**tough-cookie 2.5.0 vulnerable**

What is Prototype pollution?

Prototype pollution is a vulnerability that enables threat actors to exploit JavaScript runtimes. In a prototype pollution attack, threat actors inject properties into existing JavaScript construct prototypes, attempting to compromise the application.

This vulnerability is called prototype pollution because it allows threat actors to inject values that overwrite or pollute the “prototype” of a base object. This malicious prototype can pass to many other objects that inherit that prototype. Once threat actors can control the default values of the object’s properties, they can tamper with the application’s logic. This can lead to denial of service (DoS) or remote code execution (RCE).

How Do Prototype Pollution Vulnerabilities Happen?

Prototype pollution was introduced when the European Association of Computer Manufacturers (ECMA) published the ECMAScript2015, 6th Edition specification. This specification created a JavaScript standard that ensures compatibility across various web browsers.

The specification includes standardization of the \_\_proto\_\_ feature, a special attribute that refers to Prototypes of an object. All JavaScript objects now have this attribute, and \_\_proto\_\_ itself is also defined as an object. “Prototype” refers to a mechanism that enables JavaScript objects to inherit features from one to another.

Everything you type in JavaScript (except primitives) is an object. Furthermore, an object Prototype may have other Prototypes nested under it, and can inherit Prototypes from other objects. This is referred to as a prototype chain.

Prototype pollution occurs when an attacker manipulates the \_\_proto\_\_ attribute, usually by adding a new Prototype into it. Because every JavaScript object now has the \_\_proto\_\_ attribute, and every object inherits Prototypes, whenever a Prototype is added, it is inherited by all objects in the prototype chain.

This means a malicious player could inject properties into existing JavaScript code to trigger a JavaScript exception causing denial of service, malicious code that achieves remote code execution, or other severe consequences.

Prototype Pollution Security Risks

JavaScript can run on the client-side and server-side of a web application, which is why prototype pollution vulnerabilities may exist on both sides. As a result, prototype attacks can vary greatly in scope and damage, depending on the application logic and implementation. Here are several examples:

Denial of Service (DoS)—a prototype pollution attack can enable threat actors to perform DoS against a targeted user in a client-side attack or on the web server that hosts the application.

Interfering with the JavaScript environment—a prototype pollution attack can add or modify a property to cause unexpected behaviors. It can result in some logic not executing as intended.

A starting point for additional attacks—a prototype pollution attack allows threat actors to leverage other components (Gadgets) loaded in the same context. It enables them to initiate complex attacks and potentially escalate their privileges or gain access to sensitive information.

Client-Side Exploitation

A prototype pollution exploitation starts when threat actors inject a payload into an input, like a URL, that builds the client-side logic or application rendering. For example, a URL parser can assign JavaScript objects properties without verifying if the target property is linked correctly to the Object prototype.

Client-side exploitation of a prototype pollution vulnerability can result in several attacks, such as cross-site scripting (XSS) attacks. In this case, threat actors look for a gadget that relies on the property of an object susceptible to pollution. If the object interacts with the page’s document object model (DOM), threat actors can trigger client-side JavaScript code execution.

Server-Side Exploitation

Server-side exploitation occurs when threat actors exploit prototype pollution vulnerabilities to modify the Object prototype’s properties using gadgets located in the application context. In this case, the modified JavaScript runtime objects are executed on the server, leading to a more severe impact than client-side attacks.

Exploiting prototype pollution vulnerabilities can be complex. It typically requires a deep analysis of the application logic to determine the attack’s impact. However, server-side exploitation can lead to severe consequences, including remote code execution (RCE), SQL injection (SQLi), and authorization and authentication bypass.

Prototype Pollution Mitigation

Use Safe Open-Source Libraries

Open-source libraries can use multiple practices that expose them to prototype pollution:

Merging two objects recursively

Any function that recursively sets nested properties

Deep cloning of objects

Creating objects by recursively zipping properties with values

Whenever a library recursively sets a nested property, it must sanitize the untrusted inputs. It is important to carefully vet open-source libraries and ensure your project does not use a library that is exposed to these weaknesses.

