Demonstrating the dangers of Prototype pollution in different types of applications

DD2525 – Language Based Security

Michael Hartmann, Christopher Gauffin

Group 12

October 9, 2022

Contents

1 Background **2**

2 Goal **2**

3 Method **2**
3.1 Analysis of the vulnerability . . . 2
3.2 Spread . . . 2
3.3 Protection techniques . . . 3

4 NodeJS Chat application **3**
4.1 Documentation . . . 3
4.1.1 Installation . . . 3
4.1.2 Running normally . . . 3
4.1.3 Running with exploit . . . 3
4.1.4 Running with mitigation . . . 4
4.2 Exploitation . . . 4
4.3 Mitigation . . . 4
4.3.1 Less generic payloads . . . 4
4.3.2 Restricting object properties . . . 4

5 Blog with XSS filter **4**
5.1 Documentation . . . 4
5.1.1 Installation . . . 5
5.1.2 Running normally . . . 5
5.1.3 Running with exploit . . . 5
5.1.4 Running with mitigation . . . 5
5.2 Exploitation . . . 5
5.3 Mitigation . . . 5

6 Summary **6**

7 Contributions **6**

1 Background

Object-oriented programming languages often use prototype-based inheritance by reusing existing objects that serve as prototypes. This means that if the prototype of a variable is changed it is reflected globally within the program for all other variables that also use that prototype. Prototype pollution is a vulnerability where an attacker may modify an object's prototype at runtime and trigger the execution of malicious code.

2 Goal

Explain, study and practically demonstrate how Prototype pollution works and show where the problem stems from. We do this by creating one or more modern applications and showing what the unintended consequences might be if this vulnerability is not considered. Then we show examples of how we can protect ourselves against these types of vulnerabilities.

3 Method

Prototype pollution is an attack on JavaScript applications aiming at the alteration of the so called "prototype" which is a JavaScript feature that provides something akin to object based inheritance. This prototype contains basic functions like ToString() etc. which can then be used by all objects.

3.1 Analysis of the vulnerability

While inheritance from a prototype object can provide great flexibility, it can also lead to application-wide changes when this prototype object is altered. To conduct the attack the malicious actor needs to get control of an object, since usually every object has the __proto__ property or the constructor.prototype property, through which the prototype can be accessed. By providing malicious input to vulnerable functions the attacker can make changes to the properties of the prototype e.g. by replacing the ToString() function with something else. The changed properties would now apply to all current and all future objects sharing this prototype, which don't explicitly overwrite the changes.

To actually carry out this attack the malicious actor requires some sort of entry point which allows him to define or redefine properties. Server-side applications such as an API or socket server that are vulnerable to Prototype pollution often contain functions like recursive object merging or property definition by path. Here the attacker needs to be able to control one input value of the function to be able to perform the attack.

On the client-side this exploit can be used to, in some cases, bypass security mechanisms aimed at the prevention of XSS attacks like HTML sanitizers or similar tools. On the server-side this exploit can easily lead to Denial of Service since the added or altered properties of the prototype can crash the application. Additionally there is in some cases also the possibility of remote code execution. Since every application behaves differently, Prototype pollution attacks need to be tailored to the specific application if the attacker wants to do anything other than a simple DoS.

3.2 Spread

While it is not possible to precisely tell how widespread this vulnerability actually is at the moment, it is safe to assume that a general lack of knowledge about it, as well as a certain amount of ignorance towards it, still lead to the deployment of vulnerable applications some of which are not fixed until this day even after their problems were made public[5]. To find out whether a library is potentially susceptible to prototype pollution attacks, one can follow the approach described by Olivier Arteau[3] which lists all functions of a installed library and looks for dangerous merge and copy functions as well as property assignment by path.

3.3 Protection techniques

The easiest mitigation of the Prototype pollution vulnerability is to freeze the prototype. The Object.freeze function prevents any change of the object by silently failing any attempt at modification.

Another way of protection, when the input comes via JSON, is to use Schema validation, which means running a check on the input that rejects any unnecessary attributes.

Since it is a object based vulnerability, it is also possible to avoid the usage of objects in general. An alternative to Object was introduced in the EcmaScript 6 standard which is Map. In most cases Map can be used in the same ways as Object, but provides more security.

It is also possible to avoid the existence of the Prototype property in objects, by initializing them with Object.create(null). This way an object will be created without the __proto__ and constructor attributes.

