Object Prototype Pollution attack

---*---**

Submit by: SunCSR (Sun* Cyber Security Research)

I. Overview

1. What is prototype pollution

JavaScript is prototype-based: when new objects are created, they carry over the properties and methods of the prototype “object”, which contains basic functionalities such as toString, constructor and hasOwnProperty.

Object-based inheritance gives JavaScript the flexibility and efficiency that web programmers have come to love – but it also makes it vulnerable to tampering. Malicious actors can make application-wide changes to all objects by modifying object, hence the name prototype pollution.

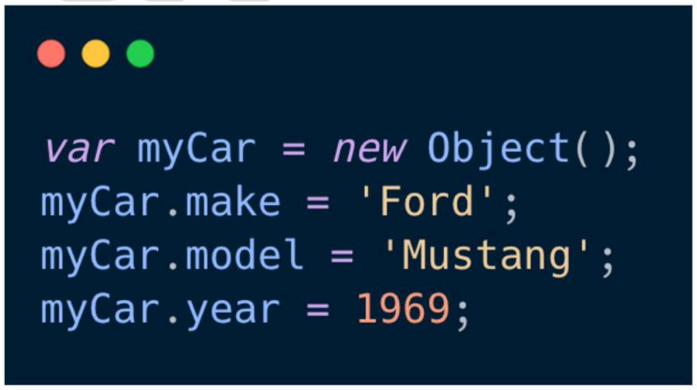
Interestingly, attackers don’t even need to directly modify object – they can access it through the ‘__proto__’ property of any JavaScript object. And once you make a change to object, it applies to all JavaScript objects in a running application, including those created after tampering.

II. Object Prototype

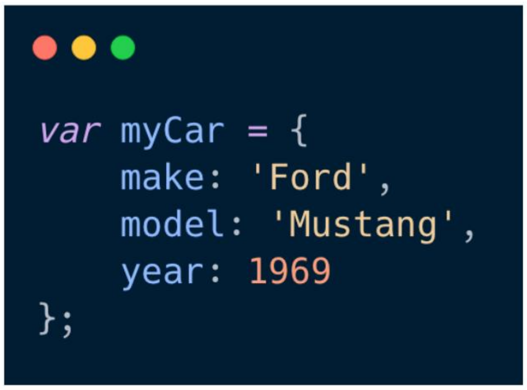
1. Object

JavaScript objects are containers for named values called properties or methods.

Two ways to define Object in JavaScript:



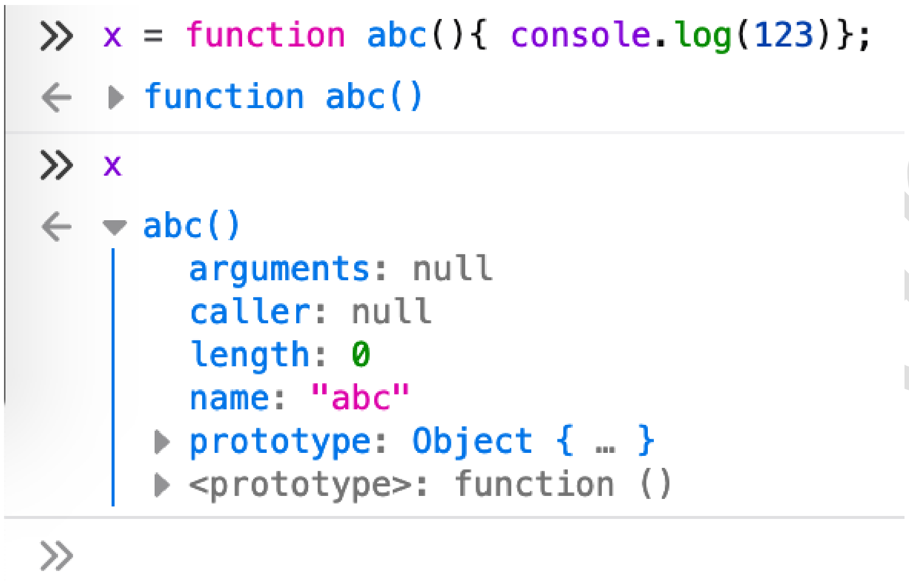
```
var myCar = new Object();
myCar.make = 'Ford';
myCar.model = 'Mustang';
myCar.year = 1969;
```



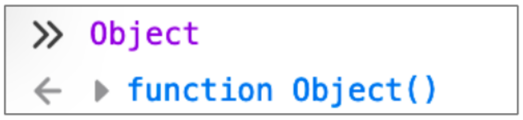
```
var myCar = {
    make: 'Ford',
    model: 'Mustang',
    year: 1969
};
```

