Proceedings of the 2023 International Conference on Machine Learning and Automation
DOI: 10.54254/2755-2721/43/20230839

Exploitation and prevention of Python prototype chain pollution

Zhang Qingyun

Beijing University of Posts and Telecommunications

zhangqingyun@bupt.edu.cn

Abstract. As more and more enterprises and organizations migrate their business infrastructure to the internet, the significance of software security vulnerabilities becomes increasingly prominent. Python, as a flexible and efficient programming language, plays a crucial role in web application development. Consequently, research on its security is imperative. In recent years, researchers have explored Python prototype chain pollution vulnerabilities. This paper aims to further investigate the exploitation of prototype chain pollution in Python and provide an explanation of pollution prevention. The paper will introduce the working principles of the prototype chain, attack scenarios, and analyze the exploitation of prototype chain pollution in Python using experimental samples. Finally, the article will propose two possible defense measures to assist developers in better mitigating potential development risks and safeguarding cyberspace security.

Keywords: Prototype Chain, Prototype Chain Pollution, Reflection, Dynamic Pollution Analysis, Fuzz Testing.

1. Introduction

Prototype pollution is a dangerous vulnerability that affects prototype-based languages, such as JavaScript and the Node.js platform. It refers to the attacker's ability to inject properties into the root prototype of objects at runtime and subsequently trigger the execution of legitimate code snippets accessing these properties on the object's prototype. This can lead to attacks such as Denial of Service (DoS), privilege escalation, and remote code execution (RCE) [1].

Python is a high-level scripting language with interpretive, compiled, interactive, and object-oriented features. However, like other programming languages, Python is not immune to security risks. One of the recently highlighted risks is prototype chain pollution attacks. Prototype chain pollution attacks were initially discovered in JavaScript, but with Python's widespread use in web development, the threat of such attacks has also become evident.

This paper aims to discuss the further exploitation and prevention of prototype chain pollution in Python. The article will first explain the principles of Python prototype chain pollution attacks, including the use of the merge function and the modification of built-in variables in the prototype chain pollution principle. It will then elaborate on the further exploitation of Python prototype chain pollution, specifically based on global variable pollution. Finally, the article will provide two measures for defending against prototype chain attacks, namely practical measures to check object prototypes before and after object reception and theoretical measures for detecting prototype chain pollution vulnerabilities.

© 2024 The Authors. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (https://creativecommons.org/licenses/by/4.0/).

DOI: 10.54254/2755-2721/43/20230839

The latter combines pollution analysis with fuzz testing. The target audience of this article includes Python developers and security engineers.

2. Materials and Methods

(1) Relevant Concepts and Technologies

a. Python[2]

Python is a high-level scripting language that combines interpretive, compiled, interactive, and object-oriented features. It was created by Guido van Rossum in 1991. Python has an extensive standard library and is used for a wide range of applications, including web development, artificial intelligence, data science, cryptography, machine learning, and more.

b. JavaScript[3]

JavaScript is an interpreted dynamic programming language that supports object-oriented, imperative, and functional programming styles. It was originally designed and implemented by Brendan Eich in 1995. JavaScript has a vast ecosystem, including rich libraries and frameworks, enabling developers to efficiently create various applications. JavaScript is a critical component of modern web development and also excels in backend development, mobile application development, desktop application development, and the Internet of Things (IoT).

c. Prototype

A prototype is an attribute of a function object that defines the common ancestor for objects produced by that function. Objects created by a function can inherit the properties and methods of this prototype, as the prototype itself is an object. Prototypes are used to extract common properties of objects, addressing memory consumption and code redundancy issues, and can be employed to add, remove, query, or modify properties.

d. Prototype Chain

Connecting prototypes into a chain forms a prototype chain, which records the entire process of creating objects using prototypes. The prototype chain serves as a historical record of object creation through prototypes, allowing sequential property lookup when a property is not directly present in an object.

e. Reflection

Reflection is primarily applied to objects of classes and involves actions such as looking up, retrieving, deleting, and adding members (properties or methods) using strings. It is a string-based event-driven technique. Reflection is commonly used for dynamically adding properties and methods to objects, as well as dynamically invoking methods or accessing properties. In software automation testing, reflection can be combined with keyword-driven and data-driven approaches to develop more flexible and maintainable scripts.