Each of these techniques will of course lead to a little less flexibility for the developer which is why they are not by default activated.

4 NodeJS Chat application

This application is similar to a lightweight Discord clone, where we have a Socket IO server and client communication, where multiple clients can join and send messages to each other. The server side is vulnerable because it naively merges objects with client data.

4.1 Documentation

4.1.1 Installation

- Open new terminal:
  git clone https://github.com/Christopher96/lang-sec-chat && cd lang-sec-chat

4.1.2 Running normally

- Open three new terminals and navigate to the installation directory then run the following in each terminal:
- **cd server && npm i && node server.js**
- **cd clientA && npm i && node client.js**
- **cd clientB && npm i && node client.js**

4.1.3 Running with exploit

- Open three new terminals and navigate to the installation directory then run the following in each terminal:
- **cd server && npm i && node server.js**
- **nc -l localhost -p 1337**
- **cd malicious && npm i && node client.js**
- Select command "Version"
- You should now have a reverse shell in the terminal running netcat (nc)

4.1.4 Running with mitigation

- **git checkout mitigation-freeze**
- Do the same as above, notice that the attack no longer works. (the prototype is frozen in *server/server.js*)

4.2 Exploitation

This exploit is inspired by an article from HackTricks which demonstrates a method of gaining a reverse shell through Prototype pollution that is still viable with the latest version of NodeJS. [1] The way we pollute the prototype in this exploit is by abusing a dangerous function that is used on the server side to merge objects together, called *merge* (*server/server.js*). Every key-value pair of an object is copied over to another, where it checks whether a property is an object or not. This may seem like it is safe to do, but it is in fact not because if copy over a key-value pair with __proto__ as the key then the prototype will get polluted. The *merge* function is used whenever a user wants to update their data, for example if they change their name send the new data with the socket event *COMMAND* and command *CHANGE_DATA* from the client to the server, then the server will try to merge and update the old data with the new. This works well if the client is benign and has no malicious intent, however if they do, they could easily manipulate the data being sent to the server and pollute the object prototype on the server side by editing their configuration file (*malicious/saved.json*). Now, if we send over the malicious data, the server will pollute *Object.__proto__.env* with a new environment variable containing the malicious code to gain RCE as well as *Object.__proto__.NODE_OPTIONS* with "–require /proc/self/environ" which tells node to include the environment variables whenever a new process is spawned. Polluting the prototype is bad in itself but now we can set up a new socket on the malicious side to listen for incoming connections. Whenever the *VERSION* event is executed on the server it will spawn another process checking for the current version, this will however also run the environment variables and naively connect to the malicious user, giving them full access to the server.

4.3 Mitigation

There are multiple approaches to mitigating this type of pollution:

4.3.1 Less generic payloads

By sending all of the data each time the client changes something, we use the lazy approach of simply updating the object server side corresponding to the client with all of the data. It would be more safe to have specific events for each action such as *CHANGE_USERNAME* for changing the username, and then explicitly changing the *.username* property of the server object. This way there is no possibility to take advantage of an assignment of the type *obj[key] = value*, as it is no longer part of the code.

4.3.2 Restricting object properties

If we still would like to merge objects without having to create a single event for each property we could whitelist the property names that we consider safe. This could be implemented with an array consisting of for example "username", "age", "interests", where only those are properties are allowed to be merged, otherwise the server will respond with an error code. Let us also consider the case when we have hundreds of properties which are constantly changing and we can not be bothered with updating the whitelist, we could instead freeze specific properties, which would disallow them to be changed at all, such as "prototype", "constructor" and "__proto__".

5 Blog with XSS filter

5.1 Documentation

This is an example of a normal blog built on HTML, JS and CSS, it is served with a simple webserver with NodeJS. We exploit the HTML sanitization to achieve XSS.

5.1.1 Installation

- Open new terminal:
  git clone https://github.com/Christopher96/lang-sec-blog && cd lang-sec-blog

5.1.2 Running normally

- **npm i && node index.js**
- Go to *http://localhost:8080/blog.html* in the browser
- Click create new post
- Create a normal post, test that sanitization works by entering <img src onerror=alert("pwned")> as the text body.
- Check that the latest post does not generate any XSS

5.1.3 Running with exploit

- Do the same as above, but before creating the post, enter *Object.prototype.whiteList = { img : ["onerror", "src"] }* in the browser console.
- Observe XSS being triggered when the latest post is rendered.