2. Function

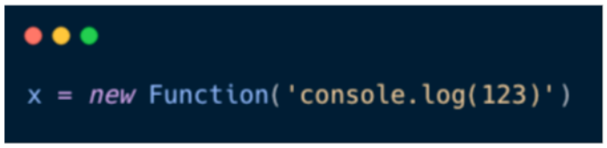
In JavaScript functions are also objects, which can be constructed using its own constructor which is Function



```
>> x = function abc(){ console.log(123)};
<- ▶ function abc()
>> x
<- ▼ abc()
      arguments: null
      caller: null
      length: 0
      name: "abc"
    ▶ prototype: Object { … }
    ▶ <prototype>: function ()
>>
```



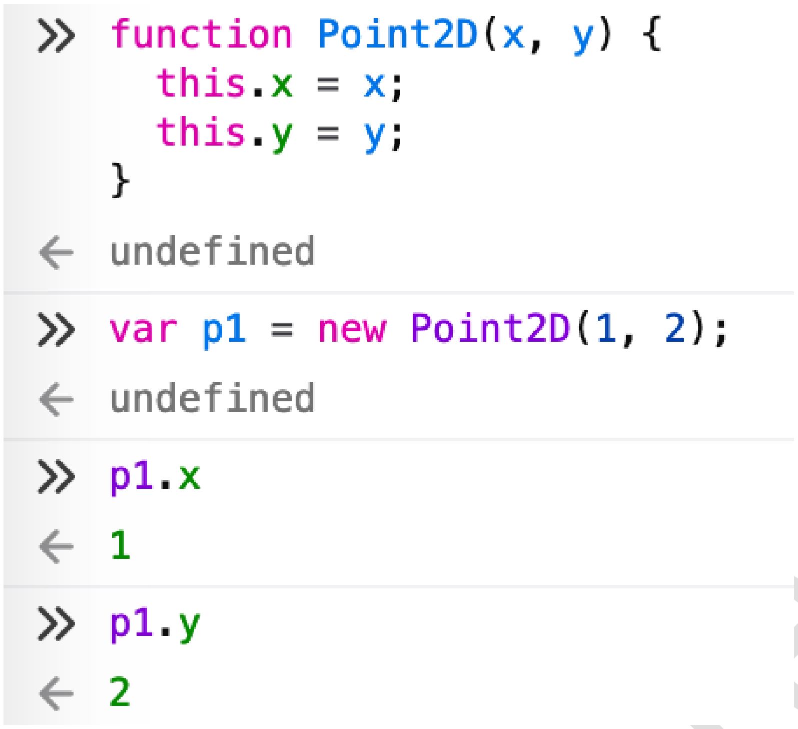
```
>> Object
<- ▶ function Object()
```



```
x = new Function('console.log(123)')
```

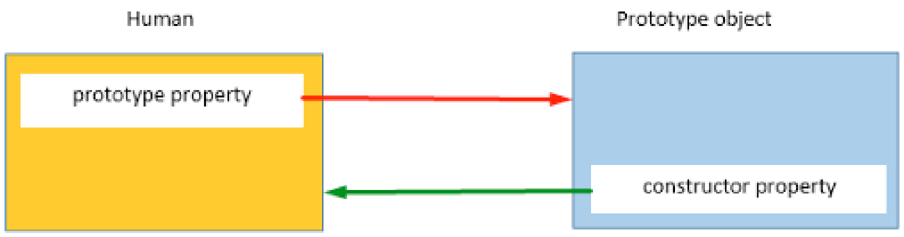
3. Constructor Function

- Constructor functions are templates for creating objects. We can use it to create different objects using the same constructor, which has the same instance methods and properties with different values for the non-method properties
- **this** keyword
- Objects of the same type are created by calling the constructor function with the new keyword



```
>> function Point2D(x, y) {
     this.x = x;
     this.y = y;
   }
<- undefined
>> var p1 = new Point2D(1, 2);
<- undefined
>> p1.x
<- 1
>> p1.y
<- 2
```