f. Dynamic Pollution Analysis

Dynamic pollution analysis involves marking and tracking certain data at runtime. This type of dynamic analysis is becoming increasingly popular and has been successfully used in application security environments to prevent various attacks, including buffer overflows, format string attacks, SQL and command injection, and cross-site scripting [4].

g. Fuzz Testing

Fuzz testing is currently one of the most popular vulnerability discovery techniques. In the 1990s, Barton Miller of the University of Wisconsin-Madison first introduced the concept of fuzz testing. In essence, fuzz testing generates a large volume of normal and abnormal inputs for a target application and attempts to detect anomalies by providing these inputs to the application and monitoring its execution. Compared to other techniques, fuzz testing is easy to deploy, highly scalable, and applicable even without access to source code. Furthermore, conducting fuzz testing in practical scenarios can yield higher precision [5].

DOI: 10.54254/2755-2721/43/20230839

(2) Background Introduction

a. Prototype Chain Pollution Background

Prototype chain pollution was first introduced by Arteau in 2018 [6]. The existence of this vulnerability is due to a feature in JavaScript called the prototype chain. This feature not only allows searching for properties under the current object but also permits the retrieval of properties through a chain of prototype objects. Specifically, prototype pollution enables attackers to inject or modify prototype properties under prototype objects (e.g., object), thereby affecting the normal execution of vulnerable programs (e.g., control flow and data flow).

b. Pollution Attacks

In Python, mutable objects are objects that can be modified after creation, such as lists, dictionaries, and so on. When we use a mutable object as the default parameter of a function, the same default parameter object is used each time the function is called. If we add tainted properties to the default parameter object, those properties will be retained and affect the results of subsequent function calls. Attackers can inject properties into the root prototype of objects at runtime and subsequently trigger the execution of legitimate code snippets accessing these properties on the object's prototype. This can lead to attacks such as Denial of Service (DoS), privilege escalation, and remote code execution (RCE).

(3) Principles of Pollution Attack on Prototypes

Since classes in Python inherit properties from their parent classes, and properties declared within classes (not within instances) are unique, our goal is to target these properties that still refer to a unique attribute in multiple classes and instances, such as built-in properties in classes prefixed with "__".

The most common method for Python prototype chain pollution attacks is to use the merge function. As shown in the following code, it achieves parameter overwriting through recursion and can easily be used to maliciously overwrite existing properties.

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

Furthermore, we can also achieve a similar class attribute assignment logic to the merge function using the "set_" and "set_with" functions within the Pydash module, thereby implementing pollution attacks.

Regarding pollution attack methods, modifying built-in objects is one of the most common techniques in Python prototype chain pollution. Attackers can add malicious properties to built-in objects, such as modifying built-in objects within the global scope. Then, objects containing these malicious properties are passed as default parameters to functions. When the function is called, the interpreter first looks for the object's properties and then searches up the object's prototype chain. When the interpreter discovers the malicious property, it executes the code. Therefore, when using mutable objects, developers are advised to use "copy()" or "deepcopy()" to create copies instead of using them directly in the program.

(4) Attack Scenarios for Prototype Chain Pollution

Web applications typically allow user input, and attackers can pass objects containing malicious properties through user input to taint default parameter objects, then execute arbitrary code using the

DOI: 10.54254/2755-2721/43/20230839

tainted properties. Therefore, when web applications accept user input, input validation and filtering should be used to prevent malicious input.

Furthermore, many programs use third-party dependency packages that may also contain potential security risks. Hackers can create malicious packages containing prototype chain pollution code. When developers use these malicious packages in their applications, attackers can gain control over code execution through pollution of the prototype chain. Some well-known npm packages, such as lodash, request, and bluebird, have also been found to have prototype chain pollution vulnerabilities [7].

