

Client-side prototype pollution (CSPP)

One of the key technologies of the World Wide Web, along with HTML and CSS, is the programming language known as JavaScript. It is a programming language that is used extensively and frequently.

An attacker might add any properties to global prototypes via the JavaScript vulnerability known as prototype pollution, and user-defined objects could subsequently inherit those properties. The vulnerability affects prototype-based languages.

Only the JavaScript language has the prototype pollution vulnerability. As a result, we must first comprehend the JavaScript characteristics that contribute to the vulnerability before addressing it. We should first comprehend what objects are before intending to mimic a prototype pollution attack.

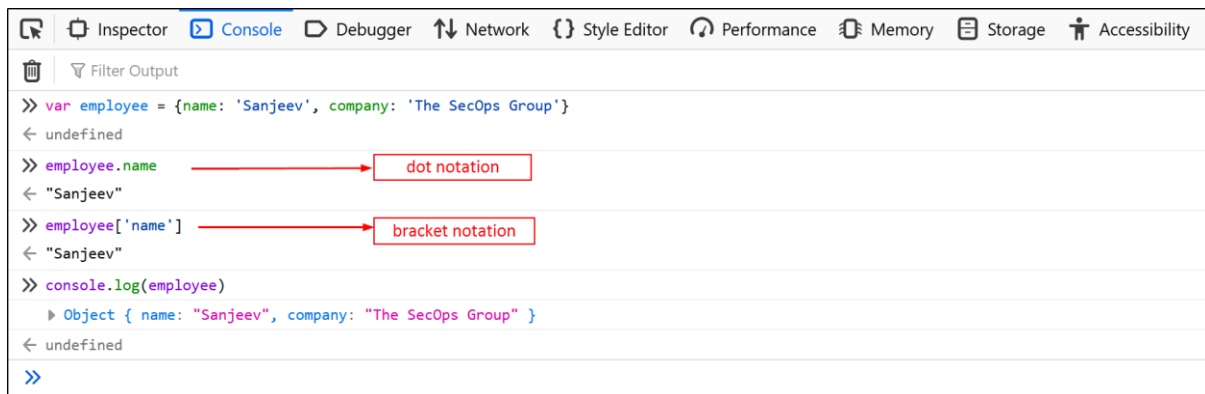
What is an object in JavaScript?

First, we must understand what an object is. Fundamentally, an object is essentially a collection of 'properties,' or key:value pairs. Let's imagine it represents an object called a employee, and the 'name' of the user, which is the key and has the value of 'Sanjeev' .

Open the developer tools in browser, then construct a simple object with three properties.

```
var employee = {name: 'Sanjeev', company: 'The SecOps Group'}
```

We can access the properties of an object by using either dot notation or bracket notation as shown below, In contrast, the console.log() method is used to log any messages or output to the console

A screenshot of a web browser's developer console. The console shows several lines of JavaScript code being executed. The first line is 'var employee = {name: 'Sanjeev', company: 'The SecOps Group'}', which returns 'undefined'. The second line is 'employee.name', which returns '"Sanjeev"', with a red arrow pointing to the text 'dot notation'. The third line is 'employee['name']', which also returns '"Sanjeev"', with a red arrow pointing to the text 'bracket notation'. The fourth line is 'console.log(employee)', which returns 'Object { name: "Sanjeev", company: "The SecOps Group" }'. The console interface includes tabs for Inspector, Console, Debugger, Network, Style Editor, Performance, Memory, Storage, and Accessibility. A 'Filter Output' button is visible at the top of the console panel.

```
>> var employee = {name: 'Sanjeev', company: 'The SecOps Group'}  
← undefined  
>> employee.name  
← "Sanjeev"  
>> employee['name']  
← "Sanjeev"  
>> console.log(employee)  
  ▶ Object { name: "Sanjeev", company: "The SecOps Group" }  
← undefined  
>>
```

In addition to data, properties can also have executable functions. In this case, the function is referred to as a 'method'. Now, try to add the HR() method to the employee object and call it.

```
< undefined
>> employee.name
< "Sanjeev"
>> employee['name']
< "Sanjeev"
>> console.log(employee)
  ▶ Object { name: "Sanjeev", company: "The SecOps Group" }
< undefined
>> employee.HR = function() { console.log("Human Resource") }
< ▶ function HR()
>> employee.HR()
  Human Resource
< undefined
```

Let's use the function 'toLocaleString()'. Though the employee object doesn't have a function 'toLocaleString()', all of a surprise it gets executed

```
>> employee.HR = function() { console.log("Human Resource") }
< ▶ function HR()
>> employee.HR()
  Human Resource
< undefined
>> employee.toLocaleString()
< "[object Object]"
```

How did the function 'toLocaleString()' get start ?, Let's understand this.

