Vulnerability Assessment of PyYAML v5.1.1

By 45820384

12/05/2022

Contents

Demonstration Link: https://youtu.be/BtmjlGRnPHA

1 Executive Summary

This report provides a vulnerability assessment of the PyYAML 5.1.1 python package. PyYAML is a YAML parser and emitter for python. Through the use of appropriate methodology, an analysis of the package was performed. Two core vulnerabilities were identified that put any system utilising the package at risk. These vulnerabilities all take advantage of the instantiation of python objects written in YAML with python tags. By doing this an attacker can perform RCE on a system and destroy all integrity of such a system and its information. To conclude this report, possible mitigations for both users and developers of PyYAML 5.1.1 are identified and advised to be employed as soon as possible.

2 Introduction

I was contracted by UQ-COMP3320 to perform a vulnerability assessment on one of a provided selection of source packages. The task required an in-depth explanation of the methodology carried out during assessment, as well as, demonstrating an understanding of the appropriate frameworks, tools and techniques utilised. Additionally, threat identification and risk assessment of discovered vulnerabilities had to be carried out. Using these assessments, mitigations and remedies were identified and suggested to counteract and/or limit the impact of the discovered vulnerabilities. This report aims to present an analysis of vulnerabilities found and recommend counter measures for reducing or mitigating the risk of compromise through the use of the PyYAML package. Additionally, the report demonstrates a POC for the vulnerability deemed to have the highest potential impact.

2.1 Chosen Package: PyYAML 5.1.1

From the provided list of software packages, PyYAML 5.1.1 was chosen for assessment. PyYAML is a full-feature YAML processing framework for Python containing a complete YAML 1.1 parser (Simonov, 2022). As such, it is applicable for a broad range of tasks from complex configuration files to object serialization and persistence (Simonov, 2022). Additionally, it is currently the most popular python YAML parser and one of the most popular python packages with 135,746,928 downloads in the last month (PyPI, 2022). PyYAML Version 5.1.1 was released Jun 7, 2019 (Simonov, 2022). Of particular interest is the different YAML loader and dumper methods supplied by the python package. This is the primary way PyYAML would interact with user input and, as such, is the primary attack vector an attacker might consider when exploiting the system.

2.2 Objectives and Scope

The purpose of this vulnerability assessment is to identify vulnerabilities in PyYAML 5.1.1 by employing techniques such as static analysis, dynamic analysis and fuzzing against the package and its methods. To complement these tools, manual code analysis and package documentation were utilised such that a greater insight into the behaviour of the Python package could be ascertained. In particular, this assessment aims to evaluate the publicly accessible methods of this python package as these are the methods which are utilised by external scripts to interact with the PyYAML framework. The relevant functions focused on during this assessment are listed within the Methodology section below.

As PyYAML supports both Python 2 and Python 3, in order to achieve full coverage both implementations were examined. Additionally, PyYAML supports bindings to the LibYAML library, which allow the use of LibYAML parsers in PyYAML. This functionality is deemed

outside the scope of this assessment as it requires the installation of additional software in the deployment (LibYAML) in order to be utilised. Secondary functionalities, such as the utilisation of user written constructors, were not evaluated as they would not typically be parsed user input but would affect the behaviour of the other functions of the module. As such a possible extension to the procedures presented here would be to analyse the interaction of such methods on the functions which typically parse user input.

The PyYAML 5.1.1 package was sourced using the latest pip module (v22.0.4). The source code of the package was obtained from:

https://pypi.org/project/PyYAML/5.1.1/

The latest versions of Python 2 and Python 3 were utilised in the testing and working environments. At the time of environment construction these versions were Python 2.7.18 and Python 3.10.4. The source code and installers for these versions can be acquired from here:

https://www.python.org/downloads/

3 Methodology

For this vulnerability assessment, a rigorous methodology was employed based on a few of the leading industry standards. In particular, the following methodology aims to follow the guiding principles as laid out in the National Institute of Standard and Technology (NIST) special publication 800-115.

