that the function is actually vulnerable.

A problem which affects CodeQL is that it often does not recognize javascript modules as modules but treats them as top level scripts. Consequentially data flow queries ignore those functions, as the user cannot call them. According to CodeQL documentation it should recognize a javascript file as module if it contains statements such as `import` or `export`, however in many cases this is not enough. We found that a workaround is to include a package.json file (listing 6.1.1) in the same folder which states that the file to analyze is the primary entry point of a Node.js module.

When analyzing entire applications / packages is important to remember that CodeQL and semgrep, by default, do not analyze packages in the `node_modules` directory (where the Node.js dependency packages are stored). Also, by default they ignore minified (.min.js) files.

6.1.2 Taint analysis limits

Semgrep

Semgrep does not perform interprocedural data flow analysis, therefore its rules may miss some vulnerabilities that are distributed over more than one function (e.g. Yui3: indirect recursive call, section 5.10). Odgen/ObjLupAnsys and CodeQL instead perform advanced taint analysis, but can still miss some cases.

```
1  function exp1(a, b, val){
2      var a1 = a + "";
3      var b1 = b + "12";
4      var b2 = b1 + "3";
5      var val1 = val + 4;
6      var obj = {};
7      obj[a1][b2] = val1; //detected
8  }
9  function exp2(a, b, val){
10     var obj = {};
11     obj[a][b] = {prop: val}; //False Negative
12     obj[a][b] = {prop: val}.prop; //detected
13 }
```

Listing 6.1.2: ODGen taint analysis limits

ODGen/ObjLupAnsys

ODGen/ObjLupAnsys has object-level taint marking. In the case of listing 6.1.2 in the first function the object `a1` is marked as tainted because it is created using the tainted object `a`, which is controllable by the user, and so on. In the second function the object `{prop: val}` is not marked as tainted since ODGen does not consider the case when the vulnerable object `val` is used as a child of the new object, since no new object is created based on the existing tainted objects. If instead we access the property `prop` of the new object the tool will flag the vulnerability (line 12).

Odgen/ObjLupAnsys also may not propagate taints when a new object is created using a function (e.g. `trim()`, which is available in the abstract implementation of the tool). In listing 6.1.3 ODGen detects the vulnerability in the first function but not in the second.
Under the same category is the case of the js-data set function (listing 5.4.2), both `call` and `toString` functions are part of the abstract implementation, however the use of the `call` function on the tainted value cause problems in the generation of the ODG.

CodeQL

CodeQL may not propagate the taints when objects are encapsulated inside arrays. For instance in extend2 (section 5.3) the parameters are passed using `arguments`, (first function in listing 6.1.4), an array-like object but not an Array, which contains the values of the arguments passed to the function [51] (used to access parameters in variadic functions). As we have seen CodeQL can detect the vulnerability, however if we implement the function using rest parameters, which

```
function exp1(a, b, val){
    var obj = {};
    obj[a.trim()][b] = val; //detected
}
function exp2(a, b, val){
    var a1 = a + "";
    var obj = {};
    obj[a1.trim()][b] = val; //false negative
}
```

Listing 6.1.3: Vulnerable assignments using trim function

```
function extend(){
    var target = arguments[0];
    target[name] = extend(deep, clone, copy);
}
function extend(...args){
    var target = args[0];
    target[name] = extend(deep, clone, copy);
}
function extend(args){
    var target = args[0];
    target[name] = extend([deep, clone, copy]);
}
```

Listing 6.1.4: Different implementations of variadic function in JavaScript

is JavaScript newer standard to accept an indefinite number of arguments as an array [52] (listing 6.1.4 second function) or by directly passing an array (third function), CodeQL function query won't detect the vulnerability.

CodeQL engine also encounters difficulties in propagating taints when there is a complicated function calls sequence, has we have seen with the js-data deepFillIn function (listing 5.4.1).

6.1.3 Syntax problems

In the extend2 package (section 5.3) we noticed that semgrep function rule may not detect a vulnerability because it does not consider the case of a "named" function declaration if it is directly assigned to the exports object (or also any other object). When the function is declared and then exported the rule can detect it. In listing 6.1.5 are presented some meaningful examples.