What is a prototype?

It is possible for JavaScript objects to inherit features from one another through the usage of a prototype, which is an attribute associated to an object. A prototype is an object since practically everything in JavaScript is an object. When new objects are formed in JavaScript, they inherit the properties and methods of the prototype "object," which includes fundamental features like function toLocaleString, toString, constructor, and hasOwnProperty.

There are many ways to discover an object's prototype, such as by using the Object.getPrototypeOf() method.

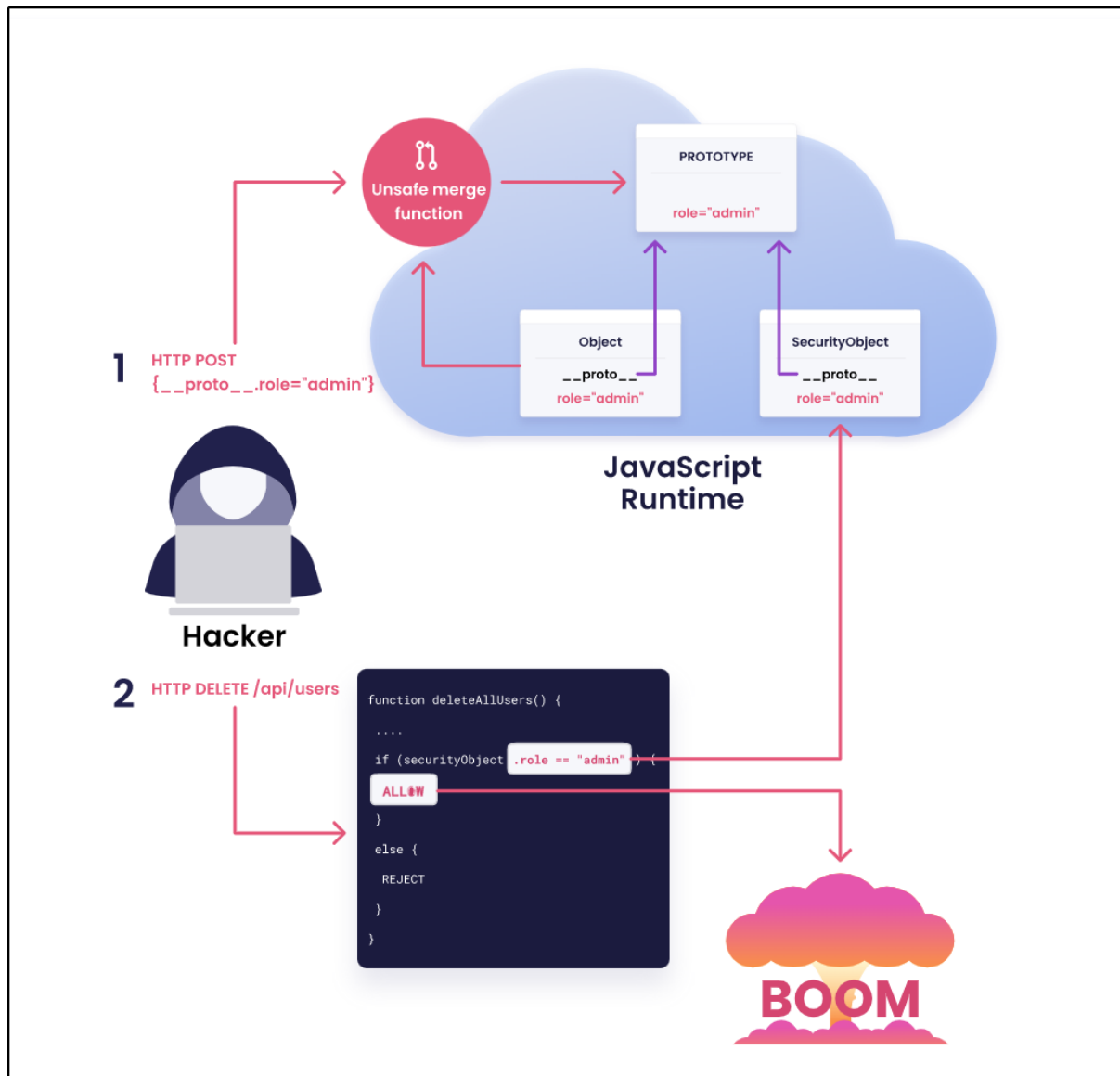
```
>> Object.getOwnPropertyNames(employee)
< ▶ Array(3) [ "name", "company", "HR" ]
>> Object.getPrototypeOf(employee)
< ▶ Object { ... }
  ▶ __defineGetter__: function __defineGetter__()
  ▶ __defineSetter__: function __defineSetter__()
  ▶ __lookupGetter__: function __lookupGetter__()
  ▶ __lookupSetter__: function __lookupSetter__()
  ▶ __proto__: >>
  ▶ constructor: function Object()
  ▶ hasOwnProperty: function hasOwnProperty()
  ▶ isPrototypeOf: function isPrototypeOf()
  ▶ propertyIsEnumerable: function propertyIsEnumerable()
  ▶ toLocaleString: function toLocaleString()
  ▶ toString: function toString()
  ▶ valueOf: function valueOf()
  ▶ <get __proto__(): function __proto__()
```