1. Creation of well documented, (software) isolated and reproducible testing environments
2. Enumeration and evaluation of system assets
3. Modelling of threats to these assets
4. Vulnerability detection using a range of appropriate techniques
5. Risk assessment of discovered vulnerabilities
6. Confirmation and identification of discovered vulnerabilities and their causes
7. Establish mitigation recommendations

Asset identification was performed in accordance with the process approach outlined by ISO 27001. Threat identification and modelling were evaluated using the STRIDE model. Static and dynamic analysis was performed on the python source code of the PyYAML package. Static analysis was done through the use of automated python analysis tools such as Prospector, its sub tools and Semgrep. Dynamic analysis was done through the use of coverage-guided mutation-based fuzzer, Atheris. Additionally, to increase the effectiveness of Atheris, structure aware fuzzing was implemented. This provided a benefit as YAML is a structured language and non-compliant YAML documents would produce frequent and uninteresting errors. Generation-based fuzzing was also considered. However, it was ultimately decided that the resources needed to setup a python generation based fuzzer with a proper configuration outweighed any potential benefit. Ultimately, tools were chosen based on their compatibility with Python and their relevance to the analysis required to efficiently evaluate the target package.

3.1 Environment Setup

Both the testing environment and vulnerable target environment were hosted on a Kali Linux 2022.1 virtual machine with default configurations virtualized through Oracle VM VirtualBox. The latest KALI image can be retrieved from here:

https://www.kali.org/get-kali/#kali-virtual-machines

The relevant python versions were installed on the machine and PyYAML 5.1.1 was installed to the machine via the pip module. The following environments were designed around achieving a clearer understanding on the behaviour of the PyYAML package by establishing a controlled environment with minimal interactions with external functionality.

3.1.1 Testing Environment

Utilising the docker framework, a containerised environment was created through the command:

```
docker build -f testenvDocker -t pyyaml-test:5.1.1 .
docker volume create pyyaml-test-env
```

The testenvDocker docker file used to create the environment can be found at Appendix 6.1. During testing the docker was then run with the command:

```
docker run -it -v pyyaml-test-env:/env pyyaml-test:5.1.1
```

3.1.2 Exploitable Environment

In order to demonstrate the main RCE exploitation found, a simple PyYAML parsing python server was written and executed within a docker container. The vulnerable python script 'target.py' can be found at Appendix 6.4. The following steps outline the process of setting up the docker container and running the python server.

Utilising the docker framework, a containerised environment to represent a remote target system utilising the vulnerable package was created through the command:

```
docker build -f targetDocker -t pyyaml-serv:5.1.1 .
```

The targetDocker docker file used to create the environment can be found at Appendix 6.2. During testing the environment was then initialised with the command:

```
docker run -p 7000:7000 pyyaml-serv:5.1.1
```

3.2 Asset Identification

Identification of highly sensitive assets is an important first step in any vulnerability assessment. The knowledge of these assets forms the basis of approximating potential losses should a vulnerability be found within the software package. It can also help to focus the assessment to key targets which an attacker might attempt to disrupt. As PyYAML is run within a Python environment there are a few key assets which could be exploited to invalidate the confidentiality, integrity, availability, authentication, authorization and/or the accountability of the information managed by the underlying system. The highly sensitive assets of the constructed environment (python server within a container hosted on a Kali VM) are:

- Python Memory \ Memory of Base System
- CPU capacity/resources
- Logs maintained by the system
- Files managed by the system (sensitive files such as /etc/passwords)

- System Kernel
- Functionality of python scripts utilising the package

It is noted that deployments of the same PyYAML package on different systems may have different and more sensitive resources. This is due to PyYAMLs nature as a python (cross platform) data serialisation package. As such, the assets identified above represent an array of assets which would be critical for almost any environment. Ultimately, this assessment is significant limited in identifying all possible potential threats as this would be a massive undertaking. Despite this, it is hoped that the core assets of as many potential systems using PyYAML are encapsulated within the evaluation. It is noted that the methodology employed during this assessment is still limited and cannot encase all possible deployment environments.

3.3 Threat Modelling

During the course of this vulnerability assessment, threat modelling was used to guide the analysis of source and detection of vulnerabilities by identify potential attack vectors through the functionality of the PyYAML module. By identifying trust boundaries and the entry points of user supplied data, the subsequent vulnerability detection and risk assessment can be focused on areas of potential weakness. The following categorized threat list was generated utilising the STRIDE model.

