**Introduction to Prototype Pollution**

After knowing how JavaScript instantiates objects and what prototypes are in the previous section, let us discuss prototype pollution.

**Prototype Pollution**

[Prototype Pollution](https://learn.snyk.io/lessons/prototype-pollution/javascript/) is a vulnerability that can arise under specific conditions when vulnerable code or libraries are used. Depending on the implementation of the vulnerable function, prototype pollution can lead to Denial-of-Service (DoS), privilege escalation, remote code execution, or any other common web vulnerability.

Since the prototype of an object is just a reference to another object, we can edit the properties of the prototype just like we can edit properties of any object by accessing the \_\_proto\_\_ property, which references our object's prototype. Consider our previously used module object again, without the shadowed toString property. We can change the toString function of our modules prototype, which is the Object.prototype object that all objects inherit from, like so:

Code: javascript

module.\_\_proto\_\_.toString = function () {return "shadowed";}

Now, if we instantiate an entirely different object and call its toString function, it uses the property we provided since we changed the property in the Object.prototype object, and our newly instantiated object inherits the toString property from that object:



Prototype pollution occurs if we can set a property in an object's prototype when it is not intended. Depending on the actual implementation of the vulnerable code, this can lead to privilege escalation, remote code execution, or other vulnerabilities, as we will discuss in the following sections.

As a simple baseline example for prototype pollution, consider the following code:

Code: javascript

function Module(name, author, tier) {

this.name = name;

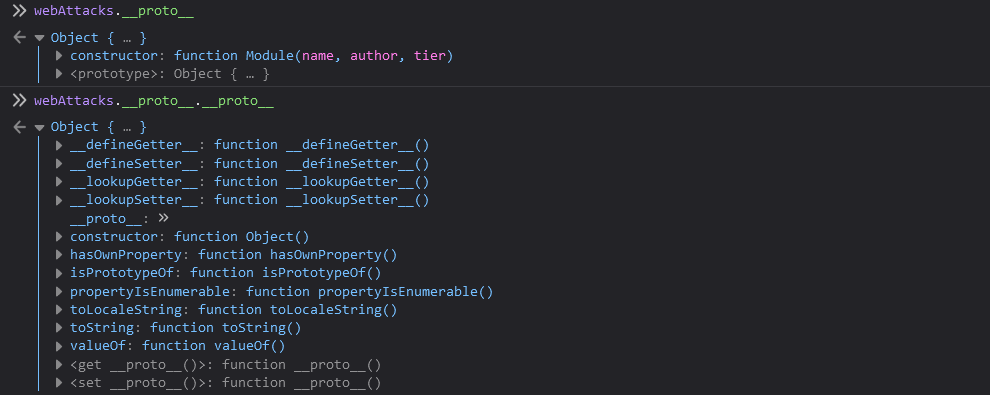
this.author = author;

this.tier = tier;

}

var webAttacks = new Module("Web Attacks", "21y4d", 2)

With the new operator, we can instantiate an instance of the Module type we defined in the function with the same name. Now let us consider a scenario where we can set an arbitrary property of the webAttacks object to pollute the Object.prototype object, which all JavaScript objects inherit from. In order to reach the Object.prototype object, we need to traverse two steps up the prototype chain since the prototype of the webAttacks module is the Module function:

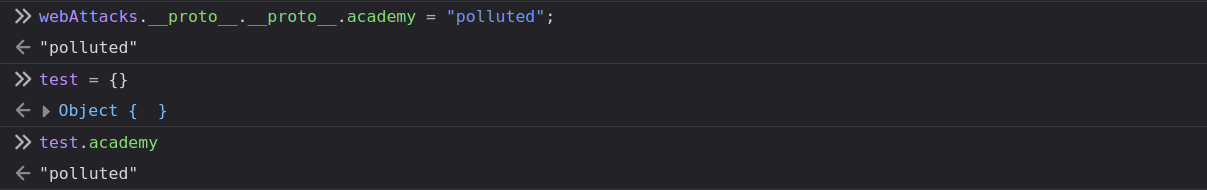


For instance, if we want to pollute the academy property, we could use the following payload:

Code: javascript

webAttacks.\_\_proto\_\_.\_\_proto\_\_.academy = "polluted";

This successfully pollutes the property for all newly instantiated objects:



**Prototype Pollution Vulnerabilities**

While changing properties of an object's prototype can be intended, prototype pollution vulnerabilities arise when user input is used in such a way that it enables prototype pollution with dangerous consequences.

Prototype pollution vulnerabilities typically arise when user input is used to set properties of existing objects. As an example, consider a web application that operates on the module objects we used as an example before in this section. The web application accepts user input in JSON format to add data to these module objects such as comments. For this, the user sends a request with the following JSON body:

Code: json

{"comment": "Great module."}

After receiving this request, the web application sets the comment property of the corresponding module object. However, the web developer wants to support arbitrary keys instead of hardcoding the comment property to allow for the support of new properties in the future. As such, the developer might implement the following function to merge the user-supplied JSON object with the existing module object:

Code: javascript

// helper to determine if recursion is required

function isObject(obj) {

return typeof obj === 'function' || typeof obj === 'object';

}

// merge source with target

function merge(target, source) {

for (let key in source) {

if (isObject(target[key]) && isObject(source[key])) {

merge(target[key], source[key]);

} else {

target[key] = source[key];