The prototype of an object is automatically added to 'Object.prototype' when it is created, therefore we were able to call function 'toLocaleString()'.

What is prototype pollution?

An attacker can add any properties to global object prototypes via the JavaScript vulnerability known as prototype pollution, and those values may then be inherited by user-defined objects.

When a JavaScript method iteratively combines an object with user-controllable properties into existing object without proper filtering or sanitization of the keys, prototype pollution vulnerabilities commonly result.



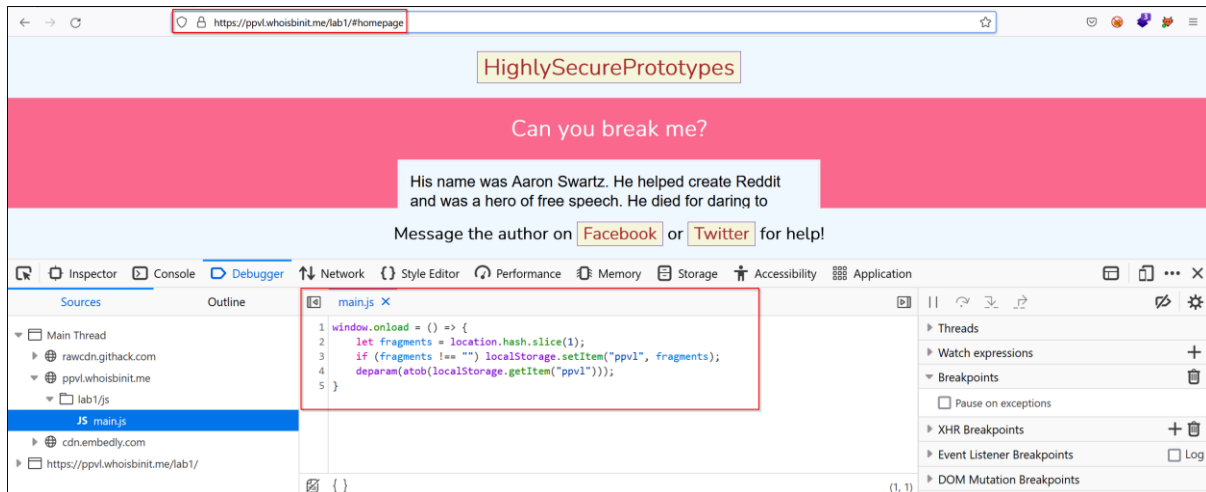
Let's try to find prototype pollution on a vulnerable site

Consider the following URL, which contains an attacker-constructed query string:

```
https://vulnerableWebsite.com/?_proto_[attackerProperty]=payload
```

Keyword `'_proto_'` may be interpreted by a URL parser as any string. But let's consider what happens if these keys and values are subsequently added as attributes to an existing object.

Let's test out one of the payloads using our hands: Open the developer tools and examine the JS file source code:



The window.onload method, which is used when a website is opened or a window is attempted to be loaded, can be seen being used in the main.js source code. Additionally, it transfers the value found in the URL fragments following the # symbol to the deparam file. Therefore, anything can be entered after the hash symbol (#).