4. prototype and constructor

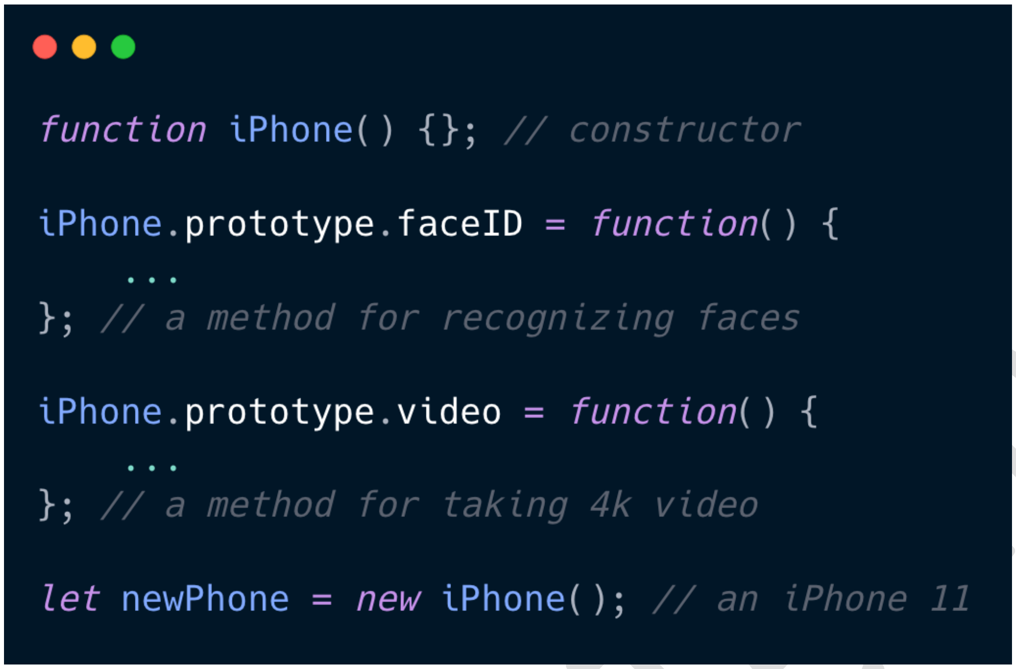


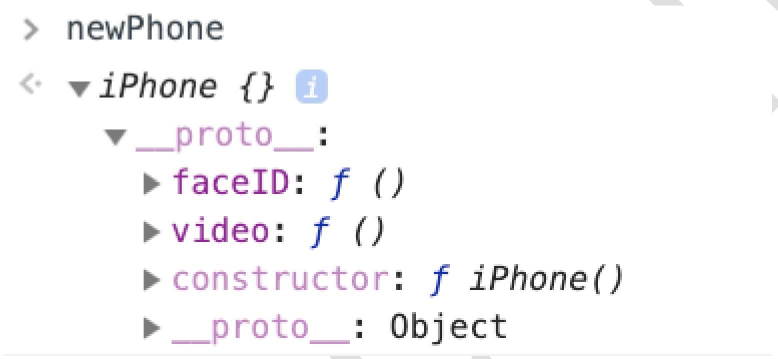
Point 2D (function)

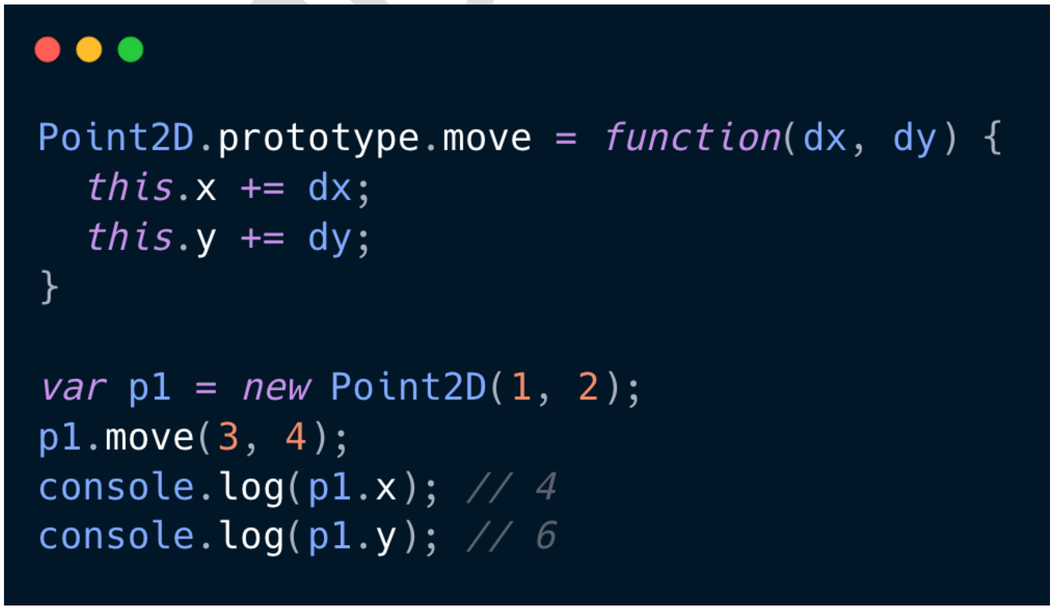
Point2D.prototype.constructor = Point2D (function)

5. Prototype

- Prototypes are the mechanism by which JavaScript objects inherit features from one another. In this article, we explain how prototype chains work and look at how the `prototype` property can be used to add methods to existing constructors.
- All JavaScript objects inherit properties and methods from a prototype.