Create Objects without Java Prototypes

Another technique that lets you avoid prototype pollution is Object. Create(). You can use this method instead of the object constructor Object ()or the object-literal {} when you create a new object. This approach lets you set the created object’s prototype directly using the first argument passed to the Object.create()element. If you pass a null value, the created object won’t have a prototype and will not be pollutable.

Prevent All Changes to the Prototype

Another method that JavaScript supports is Object.freeze(). It lets you block all changes to an object’s attributes. Given that a prototype is simply an object, you can freeze it as you would a regular object. You can invoke the Object.freeze(Object.prototype)element to freeze the default prototype and prevent it from being polluted.

You can also install the nopp npm package to automatically freeze all common object prototypes.

Preventing JavaScript Attacks with Imperva

Imperva Runtime Application Self-Protection (RASP) and Imperva Web Application Firewall (WAF) can mitigate attempts to exploit a prototype pollution vulnerability.

Runtime Application Self-Protection (RASP) – Real-time attack detection and prevention from your application runtime environment goes wherever your applications go. Stop external attacks and injections and reduce your vulnerability backlog.

Web Application Firewall – Prevent attacks with world-class analysis of web traffic to your applications.

Imperva provides comprehensive protection for applications, APIs, and microservices:

API Security – Continuous protection for all APIs using deep discovery and classification of sensitive data to detect all public, private and shadow APIs to empower security teams to implement a positive security model.

Advanced Bot Protection – Prevent business logic attacks from all access points – websites, mobile apps and APIs. Gain seamless visibility and control over bot traffic to stop online fraud through account takeover or competitive price scraping.

DDoS Protection – Block attack traffic at the edge to ensure business continuity with guaranteed uptime and no performance impact. Secure your on premises or cloud-based assets – whether you’re hosted in AWS, Microsoft Azure, or Google Public Cloud.

Attack Analytics – Ensures complete visibility with machine learning and domain expertise across the application security stack to reveal patterns in the noise and detect application attacks, enabling you to isolate and prevent attack campaigns.

**tough-cookie 2.5.0 CVE-2023-26136, Security Risk #282**

Vulnerability occurs by using CookieJar in rejectPublicSuffixes=false mode.

* (*rejectPublicSuffixes* - boolean - default true - reject cookies with domains like "com" and "co.uk")

Prototype Pollution can occur by passing Domain=\_\_proto\_\_  
It comes from this point in the code:

this.idx = {}

...........

putCookie(cookie, cb) {

if (!this.idx[cookie.domain]) { // this.idx[\_\_proto\_\_]

this.idx[cookie.domain] = {};// this.idx[\_\_proto\_\_]= {} you are define using in prototype

}

if (!this.idx[cookie.domain][cookie.path]) {

this.idx[cookie.domain][cookie.path] = {};

}

this.idx[cookie.domain][cookie.path][cookie.key] = cookie;

cb(null);

}

this.idx is an Object  
If cookie.domain equals to \_\_proto\_\_ then you override global object prototype.  
It is non-obvious behavior. It also carries certain risks.

Solution:

Upgrade tough-cookie to version 4.1.3 or higher Or if you feel shorthanded (pun intended) and don't have the necessary capacity use my fixed code.

**References:**

|  |  |
| --- | --- |
| **Hyperlink** | **Resource** |
| https://github.com/salesforce/tough-cookie/commit/12d474791bb856004e858fdb1c47b7608d09cf6e | Patch |
| https://github.com/salesforce/tough-cookie/issues/282 | Exploit Issue |
| https://github.com/salesforce/tough-cookie/releases/tag/v4.1.3 | Release Notes |
| https://lists.debian.org/debian-lts-announce/2023/07/msg00010.html |  |
| https://lists.fedoraproject.org/archives/list/package-announce@lists.fedoraproject.org/message/3HUE6ZR5SL73KHL7XUPAOEL6SB7HUDT2/ |  |
| https://lists.fedoraproject.org/archives/list/package-announce@lists.fedoraproject.org/message/6PVVPNSAGSDS63HQ74PJ7MZ3MU5IYNVZ/ |  |
| https://security.netapp.com/advisory/ntap-20240621-0006/ |  |
| https://security.snyk.io/vuln/SNYK-JS-TOUGHCOOKIE-5672873 | Exploit Technical Description Third Party Advisory |