(5) Prevention and Mitigation of Prototype Chain Pollution Attacks

Many third-party dependency packages used in programs may contain potential prototype chain pollution risks. However, not all dependency packages receive timely patches. Therefore, this article will provide a practical mitigation solution, namely, recursively checking the object's prototype before and after receiving it. Using this method, all information from parent classes can be obtained, and a comparison of parent class information before and after calling the function can be used to detect whether the prototype chain has been tampered with. By using this temporary mitigation solution, developers can help ensure the security of their programs.

(6) Experimental Samples

Modifying built-in objects to achieve malicious intent is the most common method of exploitation in prototype chain pollution. Additionally, there are techniques for attacking by modifying global variables through prototype chain pollution. In Python, methods are stored as global variables, and any class method can access global variables using Python's built-in "__globals__". This paper will use code implemented using this technique as experimental samples.

In Python, both functions and class methods have a "__globals__" property. This property returns global variables in the variable space declared by the function or class method in the form of a dictionary. This dictionary includes globally defined variables, functions, and class definitions at the module level. In other words, by accessing "b.__init__.__globals__," you can modify methods of class "a," which are unrelated to it, and the value of the global variable "tmp."

```
tmp = 1
class a:
    class_tmp = "the most popular programming languages"
class b:
    def __init__(self):
        pass
```

First, I defined a global variable "tmp" and assigned it a value of 1. I also defined a class "a," which includes a static property "class_tmp" with an initial value of the string "the most popular programming languages." I defined another class "b," which contains an empty "__init__" method.

```
payload = {
    "__init__" : {
            "__globals__" : {
                "tmp" : 2,
                "a" : {
                    "class_tmp" : "Python"
                }
            }
        }
}
```

DOI: 10.54254/2755-2721/43/20230839

Next, I defined a dictionary called “payload” with a special structure for passing data to be merged. It contains a sub-dictionary with a key “__init__,” and this sub-dictionary further contains a key “__globals__.” This structure is used to modify variable values in the global namespace.

```
instance = b()
print(a.class_tmp)#the most popular programming languages
print(tmp)#1
merge(payload, instance)
print(a.class_tmp)#Python
print(tmp)#2
```

Then, I created an instance of class “b” named “instance.” Subsequently, I used “merge(payload, instance)” to merge the data from “payload” into “instance.” During this merging process, the values of the global variable “tmp” and the static property “class_tmp” of class “a” were modified based on the contents of the “__globals__” dictionary in “payload.”

Finally, by using “merge(payload, instance)” again, the data from “payload” was merged into “instance,” thus modifying the static property of class “a” and the global variable “tmp.”

(7) Experimental Principles

This is a function [Appendix 1] that retrieves information from all parent classes, with the parameter being one or more classes to be inspected.

First, an element “__builtins__” is added to the “bases” list. “__builtins__” is Python’s built-in module, containing information about Python’s built-in functions and exceptions. This is done to ensure that there is at least one element in the “bases” list.

Next, the function iterates through each class in the “input_classes” list. In each iteration, the current class “input_class” is added to the “bases” list. A while loop is used to iterate through the parent class chain until there are no more parent classes (i.e., when “tmpbase.__base__” is None). In each while loop iteration, the parent class “tmpbase” is added to the “bases” list, and “tmpbase” is updated to its parent class, i.e., “tmpbase = tmpbase.__base__.” The “bases” list is then converted into a set and back into a list to remove duplicate classes.

An empty dictionary “base_data” is created to store information about classes and their attributes. The function iterates through each class in the “bases” list, storing it in the variable “base.” In each iteration, an empty dictionary “base_dict” is created to store the attributes of the current class “base.” The “dir(base)” function is used to get a list of all attribute and method names of the current class “base,” and this list is iterated through. In each loop, the “getattr(base, item)” reflection function is called to obtain the value of the current attribute or method and store it in the “base_dict” dictionary with the attribute or method name as the key and the value as the value. The “base_dict” dictionary is stored in the “base_data” dictionary with the current class “base” as the key and “base_dict” as the value.

Finally, after iterating through all the classes, the “base_data” dictionary containing information about classes and attributes is returned.