STRIDE Threat	Potential Vulnerabilities	Vulnerable Asset	Notes
Spoofing	-	-	-
Tampering	Arbitrary Python Code Execution	All assets	Attackers provide input which results in the arbitrary execution of python code.
	Memory Tampering	Python Memory	Attacker makes changes to the Python environment through provided input.
Repudiation	-	-	-
Information Disclosure	Reading arbitrary files	Files managed by base system	Attackers provide input which results in the display of the content of an arbitrary file.
Denial of Service	Input Overload	Memory, Functionality of Python Scripts	Attackers can flood the input of the process resulting in an overconsumption of resources.
Elevation of Privilege	Unexpected input / Logic Flaw	All assets excluding functionality of python scripts	Attackers provide input which results in the execution of unintended functionality and an escalation of privileges.

Table 1: STRIDE Threat Model

3.4 Vulnerability Detection

In order to efficiently detect vulnerabilities, static analysis was performed on the source code of the PyYAML package first. This informed areas of interest and guided the dynamic analysis process. Finally, online research through the CVE and CWE databases was used to identify discovered vulnerabilities and weaknesses.

3.4.1 Static Analysis

Firstly, Prospector was setup and envoked within the PyYAML source code distribution using the following commands:

- pip install prospector[with_mypy,with_bandit,with_vulture,with_pyroma]
- python -m prospector -w bandit -w vulture -w pyroma -W pylint -o text:output.txt -8

A large amount of python syntax errors were uncovered within the package. Notable discoveries include:

- bandit - B506: Use of unsafe yaml load. (Allows instantiation of arbitrary objects)
- bandit - B101: Use of assert. (Asserts can be disabled leading to unexpected behaviour)
- Parser.parse_node: mccabe - MC0001: Overly complex methods
- vulture - unused-variable: Unused variable 'dummy'
- pyflakes - F401: 'sys' imported but unused (initialises unused library)

Additionally, Semgrep was setup and invoked within the PyYAML source code distribution using the following commands:

- pip install semgrep
- python -m semgrep –config auto

The output from Semgrep supported the B506 alert found by above, as well as, discovering a possible Runtime Error. This runtime error could be caused by a dictionary changing size during an iteration within the scanner functionality.

3.4.2 Dynamic Analysis

Finally, the Atheris fuzzer was employed through the python script atheris_pyyaml.py in the Test Environment. Due to the discoveries above, the first method to be fuzzed was yaml.load. After a method had been sufficiently fuzzed the script would be altered to test a new public method of the target package.

For each one of the tested public methods in PyYAML the following process was followed:

Firstly, the expected default operation of the examined method was modelled using a python script and the coverage python module.

- python3 -m coverage run -L normal_op.py
  i. An example normal_op.py for yaml.load has been included at Appendix 6.5
- python3 -m coverage report

Then fuzzing was performed on the selected function utilising the command:

- python3 -m coverage run -L atheris_pyyaml.py -atheris_runs=1000

At each crash, the raised exception and the corresponding input was recorded. The mutator and testing functions were then altered to ignore the newly discovered exception. An example atheris_pyyaml.py which handles the yaml.load method can be found at Appendix 6.3.

After a normal exit of the testing script is finally reached, a comparison of the normal coverage report and the fuzzed coverage report is made in order to identify any abnormal behaviour in the python environment.

Finally, each discovered crash input is then manually tested with the other public methods of PyYAML, as well as their respective Python 2 counterparts. This avoids rediscovering the same errors shared by methods when running the Atheris fuzzer on the other methods. The process is then repeated for the remaining untested public methods of PyYAML.

3.5 Risk Assessment

For risk assessment of the discovered vulnerabilities, the CVSS 3.1 framework was utilised. This particular framework was chosen due to its wide industry adoption and since it provides a clear reference point for comparing the severity of such vulnerabilities with those in other frameworks.

4 Findings

Static analysis directly led to the discovery of an exploit which was identified as CVE-2020-1747. Through dynamic analysis, online research and the examination of similar functionality within the package, one additional vulnerability was uncovered. This vulnerability was a similar CWE 20, which could be exploited through the unsafe_load method. POCs for these two vulnerabilities were sourced and were executed to manually confirm the exploitability of the vulnerabilities discovered. Additionally, bad inputs were discovered during dynamic analysis which were confirmed to reproducibly raise errors when processed through their relevant functions. The possible runtime error discovered in the scanner.py file could not be reliably reproduced through public methods.