5.1.4 Running with mitigation

- **git checkout mitigation-branch**
- Do the same as above, notice that the attack no longer works. (sanitization is done in *api/index.js*)

5.2 Exploitation

There are many sanitizers around which filter out dangerous HTML tags and attributes which might result in XSS (cross site scripting) vulnerabilities. We are going to take a closer look at "js-xss" [4] which has around 770k downloads every week [2]. This exploit is done client side as we take advantage of the fact that the sanitization is run within the browser and not server side. The blog that we are going to trick is run on ordinary HTML and javascript. A simple NodeJS server is both serving the HTML and an API running with Express JS accessed through the endpoint (*/api*). Whenever a post is created through the page (*/create.html*), the input is sanitizes before it is sent to the backend server as a POST request to (*/api/create*). This works well if the sanitization library is untampered with, as it will properly replace the characters with HTML entities in tags such as "<script>" and "<img>" and whatever else is whitelisted. A naive developer might think that this eliminates the vulnerability of XSS as the server will only store sanitized text, however we can very easily pollute the object prototype and overwrite the whitelist with for example *Object.prototype.whiteList = { img : ["onerror", "src"] }* and take advantage of the onerror attribute "<img src onerror=alert("pwned")>".

5.3 Mitigation

Sanitization on client side might be a good idea, but you can never be certain that what is being sent to the server is fully sanitized. In this case you would not even need to use the blog in the browser to send an API request to the server, opening up the possibility to send all kinds of malicious payloads. So to mitigate the attack we of course need a session cookie of some sort and proper authentication to begin with. But more importantly, to avoid XSS we also need to do the sanitization on the server side, where we can be certain that nothing is polluted, unless there is another source of vulnerability on the server side as we saw in the previous example. Given that the server code is secure we only store strings in our database that have been fully sanitized without any intervention by client pollution.

6 Summary

Prototype inheritance can be a very useful feature for object oriented programming, but it also comes with the risk of Prototype pollution which can be achieved in many different ways that the programmer may not have thought of. For JavaScript, when the application is polluted, every single object that is created will be affected. Objects can be polluted both server-side and client-side, the attacks may look a bit different but use the same exploitative idea, although it is often much more hazardous if the attacker gains access to the back-end on the server-side. We found that merge and path property definitions are dangerous for server-side applications and that the client-side often suffers from attacks against HTML sanitizers and by abusing both of these the application may be susceptible to Denial of Service or Remote Code Execution. We demonstrate one application built with NodeJS using Socket IO for communication. The back-end uses a merge function which can be manipulated to generate object pollution. This is used to change the environment variables and then effectively achieve Remote Code Exectuion whenever the server forks a new process. The mitigation involves removing the merge function and restricting what properties can be changed, or alternatively freezing properties. The other example involves a blog where HTML sanitization is done client-side, the attack eliminates the sanitization, which leads to unsanitized data being sent to the server when a post is created. The server naively serves content to the client which then renders the unsanitized data which results in a successful Cross Site Scripting attack. To mitigate the attack we recommend to always sanitize server side.

7 Contributions

Michael focused on general research around various different types of Prototype pollution and mitigation techniques, he also analyzed the spread and what the resulting damage may be if the vulnerability is not considered. Christopher developed the example applications and implemented the different practical use cases of Prototype pollution, as well as the mitigation's for such attacks.

References

[1] *Nodejs - __proto__ & prototype pollution*. May 2022. URL: https://book.hacktricks.xyz/pentesting-web/deserialization/nodejs-proto-prototype-pollution.

[2] *Prototype pollution – and bypassing client-side HTML sanitizers*. Aug. 2020. URL: https://research.securitum.com/prototype-pollution-and-bypassing-client-side-html-sanitizers/.

[3] *Prototype pollution attack in NodeJS application*. Oct. 2018. URL: https://raw.githubusercontent.com/HoLyVieR/prototype-pollution-nsec18/master/paper/JavaScript_prototype_pollution_attack_in_NodeJS.pdf.

[4] *Sanitize untrusted HTML (to prevent XSS) with a configuration specified by a Whitelist.* Feb. 2022. URL: https://www.npmjs.com/package/xss.

[5] *unfixed npm packages*. June 2022. URL: https://security.snyk.io/vuln/SNYK-JS-SDS-2385944.