



TENABLE BLOG

[Subscribe](#)

Identifying Prototype Pollution Vulnerabilities: How Tenable.io Web Application Scanning Can Help

[Rémy Marot](#) | Research

May 25, 2021 | 9 Min Read



Prototype pollution vulnerabilities are complex issues which can put your web applications and users at serious risk. Learn how these flaws arise and how Tenable.io Web Application Scanning can help.

JavaScript is a language implemented in nearly all web applications. From building rich





[security advisories](#) issued for the Node Packet Manager packages). For example, the popular JavaScript library [Lodash](#), used as a dependency by more than 140k packages, has been impacted by this [bug](#) referenced as [CVE-2020-8203](#). From cross-site scripting (XSS) to remote code execution (RCE) attacks, malicious actors can conduct advanced exploitation scenarios with prototype pollution vulnerabilities.

In this post, we will cover the key concepts behind prototype pollution bugs, and how [Tenable.io Web Application Scanning \(WAS\)](#) can help you proactively identify these vulnerabilities.

Prototype Concepts

[Prototype-based programming](#) is based on the process of defining objects called prototypes, which are then extended or cloned to create new objects. Once instantiated, these objects will carry over the properties and the functions of their prototype, as well as their local properties or functions.

JavaScript is one of the most commonly used prototype-based languages in modern web applications. Most of the JavaScript objects are derived from the *Object* class and share the same prototype called *Object prototype*. For example, a basic JavaScript object can be created with the following snippet of code:

```
var tenable = {"company": "Tenable"}
```

This object is simple and contains only one property using the key *"company"* and the value *"Tenable"*. Once this new object is instantiated with its [literal notation](#), it will be linked by default to the *Object prototype*.





```
Object.getPrototypeOf(tenable, Object.prototype)  
  
>> true
```

Objects can also be instantiated by using a specific constructor function.

```
function company(name) { this.name = name }
```

By calling this constructor function with the *new* keyword, the new object will then be instantiated.

```
var tenable = new Company('tenable')
```

The prototype of this newly instantiated object will be the one of its constructor function which is usually the *Object prototype*.





```
tenable.__proto__  
>> {constructor: f}  
  
tenable.__proto__.__proto__ === Object.prototype  
>> true  
  
tenable.constructor.prototype === tenable.__proto__  
>> true
```

The inheritance between the objects and their prototype is called the prototype chain. Each time the code checks for a property of an object, the JavaScript runtime will first verify the properties of the current object and then go up the chain until it reaches the top level, which is usually the *Object prototype*.

The code below shows an attempt to access a property named "theme", which is not declared in the object. The property is then set on the *Object prototype* and becomes available in the object through the prototype chain.

```
tenable.theme  
>> undefined  
  
Object.prototype.theme = 'Cyber Exposure'  
tenable.theme  
>> "Cyber Exposure"
```

For object instances, the link to their prototype is done through the built-in property





Prototype pollution vulnerabilities occur when the code of the application allows the alteration of any prototype properties, usually those of the *Object prototype*. By inserting or modifying a property of a prototype, all inherited objects based on that prototype would reflect that change, as will all future objects created by the application.

In the previous example, the following object was created:

```
var tenable = {"company": "Tenable"}
```

The following code snippet accesses the *Object prototype* through its `__proto__` property to redefine the basic `toString()` function, which returns a string representation of an object:

```
tenable.__proto__.toString = () => { console.log('stop pollution') }
```


Now, each time the `toString()` function is called on this object, the console will display the 'stop pollution' message:





```
tenable.toString()  
  
>> "stop pollution"
```

When creating a new object, the same behavior is observed, as it inherits from the *Object* prototype:

A dark-themed code editor window with standard window controls (minimize, maximize, close) in the top right corner. It contains the following JavaScript code:

```
var google = new Company('Google');  
google.toString()  
  
>> "stop pollution"
```

```
var google = new Company('Google');  
google.toString()  
  
>> "stop pollution"
```

This short demonstration shows that being able to alter the properties of a prototype has an immediate impact on the code accessing and evaluating it.

Exploitation vectors and risks

JavaScript can be used on both the client and server side of a web application. Prototype pollution vulnerabilities exist in both of these contexts and can lead to a wide range of attacks depending on the application logic and implementation.

Most of the time, the first impact of exploiting this type of vulnerability is the ability to





conducting further attacks on a targeted application. By leveraging the other components loaded in the same context (called *Gadgets*), the attacker can initiate more complex attacks in order to gain further privileges or access sensitive information.

Client-side exploitation

The exploitation starts with the injection of a payload into an input that is used to build the client-side logic or rendering of the application. The most common source of input is the URL and its different properties, like *location.search*:

```
https://vulnerable.app/?__proto__[pollutedKey]=pollutedValue
```

These types of parameters are a good example of a key-value data structure, where it is common to see URL parsers assign JavaScript objects properties from this type of string without verifying if the target property is correctly linked to the *Object prototype*.