4.1 Unexpected Input Errors (UIE)

This vulnerability forms from a niche set of possible inputs which result in crashes of applications utilising PyYAML functions. In particular, these vulnerabilities involve passing input which is valid YAML but generates unexpected behaviour within the parser, resulting in a crash. This vulnerability is a result of improper input validation (CWE-20) as well as unexpected behaviour for valid YAML inputs. One of these inputs and the resulting error are demonstrated below.

4.1.1 Proof of Concept (UIE)

Command:

```
yaml.safe_load("._")
```

Output:

```
ValueError: could not convert string to float: '.'
```

This exploit is performed by supplying input (either from a remote or local context) which results in an uncaught and unexpected exception. It should be noted that this vulnerability exists within the other loader methods as well. Additionally, invalid YAML (improper formatting, special characters without quotes, etc) passed to the same methods will also produce exceptions defined by the PyYAML package. If the application utilising PyYAML does not correctly handle these exceptions this can lead to a crash of the application. An application could easily be restarted but the exploit could easily be repeated leading to another crash. As such an attacker could cause a denial of service for any application utilising the PyYAML parser for user

supplied input. The exception demonstrated above is a result of incorrect PyYAML behaviour where it attempts to convert "." to a python float literal.

4.2 Proof of Concept and Analysis (CVE-2020-1747 & CWE-20 in unsafe_load)

The following payload was retrieved from https://github.com/yaml/pyyaml/issues/420:

```
!!python/object/new:tuple
- !!python/object/new:map
  - !!python/name:eval
  - [ __import__("os").system("echo RCE ACHIEVED!") ]
```

Figure 1: payload.yaml

The payload above has been slightly modified, in particular the random code that is executed by the payload has been changed to be more evident when exploited on the target environment. These exploits are performed by supplying the above payload to either yaml.load/yaml.full_load (CVE-2020-1747) or yaml.unsafe_load (CWE-20 in unsafe_load). In particular, these exploits take advantage of arbitrary python object instantiation allowed by the unprotected methods of the PyYAML package. As such, the payload above essentially causes the instantiation of the following python code during deserialization:

- tuple(map(eval, ['(__import__("os").system("echo RCE ACHIEVED!")']))

```
!!python/object/new:tuple
```

This creates a new tuple object by calling object.__new__(tuple, *args) and the generated map is passed in *args. The primary purpose of this tuple is to force the python environment to evaluate all elements of the map iterator resulting in the actual execution of the map function against the supplied arguments and by proxy executes our supplied random code.

```
!!python/object/new:map
```

This creates a new map object by calling object.__new__(map, *args). Two args are passed through *args to this constructor, the function of a built in python function (as full_load and load do not allow the use of non-imported modules) and a list of different arguments that an attacker wants passed to this function.

```
- !!python/name:eval
- [ print(__import__("os").system("echo RCE ACHIEVED!")) ]
```

The function supplied to the map is the built in function eval which is instantiated through the use of a complex python tag. The eval function parses a string and if the string is valid python code, eval executes it. As such, an attacker then passes a list containing any arbitrary python code they want executed. In this example, the os module is imported (as the target application may not already have it imported) and a system call is made. As this module is a part of the python standard library this payload will work against all python environments utilising this vulnerable function. The setup of a vulnerable environment has been outline in Section 3.1.2 in which this exploit can be performed.

It should be noted that unsafe_load allows the instantiation and calling of python objects outside of the current scope of the python environment (calling functions of modules which have not been imported into the environment but exist within the standard library). As such the ways of exploiting CVE-2020-1747 is a strict superset of the ways of exploiting CWE-20 in

unsafe_load. Additionally, the payload is easily generatable through the use of PyYAMLs own dump method and the construction of an appropriate Python class __reduce__ method. As seen above, an attacker can perform arbitrary python code execution and through the os.system function can perform arbitrary code execution on the underlying system.

4.4 Risk Assessment

Vulnerability	CVSS 3.1 Rating	STRIDE Threat/Nullified Property	Description
CVE-2020-1747 **CWE-20 in unsafe_load**	9.8 Critical	Tampering/Complete loss of all desired properties	RCE utilising arbitrary python object instantiation
CWE-20 - UIE	Calculated 5.3 Medium	Denial of Service/Availability	Passing invalid YAML documents to load functions leading to crash.