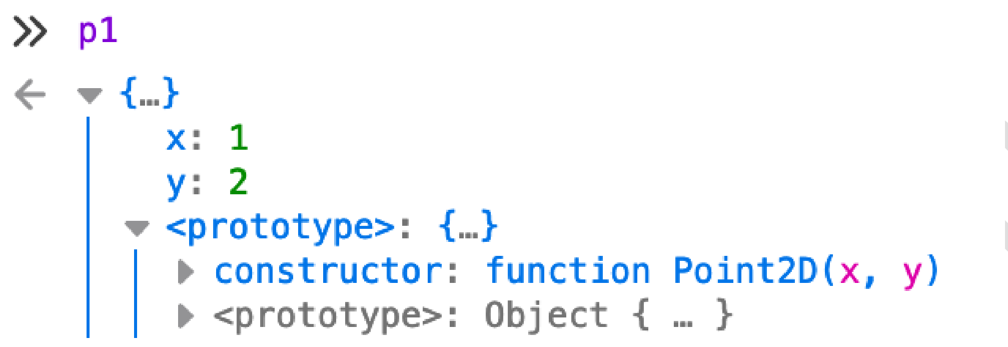

```
function iPhone() {}; // constructor

iPhone.prototype.faceID = function() {
    ...
}; // a method for recognizing faces

iPhone.prototype.video = function() {
    ...
}; // a method for taking 4k video

let newPhone = new iPhone(); // an iPhone 11
```


```
> newPhone
<· ▼iPhone {} i
    ▼__proto__:
      ▶faceID: ƒ ()
      ▶video: ƒ ()
      ▶constructor: ƒ iPhone()
      ▶__proto__: Object
```


```
Point2D.prototype.move = function(dx, dy) {
   this.x += dx;
   this.y += dy;
}

var p1 = new Point2D(1, 2);
p1.move(3, 4);
console.log(p1.x); // 4
console.log(p1.y); // 6
```

Create Object

- A default property named prototype that:
- Is an Object
- Constructor property is constructor function

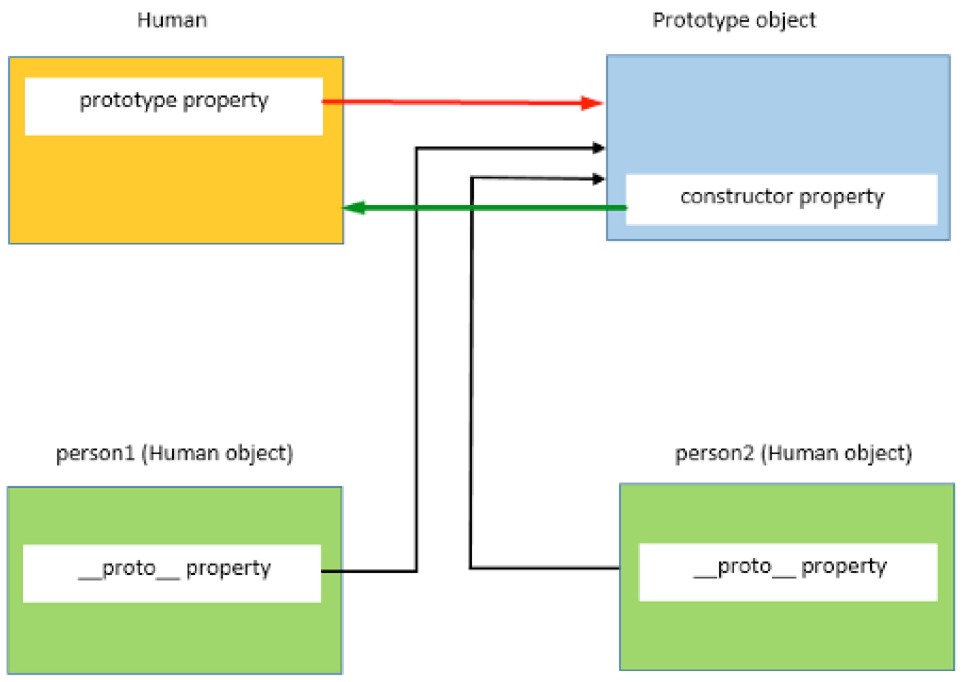


```
>> p1
<- ▼ {…}
       x: 1
       y: 2
     ▼ <prototype>: {…}
       ▶ constructor: function Point2D(x, y)
       ▶ <prototype>: Object { … }
```