The following vulnerable code, simplified for the purpose of this post, declares a function that takes a URL as an argument and returns an Object with the parameters and their values extracted from the query string.

```
function getQuery(url) {  
  let queryString = url.substring(url.indexOf('?') + 1).split('&')  
  let result = {}  
  
  for(let i = 0; i < queryString.length; i++) {  
    queryString[i] = queryString[i].split('=')  
    let params = Array.from(queryString[i][0].matchAll(/\[?(^[\]]+)\]?/g))  
  
    result[params[0][1]][params[1][1]] = decodeURIComponent(queryString[i][1])  
  }  
  return result  
}
```


The function is then called on a crafted URL which contains a payload to test for a





```
>> {}
```

The return value is an empty Object, but the vulnerability is confirmed by looking at the *Object prototype* properties which now contains the “*pollutedKey*” property with the “*pollutedValue*” value.



```
Object.prototype.pollutedKey  
>> “pollutedValue”
```

The most obvious outcome of the exploitation of a prototype pollution vulnerability in a client-side context is the ability to perform a XSS attack. By identifying a *gadget* which will rely on a property of an object that can be polluted and interact with the [document object model \(DOM\)](#) of a page, it is possible to trigger client-side JavaScript code execution.

The current research around client-side exploitation also shows a specific use case leading to the [bypass of some HTML sanitization libraries](#). These libraries maintain an allowlist of tags or attributes to be used in the various user inputs of the application to ensure that no malicious content can be submitted. By polluting specific properties, it is possible to modify the allowlist to enable further injection of payloads that could allow for XSS attacks.

Finally, prototype pollution vulnerabilities can also be used to defeat the protection of some web application firewalls (WAF) which would, under certain circumstances,





context. Because the JavaScript runtime objects executed on the server could be modified, the impact is often more severe and can leave an application open to critical vulnerabilities:

- **Remote Code Execution:** The [proof-of-concept](#) (PoC) related to [CVE-2019-7609](#) affecting the [Elastic Kibana](#) software shows that polluting the *Object prototype* can be a vector to fully compromise a target application and its system. The exploit demonstrates that an attacker can establish a reverse shell on the Kibana server by being able to pollute environment variables loaded by the application.
- **SQL Injection:** Depending on the application code and modules available, it is also possible to perform attacks like SQL injection as shown in [this PoC](#). This report shows that the TypeORM package implemented a vulnerable *mergeDeep* function. When used on an object containing properties controlled by an attacker, the *Object prototype* could be polluted and used to perform SQL injection attacks by appending additional clauses to the queries.
- **Authorization and Authentication Bypass:** Some applications, [for example](#), may check the permissions or even the authentication data of their users against the properties of JavaScript objects. If an attacker is able to pollute a property that is used during this verification, they could gain privileged access on the target application or even authenticate without having a valid user account.

Exploiting this type of vulnerability can be complex, and requires a deep analysis of the application logic to be able to determine the impact of prototype pollution.

Use Tenable.io WAS to detect prototype pollution issues

[Tenable.io WAS](#) helps identify Prototype Pollution vulnerabilities through multiple features:

- Plugin [112719](#) is dedicated to the detection of generic client-side prototype pollution issues and helps identify [CVE-2021-20083](#), [CVE-2021-20084](#), [CVE-2021-20085](#), [CVE-2021-20086](#), [CVE-2021-20087](#), [CVE-2021-20088](#), [CVE-2021-20089](#) vulnerabilities.





PUBLISHED 03/12/2021	MODIFIED 03/12/2021	SEE ALSO https://research.securum.com/prototype-pollution-and-bypassing-client-side-helm-sanitizers/ https://portswigger.net/daily-swig/prototype-pollution-the-dangerous-and-underestimated-vulnerability-impacting-javascript-applications https://nyk.io/blog/nyk-research-team-discovers-severe-prototype-pollution-security-vulnerabilities-affecting-all-versions-of-lodash/
Risk Information		
CVSS3 BASE SCORE 7.1	CVSS3 VECTOR CVSS:3.0/AV:N/AC:L/PR:N/UI:R/SC:C/LI:L/A:L	
CVSS2 BASE SCORE 7.5	CVSS2 VECTOR CVSS2:AV:N/AC:L/AU:N/C:P/I:P/A:P	
Reference Information		
CWE 1321	WASC -	OWASP -
CVE -	BID -	OWASP API 2019-API7

- Three fingerprinters are available to detect NodeJS applications and their two most common web frameworks: ExpressJS and SailsJS. These checks identify applications which are more likely to be impacted by this class of vulnerability.