Table 2 - Risk Assessment of Vulnerabilities

Utilising the STRIDE model, it is observed that CVE-2020-1747 and the CWE-20 in unsafe_load threatens the entire system and all its desired properties. As such the risk associated with these two vulnerabilities is extremely high as it can be performed remotely through user supplied input. These vulnerabilities allow an attacker access to the system at which point further exploitations might be performed in order to further compromise the system (for example elevation of privilege). The complete loss of system integrity combined with the ease of exploiting these vulnerabilities from a remote context results in an extremely high calculated CVSS 3.1 rating. Such extremes highlight the long reaching deep impact this vulnerability has on the host machine and any information it manages.

In comparison, the CWE-20 UIE located throughout the package threatens the availability of any python script utilising the affected methods of the PyYAML package. Denial of service can be achieved through this vulnerability as PyYAML raises exceptions for parsing incorrect syntactical data to its methods. As such, an attacker can crash an unprotected program which utilises PyYAML by passing invalid YAML data to yaml.load or a similar function. As this exploit only has a low impact on the availability of the python application utilising PyYAML, the CVSS 3.1 score is significantly lower than the one generated for the other vulnerabilities. However, the CVSS score is still considerably high due to the ease in which an exploit leveraging this vulnerability can be performed. Since this impact is limited to the python environment and the application can be easily reinstated, this vulnerability should be considered as low risk to the overall environment.

As such in its current state, this product when utilised on any system, places both the overall system, any information it controls and any connected systems at extreme risk.

4.5 Mitigations & Remediations

4.5.1 CWE-20 – UIE

Mitigations suggested for users:

- Use of try catch blocks to handle known and expected exceptions when using PyYAML methods.
- Implement appropriate input validation (valid YAML syntax) before passing arbitrary input to PyYAML methods.

Remediations suggested for developers:
- Implement clear indication in documentation of expected input syntax of PyYAML methods.
- Address unexpected errors from novel inputs (handling of float interpreters for "._").

4.5.2 CVE-2020-1747

Mitigations suggested for users:
- Update to the latest version of PyYAML (does not resolve all issues)
    - Python 2 users should note that recent versions of PyYAML do not support Python 2. As such they should update to the latest possible version of PyYAML.
- Use the safe_load method when parsing data using PyYAML from an untrusted source.
- Implement appropriate input validation (potentially dangerous python object instantiation) before passing arbitrary input to PyYAML methods.

Remediations suggested for developers:
- Removing/fixing the full_load and full_loader functionality, as it is just as vulnerable as the unsafe_loader functionality it was implemented to address.
- Future versions should also break backwards compatibility in order to follow the principles of secure defaults. To do this the default load method should utilise the uncompromised SafeLoader and its methods.
    - This measure is to mitigate the impact of application developers utilising the defaults of the PyYAML package without comprehending exactly how the method behaves. Whilst this fix would break any previous applications which utilised earlier versions of the package, it is a necessary step in ensuring the security of applications utilising PyYAML and their host systems.

4.5.3 CWE-20 in unsafe_load

Mitigations suggested for users:
- Use the safe_load method when parsing data using PyYAML from an untrusted source.
- Implement appropriate input validation (potentially dangerous python object instantiation) before passing arbitrary input to PyYAML methods.

Remediations suggested for developers:
- Package documentation should be updated to clearly outline the potential dangers of utilising the unsafe_load method.

5 Summary

Two important vulnerabilities of a similar type were discovered in this assessment. These discovered vulnerabilities can be performed with the same payload, but they differ in the targeted vulnerable function. Through these two vulnerabilities, an attacker can perform remote code execution gaining significant control over the system. As such, both vulnerabilities put any host system at extremely high risk and should be mitigated through the identified approaches as soon as possible. Additionally, another easily mitigated vulnerability of medium risk was discovered and should be handled immediately by users of PyYAML. Ultimately, the primary concern raised during this assessment is the discovered vulnerability within the default functions of the PyYAML package. Despite the inclusion of safe alternatives, a vulnerability in the default methods means there is likely a significant amount of applications utilising vulnerable functionality without understanding the risks they are adopting. As such the mitigations suggested by this report should be addressed with haste, both by users of the PyYAML and its developers.