3. Results and Discussion

(1) Experimental Results and Analysis

```
instance = b()
data1=get_proto_data([a])
print(a.class_tmp)#the most popular programming languages
print(tmp)#1
merge(payload, instance)
print(a.class_tmp)#Python
print(tmp)#2
data2=get_proto_data([a])
assert data1==data2#AssertionError
```

DOI: 10.54254/2755-2721/43/20230839

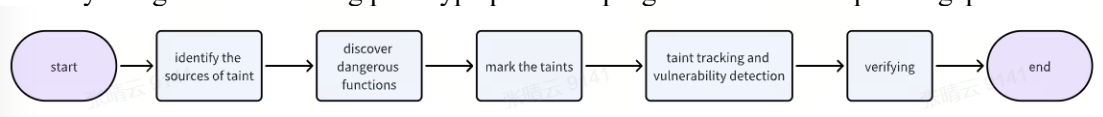
As shown above, the "get_proto_data()" function was inserted into the code that utilizes global variables. A comparison was made between the dictionaries before and after executing the "merge" function to obtain information about all parent classes, which was used to determine whether class "a" had undergone any changes.

Based on the execution results and output of the code, the final assertion "assert data1 == data2" failed, indicating that after merging the data, the properties and methods of class "a" had changed, causing the dictionaries returned by "get_proto_data()" to no longer be identical. This confirms that the method of recursively checking the object's prototype before and after receiving it can effectively detect prototype chain pollution in the code.

The method of recursively checking the object's prototype before and after receiving it can retrieve information about parent classes, allowing a comparison of parent class information before and after the function call to detect whether the prototype chain has been tampered with. However, this method requires security engineers to know the specific location of prototype chain pollution code, making it primarily suitable for vulnerability patching and not widely applicable for vulnerability detection.

(2) Other Ideas for Prevention and Mitigation of Prototype Chain Pollution Attacks

The method of recursively checking the object's prototype before and after receiving it is a patch for prototype chain pollution vulnerabilities and cannot be widely used for vulnerability detection. Currently, there are no open-source tools available in the market that can precisely detect prototype pollution vulnerabilities in Python code. Therefore, the article proposes an alternative approach that combines dynamic pollution analysis with fuzz testing for detecting prototype chain pollution. This approach is specifically designed for scanning prototype pollution programs and can help fill a gap in the field.



First, identify the sources of taint in Python prototype chain pollution vulnerabilities. Heuristic strategies can be used to mark data originating from external inputs as "tainted" data, conservatively assuming that these data may contain malicious attack data. Additionally, places that are user-controllable or indirectly controllable, such as files, cookies, and databases, are also sources of taint [8]. In simple terms, functions responsible for receiving user input parameters are considered taint sources.

The second step involves using fuzz testing to discover dangerous functions. In Python application development, both user-defined functions and external library functions can potentially be affected by prototype chain pollution. For externally called library functions, dangerous functions can be identified through routine collection and analysis. For user-defined functions, a common feature analysis of prototype chain pollution can be used, along with fuzz testing, to confirm if they are dangerous functions.

The specific steps are as follows:

1. Construct fuzzy testing seeds. It's possible to collect a large number of prototype chain pollution payloads based on three-address code or other static analysis methods and add them to a prototype chain pollution seed pool. High-weight seeds can be selected for mutation. However, due to the specific syntax of payloads, traditional random mutation methods can be resource-intensive and rarely lead to successful mutations. Therefore, mutations can be performed based on payload generation rules.

2. Execute the fuzz testing script to determine if a function exhibits prototype chain pollution. With the fuzz testing script, traverse and extract functions from the Python script to be tested, and execute these functions. The function's parameters are provided by the fuzzy testing seed. Subsequently, based on whether the prototype chain receives the parameter values specified by the fuzzy testing seed, confirm if the function exhibits prototype chain pollution.

3. Mark functions exhibiting prototype chain pollution as dangerous functions. Based on the outcome of whether the function exhibits prototype chain pollution after execution, if prototype chain pollution is detected, mark the function as a dangerous function.

The third step involves traversing the list of taint sources and marking request parameters contained in the taint source list as tainted. To track pollution across procedure calls, a stack can be maintained to record pollution information about process parameters. At each process call point, inserted calls to the

DOI: 10.54254/2755-2721/43/20230839

pollution tracking library will push the pollution information for the parameters onto the stack. Subsequently, the pollution tracking calls within the called process use this stack to retrieve the pollution information for statements that use these parameters [9].