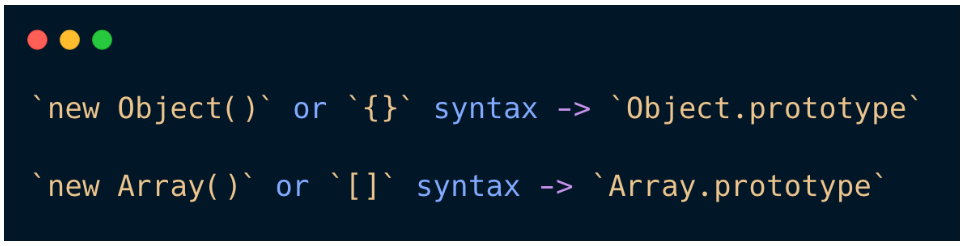
6. prototype and __proto__

In reality, the only true difference between prototype and __proto__ is that the former is a property of a class constructor, while the latter is a property of a class instance.

p1.__proto__ === Point2D.prototype

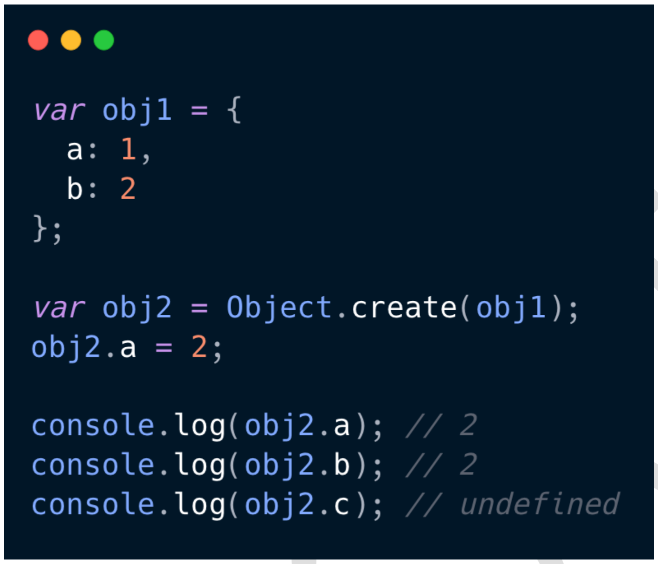


Who is my parent?

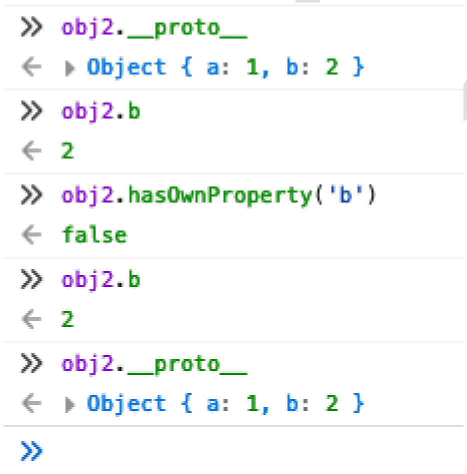


```
`new Object()` or `{}` syntax -> `Object.prototype`

`new Array()` or `[]` syntax -> `Array.prototype`
```

Property access



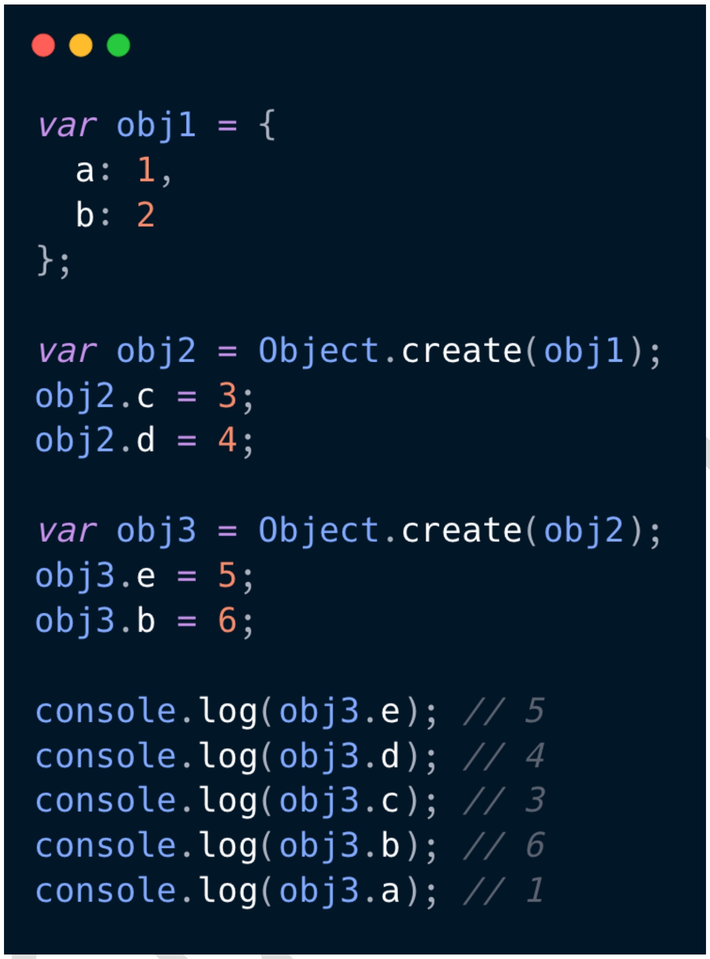
```
var obj1 = {
  a: 1,
  b: 2
};

var obj2 = Object.create(obj1);
obj2.a = 2;

console.log(obj2.a); // 2
console.log(obj2.b); // 2
console.log(obj2.c); // undefined
```