6 Appendix

6.1 testenvDocker

```
FROM python:3

RUN apt-get update -y
RUN pip install pip --upgrade pip
RUN pip install --no-cache-dir coverage
RUN pip install --no-cache-dir atheris
RUN pip install --no-cache-dir PyYAML==5.1.1 --global-option=--without-
libyaml
RUN apt-get install -y python2

ENV PYTHONPATH="/python"

WORKDIR /env
COPY atheris_pyyaml.py /env/

CMD /bin/sh
```

Figure 2: testenvDocker

6.2 targetDocker

```
FROM python:3

RUN apt-get update
RUN pip install pip --upgrade pip
RUN pip install --no-cache-dir PyYAML==5.1.1 --global-option=--without-
libyaml
ENV PYTHONPATH="/python"

EXPOSE 7000
WORKDIR /env
COPY target.py /env/

CMD [ "python3", "/env/target.py" ]
```

Figure 3:targetDocker

6.3 atheris_pyyaml.py

```
import atheris

with atheris.instrument_imports():
  import yaml
  import sys

default_yaml = b"{sound: -1989977196.9803586,\
 many: put, original: [[-763676064.9633636, ourselves, \
 true, wealth, 777579141, false], graph, true, false, \
 -99998227, 1050436243], brush: explain, motor: greater,\
 studied: false}"

def YAMLMutator(data, max_size, seed):
    try:
```

```
        yaml.safe_load(data)
    except yaml.YAMLError:
        data = default_yaml
    except yaml.ReaderError:
        data = default_yaml
    else:
        data = atheris.Mutate(data, len(data))
    return data

@atheris.instrument_func
def test_def_load(data):
    try:
        yaml.load(data)
    except yaml.YAMLError:
        pass
    except yaml.ReaderError:
        pass
    except Exception:
        input_type = str(type(data))
        codepoints = [hex(x) for x in data]
        sys.stderr.write("Input was {input_type}: {data}\nCodepoints:
{codepoints}".format(input_type=input_type, data=data,
codepoints=codepoints))
        raise

@atheris.instrument_func
def test_full_load(data):
    try:
        yaml.full_load(data)
    except yaml.YAMLError:
        pass
    except yaml.ReaderError:
        pass
    except Exception:
        input_type = str(type(data))
        codepoints = [hex(x) for x in data]
        sys.stderr.write("Input was {input_type}: {data}\nCodepoints:
{codepoints}".format(input_type=input_type, data=data,
codepoints=codepoints))
        raise

@atheris.instrument_func
def test_safe_load(data):
    try:
        yaml.safe_load(data)
    except yaml.YAMLError:
        pass
    except yaml.ReaderError:
        pass
    except Exception:
        input_type = str(type(data))
        codepoints = [hex(x) for x in data]
        sys.stderr.write("Input was {input_type}: {data}\nCodepoints:
{codepoints}".format(input_type=input_type, data=data,
codepoints=codepoints))
        raise
```

```
@atheris.instrument_func
def test_unsafe_load(data):
    try:
        yaml.unsafe_load(data)
    except yaml.YAMLError:
        pass
    except yaml.ReaderError:
        pass
    except Exception:
        input_type = str(type(data))
        codepoints = [hex(x) for x in data]
        sys.stderr.write("Input was {input_type}: {data}\nCodepoints:
{codepoints}".format(input_type=input_type, data=data,
codepoints=codepoints))
        raise

@atheris.instrument_func
def test_def_load_all(data):
    try:
        yaml.load_all(data)
    except yaml.YAMLError:
        pass
    except yaml.ReaderError:
        pass
    except Exception:
        input_type = str(type(data))
        codepoints = [hex(x) for x in data]
        sys.stderr.write("Input was {input_type}: {data}\nCodepoints:
{codepoints}".format(input_type=input_type, data=data,
codepoints=codepoints))
        raise

@atheris.instrument_func
def test_full_load_all(data):
    try:
        yaml.full_load_all(data)
    except yaml.YAMLError:
        pass
    except yaml.ReaderError:
        pass
    except Exception:
        input_type = str(type(data))
        codepoints = [hex(x) for x in data]
        sys.stderr.write("Input was {input_type}: {data}\nCodepoints:
{codepoints}".format(input_type=input_type, data=data,
codepoints=codepoints))
        raise

@atheris.instrument_func
def test_safe_load_all(data):
    try:
        yaml.safe_load_all(data)
    except yaml.YAMLError:
        pass
    except yaml.ReaderError:
        pass
    except Exception:
```

```
        input_type = str(type(data))
        codepoints = [hex(x) for x in data]
        sys.stderr.write("Input was {input_type}: {data}\nCodepoints:
{codepoints}".format(input_type=input_type, data=data,
codepoints=codepoints))
        raise

@atheris.instrument_func
def test_unsafe_load_all(data):
    try:
        yaml.unsafe_load_all(data)
    except yaml.YAMLError:
        pass
    except yaml.ReaderError:
        pass
    except Exception:
        input_type = str(type(data))
        codepoints = [hex(x) for x in data]
        sys.stderr.write("Input was {input_type}: {data}\nCodepoints:
{codepoints}".format(input_type=input_type, data=data,
codepoints=codepoints))
        raise

@atheris.instrument_func
def test_dump(data):
    try:
        yaml.dump(data)
    except yaml.YAMLError:
        pass
    except Exception:
        input_type = str(type(data))
        codepoints = [hex(x) for x in data]
        sys.stderr.write("Input was {input_type}: {data}\nCodepoints:
{codepoints}".format(input_type=input_type, data=data,
codepoints=codepoints))
        raise

@atheris.instrument_func
def test_safe_dump(data):
    try:
        yaml.safe_dump(data)
    except yaml.YAMLError:
        pass
    except Exception:
        input_type = str(type(data))
        codepoints = [hex(x) for x in data]
        sys.stderr.write("Input was {input_type}: {data}\nCodepoints:
{codepoints}".format(input_type=input_type, data=data,
codepoints=codepoints))
        raise

@atheris.instrument_func
def test_scan(data):
    try:
        yaml.scan(data)
    except yaml.YAMLError:
        pass
```

```
    except Exception:
        input_type = str(type(data))
        codepoints = [hex(x) for x in data]
        sys.stderr.write("Input was {input_type}: {data}\nCodepoints:
{codepoints}".format(input_type=input_type, data=data,
codepoints=codepoints))
        raise

@atheris.instrument_func
def test_parse_emit(data):
    try:
        mod_data = yaml.parse(data)
        yaml.emit(mod_data)
    except yaml.YAMLError:
        pass
    except Exception:
        input_type = str(type(data))
        codepoints = [hex(x) for x in data]
        sys.stderr.write("Input was {input_type}: {data}\nCodepoints:
{codepoints}".format(input_type=input_type, data=data,
codepoints=codepoints))
        raise

@atheris.instrument_func
def test_compose_serialise(data):
    try:
        mod_data = yaml.compose(data)
        yaml.serialize(mod_data)
    except yaml.YAMLError:
        pass
    except Exception:
        input_type = str(type(data))
        codepoints = [hex(x) for x in data]
        sys.stderr.write("Input was {input_type}: {data}\nCodepoints:
{codepoints}".format(input_type=input_type, data=data,
codepoints=codepoints))
        raise

def main():
    atheris.Setup(sys.argv, test_def_load, custom_mutator=YAMLMutator)
    atheris.Fuzz()

if __name__ == "__main__":
    main()
```

Figure 4:atheris_pyyaml.py

6.4 target.py

```
import yaml
import socket

if __name__ == "__main__":
    serv = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    serv.bind(('0.0.0.0', 7000))
    serv.listen(1)
    while True:
```

```
        conn, addr = serv.accept()
        user_input = b''
        data = conn.recv(12000)
        while data:
            user_input = user_input + data
            data = conn.recv(12000)
           yaml_string = user_input.decode()
        output = yaml.load(yaml_string)
```

Figure 5:target.py

6.5 normal_op.py

```
import yaml

def main():
    # Normal Random YAML input
    data = b"{sound: -1989977196.9803586, many: put, original: [[-
763676064.9633636, ourselves, true, wealth, 777579141, false], graph, true,
false, -99998227, 1050436243], brush: explain, motor: greater, studied:
false}"
    # Call the testing method (changed for each iteration)
    yaml.load(data)

if __name__ == "__main__":
    main()
```

Figure 6:normal_op.py

References

PyPI. (2022, May 9). *Most downloaded PyPI packages*. Retrieved from PyPI Stats:

https://pypistats.org/top

Simonov, K. (2022, May 9). *PyYAML 5.1.1*. Retrieved from PyPi:

https://pypi.org/project/PyYAML/5.1.1/