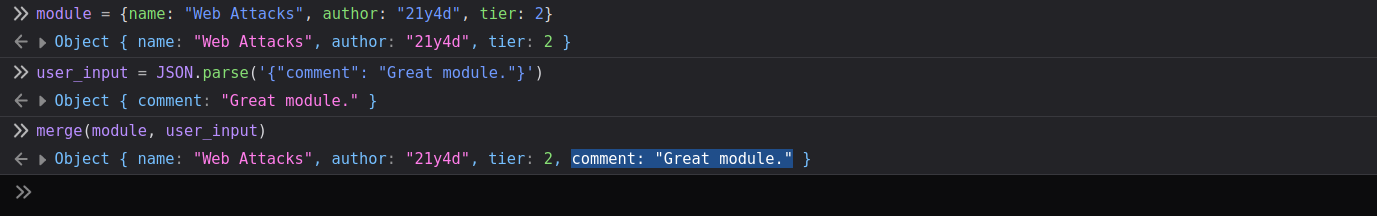
}

}

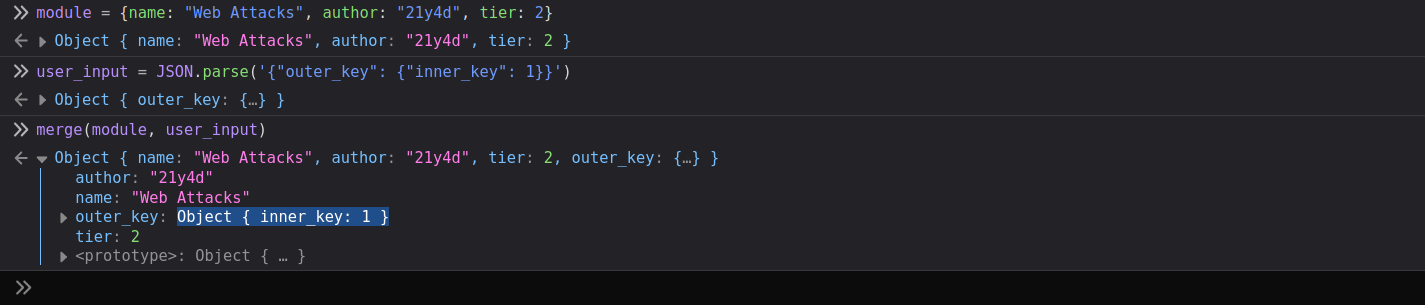
return target;

}

We can easily confirm that the function works as intended for the use case described above, as the comment property of the module object is correctly set:



However, due to the recursiveness of the function, it also supports more complicated merge tasks with objects within objects:

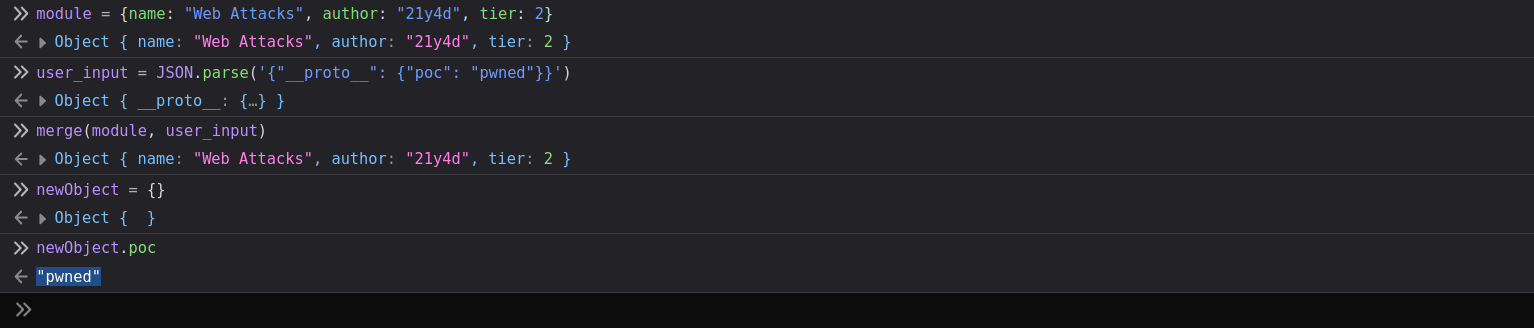


As such, the function is vulnerable to prototype pollution if the user-supplied JSON data contains the keyword \_\_proto\_\_. For instance, we can provide the following payload:

Code: json

{"\_\_proto\_\_": {"poc": "pwned"}}

Merging this user input with any existing object without any sanitization results in prototype pollution for all newly created objects:



As we can see in the screenshot above, the poc property exists for the newObject object we created after the merge function with the malicious payload was called. Since we successfully polluted the prototype, all newly created objects can now access this property via their prototype. Depending on how the web application uses specific properties and whether there is a lack of a default value, this can lead to various vulnerabilities, such as privilege escalation and remote code execution, as we will discuss in the upcoming sections.

You may wonder how common it is for developers to implement a function similar to the merge function showcased above. That is hard to tell, but many libraries provide similar functionality. Moreover, a vast list of these libraries was vulnerable to prototype pollution. For instance, check out this [list](https://raw.githubusercontent.com/HoLyVieR/prototype-pollution-nsec18/master/paper/JavaScript_prototype_pollution_attack_in_NodeJS.pdf). Functions related to merging and cloning objects are potentially susceptible to prototype pollution.