We also propose an easy fix to include this case in the rule, we have to add the code in listing 6.1.6 to the sink patterns of the rule (listing 3.2.2).

```
//Not detected
var x;
x = function vuln() { ... vuln() ... }
module.exports = function vuln() { ...  vuln() ...};
//Detected
function vuln() { ... vuln() ... };
const vuln = function() { ... vuln() ... };
```

Listing 6.1.5: Different ways of declaring functions in JavaScript, the function rule does not consider the case of "named" function declaration assigned to other objects. Note that in this case the vuln identifier is visible only inside the function scope.

```
- pattern-inside: |
  $OBJ = function $NAME(...) { ... $NAME(...) ... }
```

Listing 6.1.6: Semgrep function rule improvement

ODGen/ObjLupAnsys do not support all ES2015 (ES6) and more recent features (e.g. `class` and `extends`), thus it is highly recommended using a transcompiler like Babel.js to convert ES2015+ code to ES5 compatible code before the analysis. If incompatible code is included in the file to analyze the tool may not produce any results nor errors.

In the test1 function of the semgrep loop tests (section 5.9) ODGen/ObjLupAnsys can't detect the vulnerability because a `forEach` is used to iterate over an array's elements. Despite the `forEach` function is among the modelled ones in the abstract implementation (`modeled_js_builtins.py` file in the source code) and the tool can find vulnerabilities when it is used (e.g. CanJS-Deparam package, section 5.1.1). However, in that case the `forEach` is not used directly to iterate on the polluting properties. There are problems generating data flow edges in the ODG when a `forEach` is used. The Object-Helpers package [53] includes the set function, which is very similar to test1, but the loop is implemented using a `for` loop and ODGen detects it. Infact, if we convert the `forEach` to a `for` loop then the tool can identify the vulnerability.

6.1.4 Simple assignments

As we have seen in case studies on Parse-mockdb (section 5.6) and component-querystring (section 5.2) packages, the semgrep assignment rule does not consider "direct" assignments that could lead to prototype pollution, the basic examples are included in listing 6.1.7. The rule can only detect "two-step assignments" like in pp.js (section 5.7).

The rule uses a pattern-inside operator to force this constraint. Removing this condition would however match almost any assignment to an object property

```
function foo1(obj, a, b, val) {
    obj[a][b] = val;
}
function foo2(obj, a, b, c, val) {
    obj[a][b][c] = val;
}
```

Listing 6.1.7: Simple vulnerable assignment examples which are not detected by semgrep

```
function exploit2(string, input, val){
    try {
        var obj = {};
        throw 'myException';
    }
    catch (e) {
        obj[string][input] = val; //False Negative
    }
}
```

Listing 6.1.8: ODGen Try/Catch false negative example

causing too many false positives. This is a limit of the semgrep engine, which can't perform analysis on object lookups based on prototype chain.

6.1.5 Try Catch

ODGen/ObjLupAnsys do not detect sinks inside a `catch` or a `finally` block (e.g. listing 6.1.8). However, if the vulnerable assignment is moved into the `try` block or after the `catch/finally` blocks it is correctly flagged.

6.2 False positives analysis

A high number of false positives is often the problem of static analysis tools. While this thesis work is focused more on understanding why a certain tool could or could not identify vulnerabilities, also understanding which non-vulnerable code patterns a tool may erroneously report is interesting.