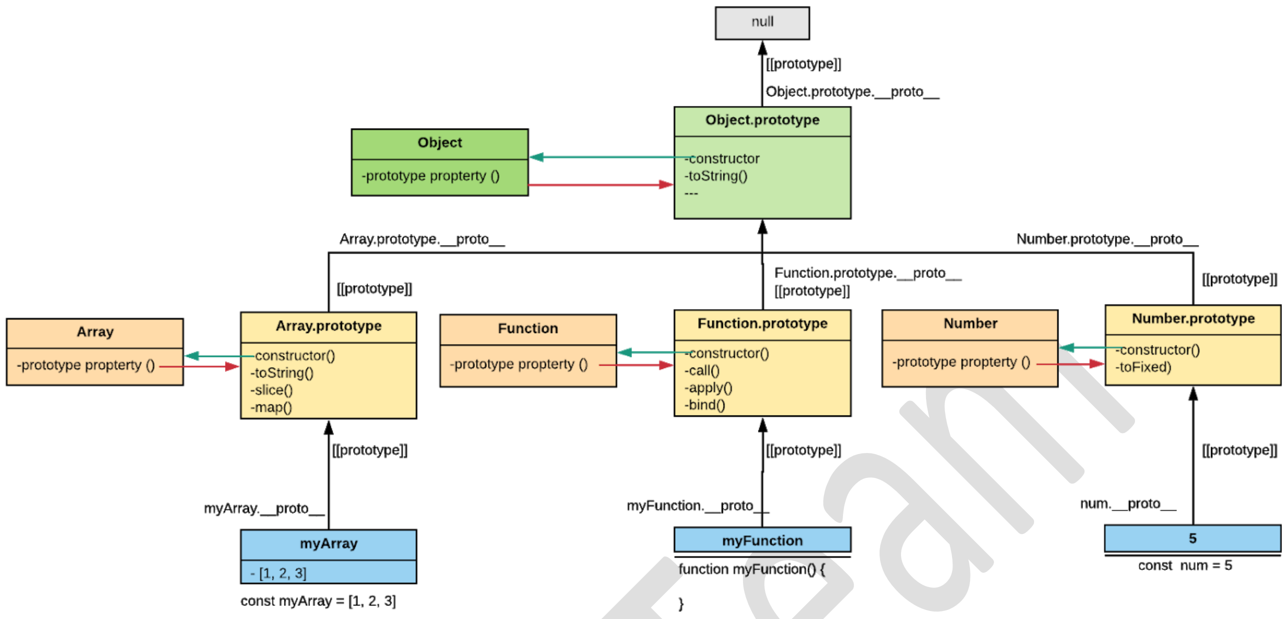
```
>> obj2.__proto__
←  ▶ Object { a: 1, b: 2 }
>> obj2.b
←  2
>> obj2.hasOwnProperty('b')
←  false
>> obj2.b
←  2
>> obj2.__proto__
←  ▶ Object { a: 1, b: 2 }
>>
```

7. Prototype chain

JavaScript objects have a link to a prototype object. When trying to access a property of an object, the property will not only be sought on the object but on the prototype of the object, the prototype of the prototype, and so on until either a property with a matching name is found or the end of the prototype chain is reached.



```javascript
var obj1 = {
  a: 1,
  b: 2
};

var obj2 = Object.create(obj1);
obj2.c = 3;
obj2.d = 4;

var obj3 = Object.create(obj2);
obj3.e = 5;
obj3.b = 6;

console.log(obj3.e); // 5
console.log(obj3.d); // 4
console.log(obj3.c); // 3
console.log(obj3.b); // 6
console.log(obj3.a); // 1
```