Prevent and mitigate

There are multiple best practices available to protect applications and users and prevent prototype pollution vulnerabilities:

- **Sanitize inputs:** One of the most common attack vectors is the assignment of a JavaScript object property from a key-value data structure. Code that performs unsafe operations like merging, cloning or extending objects will often not check for the special properties `__proto__` and `constructor`, allowing for the modification of the prototype. JSON or YAML inputs are commonly used for key-value data and are often used as a source for this type of processing. Properly designed code should perform a strong parsing and validation against the properties that are set to prevent any modification of the *Object prototype*. For example, [the fix released for the popular qs package](#) shows how a weak input validation can introduce a prototype pollution vulnerability.
- **Freeze the Object prototype:** Server-side contexts can be protected by freezing the *Object prototype* to make it immutable by using `Object.freeze(Object.prototype)`. Further modification attempts on the *Object prototype* will silently fail, keeping it safe from any pollution. Although this is a strong mitigation option, this may introduce functional bugs in the application and should be carefully implemented.
- **Create prototype-less objects:** The JavaScript API allows the creation of prototype-less objects, removing the relationship inheritance used in most cases to perform the pollution. This requires explicitly creating the object with the `Object.create(null)` function.
- **Use Map objects:** JavaScript objects are often used to store simple properties as a key-value pair and do not require the presence or the usage of inherited properties and methods. The purpose of the Map object is to offer a safe option





SEE MORE INFORMATION

- [Tenable.io Web App Scanning](#)
- [Mozilla Developer Network - Object prototypes](#)
- [Prototype pollution attack in NodeJS application](#)



Rémy Marot

Rémy joined Tenable in 2020 as a Senior Research Engineer on the Web Application Scanning Content team. Over the past decade, he led the IT managed services team of a web hosting provider and was responsible for designing and building innovative security services in a Research & Development team. He also contributed to open source security softwares, helping organizations increase their security posture.

Interests outside of work: Rémy enjoys spending time with his family, cooking and traveling the world. Being passionate about offensive security, he enjoys doing ethical hacking in his spare time.

RELATED ARTICLES





Government Advisories Warn of APT Activity Resulting from Russian Invasion of Ukraine

Government agencies publish warnings and guidance for organizations to defend themselves against advanced persistent threat groups. As governments around the world call for heightened cyber vigil...

[Team Tenable](#)

February 24, 2022

Tenable's Acquisition Of Cymptom: An "Attack Path-Informed" Approach to Cybersecurity

Tenable's recent acquisitions all had the same overarching goal: helping our customers gain better security insights across their cyberattack surface.

[Nico Popp](#)

February 17, 2022





Log4Shell: A Tale of Two Detection Techniques

Endpoint detection and response (EDR) can only take you so far in identifying Log4j exploit attempts. Here's why dynamic checks are needed to uncover vulnerable versions of Log4j.

[Glen Pendley](#)

February 15, 2022

ARE YOU VULNERABLE TO THE LATEST EXPLOITS?

Enter your email to receive the latest cyber exposure alerts in your inbox.

CYBER EXPOSURE

Overview

Lifecycle

Adaptive Assessment

Trust and Assurance

TENABLE.EP

Overview

TENABLE.IO

Overview





Overview

NESSUS

Overview

Nessus Professional

Nessus Essentials

Resource Center

TENABLE.OT

Overview

TENABLE.IO APPLICATIONS

Web Application Scanning

Container Security

PCI ASV

FEATURED SOLUTIONS

Application Security

Automotive Manufacturing

Building Management Systems

Cloud Security

Compliance

Energy

Finance

Healthcare

IT / OT

Legacy vs Risk-based VM Comparison

Medical Manufacturing

Oil & Gas

Ransomware

Retail

State / Local / Education

Transportation

US Federal

Vulnerability Management

Water

Zero Trust





[Customer Ambassador Program](#)

[Documentation](#)

[System Status](#)

[Security Advisories](#)

[GDPR Alignment](#)

[Trust and Assurance](#)

INVESTOR RELATIONS

[Corporate Profile](#)

[Stock Quote/Chart](#)

[News Releases](#)

[Investor Events](#)

[Presentations](#)

[SEC Filings](#)

[Annual Reports](#)

[Quarterly Results](#)

[Governance Highlights](#)

[Committee Composition](#)

[Analyst Coverage](#)

[Information Request](#)

[Email Alerts](#)

CONNECTIONS

[Blog](#)

[Contact Us](#)

[Newsletter Signup](#)

[Resource Library](#)

[Webinars](#)

[Research](#)

[Podcasts](#)

[VM Fundamentals](#)

GLOBAL

[English](#)

[Deutsch](#)

[Français \(France\)](#)

[Español \(América Latina\)](#)

[Português \(Brasil\)](#)





© 2022 Tenable®, Inc. All Rights Reserved | [Privacy Policy](#) | [Legal](#) | [508 Compliance](#)