6.2.1 Patched functions

The main source of false positives are patched functions which were originally vulnerable to prototype pollution. Different ways of preventing the vulnerability are explained in section 2.2.

```javascript
const isPrototypePolluted = function (key) {
  return ['__proto__', 'prototype', 'constructor'].includes(key)
}
```

Listing 6.2.1: Js-data function to blacklist polluting keys

```javascript
function foo(obj, a, b, val) {
    var obj = {__proto__ : null}
    obj[a][b] = val; //false positive
}
```

Listing 6.2.2: ___proto___ override false positive

- Filtering malicious user input is the most used approach to mitigate prototype pollution. Semgrep assignment rule does not flag assignments inside an `if` block which condition contains `__proto__` or `constructor`. However, it misses a lot of cases as the keys can be checked in multiple ways, for instance using ad-hoc functions (e.g. listing 6.2.1) or without an `if` statement. Also, even when the control is implemented in that way, the assignment may be outside the `if` body (e.g. `if(key === '__proto__') continue;`). CodeQL performs very good on this cases understanding also more complex implementation of the check like the choice of js-data maintainers, listing 6.2.1. ODGen/ObjLupAnsys, instead, fall even for the most simple cases, having a very high false positive rate.

- Another approach is using the `hasOwnProperty` method, which checks if the object has the specified property as its own property, without looking on the prototype chain. Therefore, when merging a malicious object containing the `__proto__` property with another object, if the check is performed on the non tampered object, the method returns false. Semgrep function rule consider this case but only if the possible vulnerable assignment is inside an `if` block which condition uses the `hasOwnProperty` method. CodeQL and ODGen/ObjLupAnsys successfully avoid to report false positives for these cases.

- Freezing the prototype also successfully mitigate all prototype pollution attacks, when this approach is used, all the tools report false positives.

- Creating an object without a prototype using, is another possibility. Semgrep does not consider it, CodeQL and ODGen/ObjLupAnsys instead avoid reporting the false positive.

- When an object `__proto__` property is overridden the tools still detect a possible vulnerability even when it is impossible to access the prototype (listing 6.2.2).

- Indeed, static analysis can't understand if the `__proto__` property has been disabled using the `--disable-proto` flag in NodeJS, as it requires analyzing the run-time environment.

6.2.2 General purpose JavaScript code

When dealing with random JavaScript code it's possible to find false positive also for the following reasons:

- For semgrep loop rule, a situation that could case false positives are functions that contain loops over object properties, which have a similar logic of a "property definition by path" function, but do not perform assignment of a value. Since the rule targets only the iterative assignments, it does report a false positive. As an example listing 6.2.3 contains the `hasKey` function from the minimist package.

- In listing 6.2.4 is included another false positive example: it's impossible to pollute the prototype as the value `a` cannot be both `__proto__` and another property at the same time. All tools do not consider this possibility.

- Code obfuscation techniques may create obscure patterns which could lead to both false negatives or false positives. In some cases the obfuscation may not modify the vulnerable pattern, but just the protection measures. This is however a classic limit of static analysis and not strictly related to the prototype pollution vulnerability or a particular tool.

```javascript
function hasKey (obj, keys) {
    var o = obj;
    keys.slice(0,-1).forEach(function (key) {
        o = (o[key] || {}); //false postive
    });
    var key = keys[keys.length - 1];
    return key in o;
}
```

Listing 6.2.3: Semgrep loop rule false positive example

```javascript
function foo5(obj, a, b, val) {
    obj[a][a] = val; //FP
}
```

Listing 6.2.4: Simple false positive case

6.3 Performance considerations

Semgrep and CodeQL offer great performance also when analyzing large packages, semgrep is really fast considering that it does not have to build a database every time the codebase is modified.
ODGen/ObjLupAnsys, on the other hand, suffer from serious performance problems when dealing with large packages (e.g. lodash, section 5.8) or some complex code patterns (e.g. yui3, section 5.10). The generation of the ODG may take hours. Also, there are differences in how the graph is used to find the vulnerabilities and their performance. Our tests confirm that the performance of real-time detection (ObjLupAnsys) is better than graph query (ODGen) in most cases. To be fair we must consider that they are experimental tools and not production grade tools like Semgrep and CodeQL.

Chapter 7

Conclusions

In this work we evaluated how well static analysis tools can detect prototype pollution vulnerability, a relatively new vulnerability affecting only JavaScript programs due to its prototype-oriented design.

We presented the current state of the art static analysis tools.