The fourth step entails taint tracking and vulnerability detection based on the flow of tainted information. By modifying the program's binary code for storing and propagating taint markers and supplementing with analysis, a corresponding flowchart can be created to analyze the flow of tainted information. Check if tainted information received at the taint source location will be transmitted to dangerous functions. If the functions traversed during the taint transfer process are found in the list of dangerous functions, it indicates the presence of a prototype chain pollution vulnerability [10].

Finally, the detected vulnerabilities can be verified through fuzz testing. For identified vulnerabilities, fuzzy testing seeds can be sent directly at the taint source location, and then the response can be observed at the dangerous function point. Based on whether the function exhibits prototype chain pollution after receiving the parameter values specified by the fuzzy testing seed, confirm if the function is vulnerable.

This method enables automated detection of prototype chain pollution vulnerabilities by using fuzz testing to discover dangerous functions in user-defined code and dynamic taint analysis to trace the flow of tainted information within the program.

4. Conclusion

This article begins by explaining the principles of Python prototype chain pollution attacks, using experimental samples to illustrate further exploitation, specifically focusing on prototype chain pollution based on global variables. Prototype chain pollution in Python, if left unprotected, can pose a certain degree of harm to network security and people's interests. Therefore, the article emphasizes and provides two measures for preventing and mitigating Python prototype chain pollution attacks. One is to check the object prototype before and after receiving it, which has been validated for its effectiveness in the article. The other approach combines dynamic taint analysis with fuzz testing, which, while not yet implemented, has strong theoretical support.

In summary, although prototype chain pollution is a widespread issue, its potential risks can be mitigated through careful code reviews, testing, and vulnerability detection measures. It is hoped that readers, after reading this article, will have a deeper understanding of prototype chain pollution attacks and their defenses.

References

[1] Shcherbakov, Mikhail, Musard Balliu, and Cristian-Alexandru Staicu. "Silent spring: Prototype pollution leads to remote code execution in Node. js." USENIX Security Symposium 2023. 2023.

[2] Python Software Foundation. "Python.org." Python.org, https://www.python.org/. Accessed 9 October 2023.

[3] Mozilla. "JavaScript." Mozilla Developer Network, https://developer.mozilla.org/en-US/docs/Web/JavaScript. Accessed 9 October 2023.

[4] Clause, James, Wanchun Li, and Alessandro Orso. "Dytan: a generic dynamic taint analysis framework." Proceedings of the 2007 international symposium on Software testing and analysis. 2007.

[5] Li, Jun, Bodong Zhao, and Chao Zhang. "Fuzzing: a survey." Cybersecurity 1.1 (2018): 1-13.

[6] Arteau, Olivier. "Prototype pollution attack in nodejs application." (2018).

[7] Ouyang, Ziyi. "Research and Explore of Prototype Pollution Attack in Python." 2023 3rd Asia-Pacific Conference on Communications Technology and Computer Science (ACCTCS). IEEE, 2023.

[8] Tripp, Omer, et al. "TAJ: effective taint analysis of web applications." ACM Sigplan Notices 44.6 (2009): 87-97.

[9] Ganesh, Vijay, Tim Leek, and Martin Rinard. "Taint-based directed whitebox fuzzing." 2009 IEEE 31st International Conference on Software Engineering. IEEE, 2009.

DOI: 10.54254/2755-2721/43/20230839

[10] Schwarz, Michael, Moritz Lipp, and Daniel Gruss. “JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks.” NDSS. Vol. 18. 2018.

Appendix 1:

```
def get_proto_data(inupt_classes):
    bases=[    builtins    ]
    for input  class in inupt  classes:
        tmpbase=input_class
        bases.append(tmpbase)
        while(tmpbase.__base__):
            bases.append(tmpbase)
            tmpbase=tmpbase.    base
    bases=list(set(bases))

    base_data= {}
    for base in bases:
        base  dict={}
        for item in dir(base):
            base_dict[item] = getattr(base, item)
        base_data[base]=base_dict
    return base_data
```