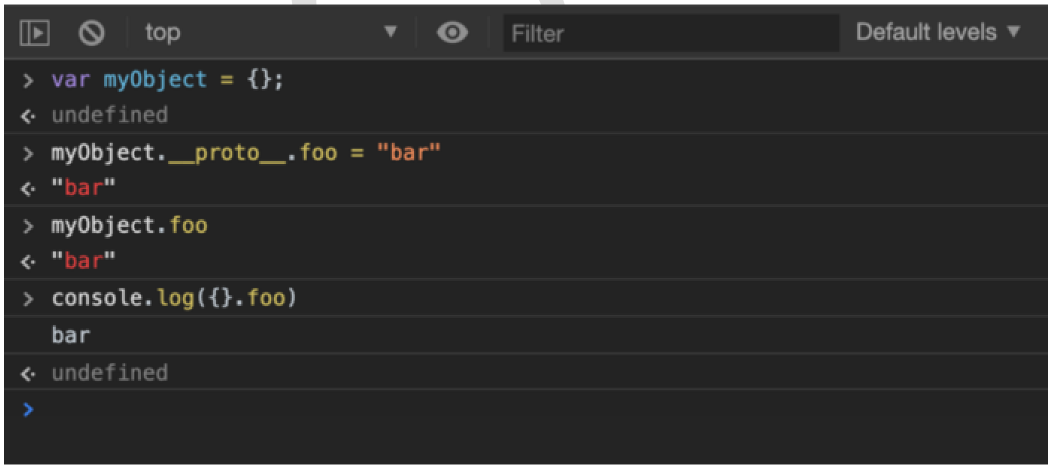


III. Object Prototype attack

1. How can you find it?

Add new unexpected property to Object.prototype to cause unexpected behavior

Everything in JavaScript is inheriting from Object.

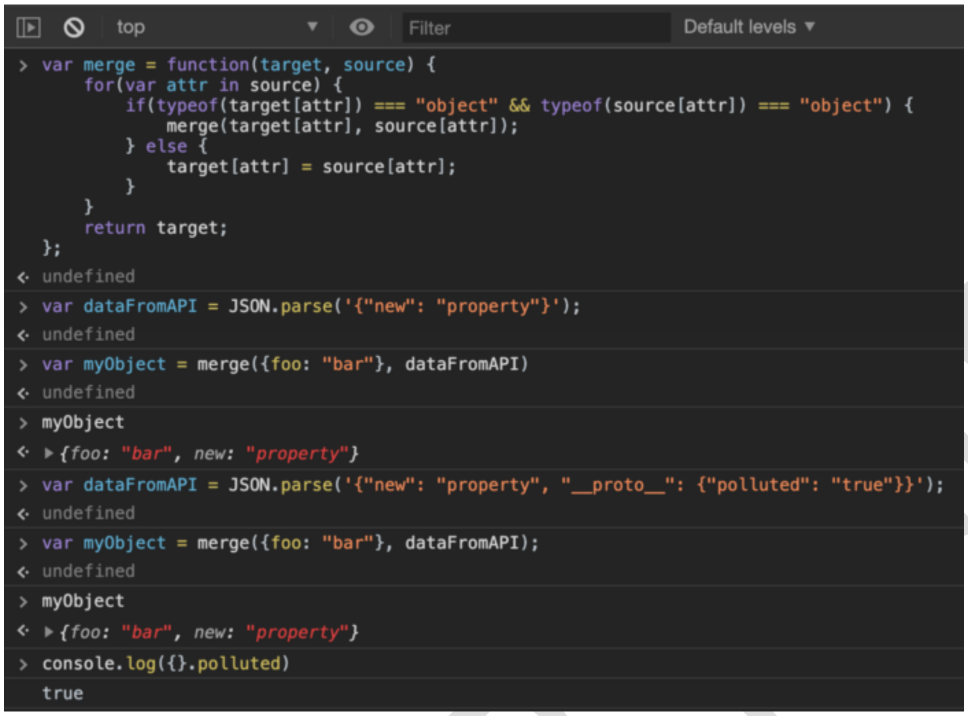


```
> var myObject = {};
<· undefined
> myObject.__proto__.foo = "bar"
<· "bar"
> myObject.foo
<· "bar"
> console.log({}.foo)
  bar
<· undefined
>
```

2. Where it occurs?

prototype pollution and it happens due to some unsafe merge, clone, extend and path assignment operations on JSON objects obtained through user inputs.