ODGen/ObjLupAnsys, an experimental static taint analysis tool that specifically targets prototype pollution vulnerability. Semgrep, a fast static analysis tool based on ASTs. CodeQL, a powerful SAST with a semantic code analysis engine which finds security bugs by querying code as if it were data. Both Semgrep and CodeQL have the explicit goal of allowing custom rules, and include official rules to detect prototype pollution.

We collected JavaScript code containing real word prototype pollution vulnerability examples searching on vulnerability databases such as GitHub Advisory and other sources. To be sure that the collected examples are actually vulnerable we wrote a simple proof of concept exploit for each vulnerable function. We ran the static analysis tools considered against the vulnerable applications. Among the vulnerability dataset we picked some case studies which we analyzed in detail to explain different code patterns that lead to the vulnerability. These case studies were chosen to show interesting results which allowed us to highlight the strengths and limitations of each tool.

We analyzed the results gathered with a stronger focus on false negatives, in particular:

- ODGen/ObjLupAnsys new approach, based on Object Dependence Graph which succesfully model object lookups based on prototype chain, can detect almost all vulnerability cases. The current experimental implementation, however, is affected by bugs when it encounters some code patterns. It also suffers from serious performance issues when analyzing large packages or certain patterns. Also, it is very prone to flag false positives when executed against patched version of precedently vulnerable functions.

- Semgrep rules do not cover all cases such as direct assignments, due to the difficulty of including this case without having a very high false positive rate. The semgrep engine offers only limited intraprocedural dataflow analysis, and as expected can't always detect vulnerability that are distributed across different functions (e.g. indirect recursive calls). The rules flag many false positives when executed against patched version of precedently vulnerable functions, as most mitigation techniques are not even considered.

- CodeQL can identify almost all vulnerability examples collected. The query considered seem to be well written, the problems encountered are most likely limitation of the closed source engine. For what concerns false postives, most mitigation techniques are considered, thus CodeQL queries performs significantly better than other tools.

We showed that there are still cases that no tool is able to identify and also several detection avoidance techniques.

We also reported our findings to the maintainers of each project by creating issues on the respective github repository.

The natural continuation of this work is to use the results to further improve the rules/queries. New rules may be created to catch specific patterns that can't be reconducted to the most common implementations.

An interesting work could be implementing the observed patterns using other static analysis engines such as Joern.

Bibliography

[1] Olivier Arteau. Prototype pollution attack in nodejs application. https://github.com/HoLyVieR/prototype-pollution-nsec18, 2018.

[2] CWE-1321: Improperly Controlled Modification of Object Prototype Attributes ('Prototype Pollution'). https://cwe.mitre.org/data/definitions/1321.html.

[3] CVE Records about Prototype Pollution. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=prototype+pollution.

[4] ppmap. https://github.com/kleiton0x00/ppmap.

[5] Ppfuzz. https://github.com/dwisiswant0/ppfuzz.

[6] Semgrep: Static analysis at ludicrous speed find bugs and enforce code standards. https://semgrep.dev/.

[7] Codeql. https://codeql.github.com/.

[8] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. Mining node.js vulnerabilities via object dependence graph and query. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association.

[9] 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.

[10] About javascript. https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript.

[11] Usage statistics of javascript as client-side programming language on websites. https://w3techs.com/technologies/details/cp-javascript/.

[12] Javascript inheritance and the prototype chain. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain.

[13] Javascript classes. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes.

[14] Prototype.js library. http://prototypejs.org/.

[15] Michał Bentkowski. Prototype pollution in kibana. https://research.securitum.com/prototype-pollution-rce-kibana-cve-2019-7609/, 2019.

[16] Sql injection or denial of service due to a prototype pollution. https://hackerone.com/reports/869574.

[17] Michał Bentkowski. Prototype pollution – and bypassing client-side html sanitizers. https://research.securitum.com/prototype-pollution-and-bypassing-client-side-html-sanitizers/, Aug 18, 2020 [Online].