First, let's learn from the browser console what happens if we enter the following payload into the console.

```
onload._proto_=alert("Your session has timed out. Please re-enter your password to resume:")
```

Wow, it was successfully executed, and the screenshot below demonstrates XSS :



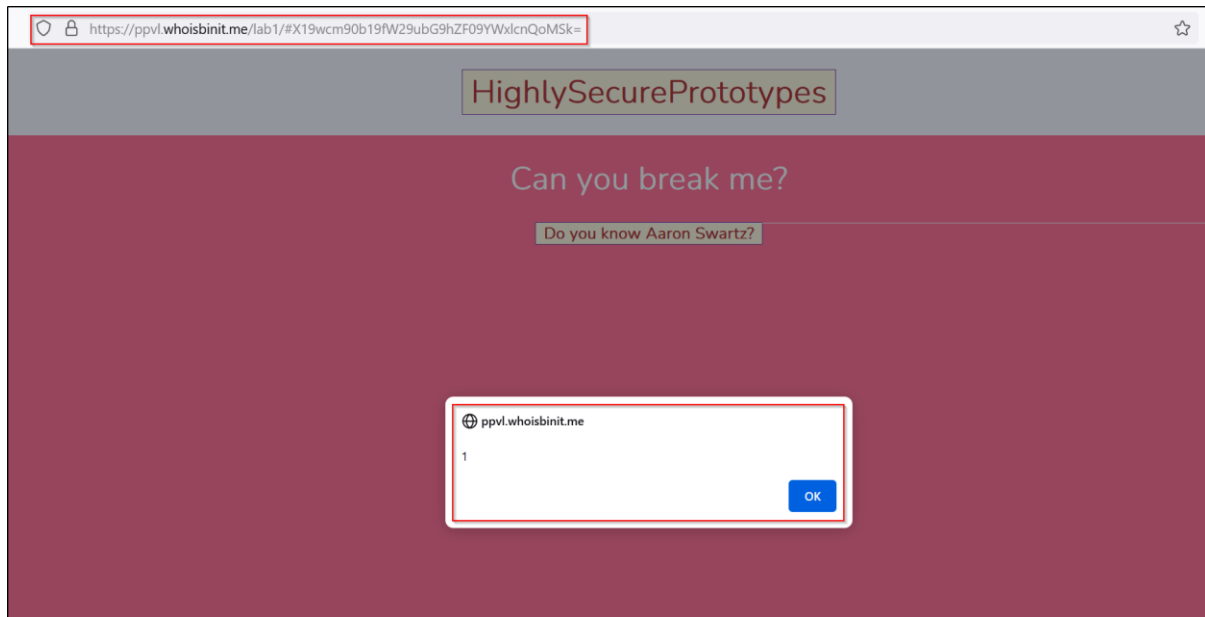
The above payload must now be modified in accordance with URL Standards, encoded, and sent as a GET request to any user of the vulnerable application.

Payload for URL without Encoding:

```
__proto__[onload]=alert(1)
```

Payload for URL with Base64Encoding:

X19wcm90b19fW29ubG9hZF09YWxlcuQoMSk=



Again, it got successfully executed and we got XSS, as shown in above screenshot.

For more payloads refer: <https://github.com/BlackFan/client-side-prototype-pollution>

How do you mitigate prototype pollution?

- Use `Object.create(null)` to completely avoid utilising prototypes or `Object.freeze(Object.prototype)` to stop any alterations to the shared prototype to better secure your code.
- Updating libraries with new patches
- Sanitizing user input is important since client-side vulnerabilities are the main source of vulnerability.
- The data structure map, which was added in EcmaScript 6 (ES6), is now widely supported and is a superior choice over objects because it functions as a hashmap (key value pairs) without all the security risks.
- Make use of a JavaScript package that uses a safe merge or extend method to recursively replicate properties from an unreliable source object.

Impact of Prototype Pollution

This could result in Denial of Service and XSS and other vulnerabilities, including RCE (Remote Code Execution), IDOR (Insecure Direct Object References), and Authentication Bypass.

Practice labs

- [Lab 1 | Prototype Pollution Vulnerable Labs \(whoisbinit.me\)](#)
- <https://tomnomnom.uk/pp/?page=home>
- <https://github.com/abhiabhi2306/prototype-pollution/>
- [https://msrqp.github.io \(Only XSS\)](https://msrqp.github.io (Only XSS))

- https://github.com/Dheerajmadhukar/Prototype-Pollution-Lab_me_dheeraj

Conclusion

JavaScript prototype pollution is a highly hazardous vulnerability that needs to be further researched in order to discover new vectors and devices (exploitation).

References

- <https://portswigger.net/web-security/prototype-pollution>
- <https://www.acunetix.com/vulnerabilities/web/prototype-pollution/>
- <https://learn.snyk.io/lessons/prototype-pollution/javascript>
- <https://blog.s1r1us.ninja/research/PP>
- <https://github.com/HoLyVieR/prototype-pollution-nsec18>
- <https://github.com/BlackFan/client-side-prototype-pollution>
- <https://bugcrowd.com/disclosures/57b28008-4653-4dec-88c3-4d38e40023ff/toolbox-teslamotors-com-html-injection-via-prototype-pollution-potential-xss>