Ex1:



```
> var merge = function(target, source) {
      for(var attr in source) {
          if(typeof(target[attr]) === "object" && typeof(source[attr]) === "object") {
              merge(target[attr], source[attr]);
          } else {
              target[attr] = source[attr];
          }
      }
      return target;
  };
<· undefined
> var dataFromAPI = JSON.parse('{"new": "property"}');
<· undefined
> var myObject = merge({foo: "bar"}, dataFromAPI)
<· undefined
> myObject
<· ▸{foo: "bar", new: "property"}
> var dataFromAPI = JSON.parse('{"new": "property", "__proto__": {"polluted": "true"}}');
<· undefined
> var myObject = merge({foo: "bar"}, dataFromAPI);
<· undefined
> myObject
<· ▸{foo: "bar", new: "property"}
> console.log({}.polluted)
  true
```

Ex 2:

https://grey-acoustics.surge.sh/?__proto__%5Bonload%5D=alert(1)

3. Impact

- Vary based on app implementation
- Bypass authentication
- Bypass sanitization
  - https://research.securitum.com/prototype-pollution-and-bypassing-client-side-html-sanitizers
- XSS
- RCE (node.js app)

4. Real case

- Reflected XSS on www.hackerone.com via Wistia embed code

  https://hackerone.com/reports/986386

- Prototype pollution – RCE in Kibana (CVE-2019-7609)

  https://research.securitum.com/prototype-pollution-rce-kibana-cve-2019-7609/

- Ghost CMS - RCE

https://www.youtube.com/watch?v=LUsiFV3dsK8

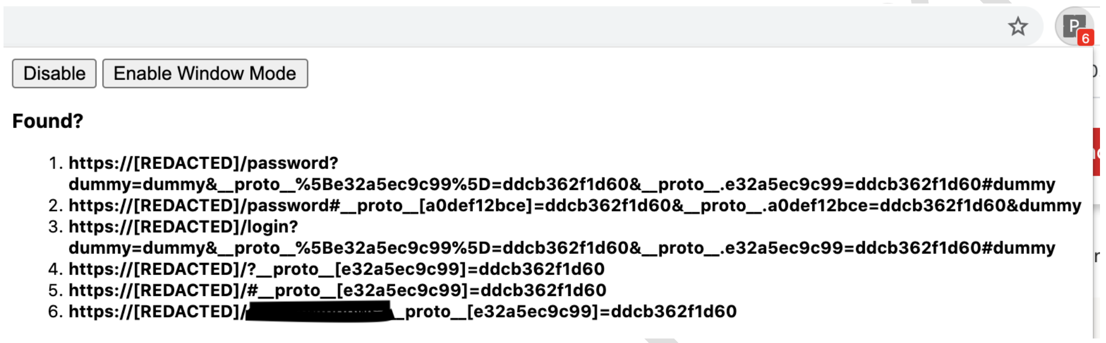
- AST Injection, Prototype Pollution to RCE

  https://blog.p6.is/AST-Injection/

5. How to hunt?

- Extensions

  https://github.com/msrkp/PPScan



Disable | Enable Window Mode

Found?

```
1. https://[REDACTED]/password?
   dummy=dummy&__proto__%5Be32a5ec9c99%5D=ddcb362f1d60&__proto__.e32a5ec9c99=ddcb362f1d60#dummy
2. https://[REDACTED]/password#__proto__[a0def12bce]=ddcb362f1d60&__proto__.a0def12bce=ddcb362f1d60&dummy
3. https://[REDACTED]/login?
   dummy=dummy&__proto__%5Be32a5ec9c99%5D=ddcb362f1d60&__proto__.e32a5ec9c99=ddcb362f1d60#dummy
4. https://[REDACTED]/?__proto__[e32a5ec9c99]=ddcb362f1d60
5. https://[REDACTED]/#__proto__[e32a5ec9c99]=ddcb362f1d60
6. https://[REDACTED]/[redacted]_proto__[e32a5ec9c99]=ddcb362f1d60
```

- Breakpoint on access to a property

  https://www.youtube.com/watch?v=OvOyW4jQNps&feature=youtu.be

  https://gist.github.com/dmethvin/1676346

- Pollute.js - Logs all the properties be polluted in the Chrome DevTools Console.

  https://github.com/securitum/research/tree/master/r2020_prototype-pollution

6. Resources

- Payloads

  https://github.com/BlackFan/client-side-prototype-pollution/

- Lab

  https://github.com/Kirill89/prototype-pollution-explained

IV. Conclusions

- Good programming practices will automatically mitigate prototype pollution attacks.
- Since this attack relies heavily on the data sent from the client side, make sure you sanitize them all and also run the npm-audit periodically to keep track of vulnerabilities in the packages you use. After all, It is better safe than to be sorry.