[18] Sergey Bobrov, Mohan Sri Rama Krishna P, et al. ”A tale of making internet pollution free” - Exploiting Client-Side Prototype Pollution in the wild. https://blog.s1r1us.ninja/research/PP, Apr. 18, 2021 [Online].

[19] Map. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map.

[20] Wolfgang Wögerer. A survey of static program analysis techniques. Technical report, Citeseer, 2005.

[21] Static Code Analysis - OWASP. https://owasp.org/www-community/controls/Static_Code_Analysis.

[22] Vulnerability Scanning Tools - OWASP. https://owasp.org/www-community/Vulnerability_Scanning_Tools.

[23] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. A survey of dynamic analysis and test generation for javascript. *ACM Comput. Surv.*, 50(5), sep 2017.

[24] Client-Side Prototype Pollution. https://github.com/BlackFan/client-side-prototype-pollution.

[25] Protoscan. https://github.com/KathanP19/protoscan.

[26] PPScan. https://github.com/msrkp/PPScan.

[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, 2021.

[28] Objlupansys. `https://github.com/Song-Li/ObjLupAnsys`.

[29] Odgen. `https://github.com/Song-Li/ODGen`.

[30] Client-side prototype pollution tools. `https://github.com/BlackFan/cspp-tools`.

[31] Semgrep prototype pollution rules. `https://github.com/returntocorp/semgrep-rules/tree/develop/javascript/lang/security/audit/prototype-pollution`.

[32] Codeql prototype pollution queries. `https://github.com/github/codeql/tree/main/javascript/ql/src/Security/CWE-915`.

[33] Oege de Moor, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjorn Ekman, Neil Ongkingco, Damien Sereni, and Julian Tibble. Keynote address: .ql for source code analysis. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 3–16, 2007.

[34] Codeql documentation: About data flow analysis. `https://codeql.github.com/docs/writing-codeql-queries/about-data-flow-analysis`.

[35] CodeQL documentation: Template to create a taint tracking path query. `https://codeql.github.com/docs/codeql-language-guides/data-flow-cheat-sheet-for-javascript/`.

[36] Issue about CodeQL for Client Side Prototype Pollution. `https://github.com/github/securitylab/discussions/209#discussioncomment-145109`.

[37] Enforcing Code & Security Standards with Semgrep. `https://owasp.org/www-chapter-newcastle-uk/presentations/2021-02-23-semgrep.pdf`.

[38] Ulziibayar Otgonbaatar. Pain-free Custom Linting: Why I moved from ESLint and Bandit to Semgrep. `https://r2c.dev/blog/2020/why-i-moved-to-semgrep-for-all-my-code-analysis/`.

[39] ODGen: A few technical questions about the source code. `https://github.com/Song-Li/ODGen/issues/1`.

[40] GitHub Advisory Database. `https://github.com/advisories?query=prototype+pollution`.

[41] Snyk Open Source Vulnerability Database. https://security.snyk.io/.

[42] CanJS. https://canjs.com/index.html.

[43] CanJS deparam source code. https://github.com/canjs/can-deparam/blob/8c902fc6b9f2e6107229d76ee3990f74952a804a/can-deparam.js.

[44] Component querystring. https://github.com/component/querystring/blob/7334366bae9b0434d2aa3a6ac9a9039eb1d17144/index.js.

[45] Extend2. https://github.com/eggjs/extend2.

[46] Snyk DB Prototype Pollution Affecting js-data. https://security.snyk.io/vuln/SNYK-JS-JSDATA-1584361.

[47] Array.prototype.reduce documentation. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce.

[48] Parse MockDB source code. https://github.com/Hustle/parse-mockdb/blob/master/src/parse-mockdb.js.

[49] Prototype Pollution Explained. https://github.com/Kirill89/prototype-pollution-explained.

[50] Yui3 vulnerable source code. https://github.com/yui/yui3/blob/25264e3629b1c07fb779d203c4a25c0879ec862c/src/querystring/js/querystring-parse.js.

[51] Javascript the arguments object. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments.

[52] Javascript rest parameters. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters.

[53] Object-Helpers package. https://www.npmjs.com/package/object-helpers.