Python Deserialization Denial of Services Attacks and Their Mitigations

Kousei Tanaka and Taiichi Saito

Abstract In recent years, many vulnerabilities in deserialization mechanisms are reported. Serialization is converting an object to a byte string, and deserialization is converting the byte string to the object. Pickle is a serialization/deserialization module in Python standard library. In the pickle module, specially-crafted data consumes huge memory in deserialization. There is a possibility that the memory consumption leads to deniable of services attacks. This paper precisely describes the DoS attacks and their mitigations.

1 Introduction

Pickle is a serialization/deserialization module in Python standard library. In serialization (*pickling*), the pickle module converts a Python object into a byte string, and in deserialization (*unpickling*), it converts the byte string into the Python object.

This paper presents *denial of service* (DoS) attacks in deserialization in Python. While being unpickled, specially-crafted pickled data causes high memory consumption in the pickle module. This paper uses Python 3.5.2.

2 Related Researches

2.1 Java Deserialization Vulnerabilities

In recent years, many vulnerabilities for remote code execution in Java deserialization were reported. For example, Apache Commons Collections 3.2.1.4.0 have

K. Tanaka (✉) · T. Saito
Tokyo Denki University, Graduate School of Engineering, 5 Senju-Asahi-Cho, Tokyo, Adachi-Ku 120-8551, Japan
e-mail: 17kmc05@ms.dendai.ac.jp

T. Saito
e-mail: taiichi@c.dendai.ac.jp

R. Lee (ed.), *Computational Science/Intelligence & Applied Informatics*, Studies in Computational Intelligence 787, https://doi.org/10.1007/978-3-319-96806-3_2

remote code execution vulnerability in object deserialization [1]. On the other hand, some vulnerabilities for DoS attacks in Java deserialization were reported. Tomáš Polešovský presented a DoS attack, OIS-DoS [2, 3]. In OIS-DoS, specially-crafted serialized data is converted into a large array in deserialization, which consumes huge heap memory in Java Virtual Machine.

2.2 Python Deserialization Vulnerabilities

Marco Slaviero [4] showed a remote code execution vulnerability in unpickling. The pickle module does not verify pickled data before unpickling. When an object is deserialized in unpickling, any code in the constructor of the object is executed. On other hand, any DoS attacks in unpickling have not been reported to our best knowledge.

3 Background

This section describes the pickle module in Python standard library.

3.1 Pickle Module Warning

It has been known that Pickle library has remote code execution vulnerabilities. They are not fixed also in Python 3.6.4, and a document for Python 3.5.5 [5] warns as follows.

3.2 Pickle Protocols

The pickle module uses unique protocols called "pickle protocols", for pickling/unpickling. Table 1 shows the minimum Python version required each protocol. Higher-

Table 1 Pickle protocols

Pickle protocol	Overview	Required Python versions
0	Printable ASCII	Python 1 or Later
1	Binary format	Python 1 or Later
2	Efficient pickling	Python 2.3 or Later
3	Support bytes object	Python 3 or Later
4	Support large object	Python 3.4 or Later

type protocol requires newer Python version. Pickle protocol 0 is supported by all the versions of Python and generates pickled data that consists of printable ASCII characters. The latest version (Python 3.6.4) supports the five types of pickle protocols.

3.3 Pickle Virtual Machine's Overview

Pickled data is a program on a virtual pickle machine(PM, but more accurately called unpickling machine). It is a sequence of opcodes to be interpreted by PM and to build an arbitrarily complex Python object. It does not include looping, testing, or conditional instructions, or arithmetic or function calls. Opcodes are executed once each, from first to last, until a `STOP` opcode is reached. PM has two data areas, *the stack* and *the memo*.

Many opcodes push Python objects onto the stack; e.g., INT opcode pushes a Python integer object on the stack, whose value is gotten from a decimal string literal immediately following INT opcode in the pickle bytestream. Other opcodes take Python objects off the stack. The result of unpickling is whatever object is left on the top of the stack when `STOP` opcode is executed.

The memo is simply an array of objects, which is implemented as a `dict` object mapping little integers to objects. The memo serves as "long term memory in PM", and the little integers indexing the memo are akin to variable names. Some opcodes pop a stack object into the memo at a given index, and others push a memo object at a given index onto the stack [6].

3.4 Process of Unpickling

This section describes process of unpickling, by showing the result of unpickling pickled data "`}.`" as an example. When the pickled data "`}.`" is unpickled, `load` method, `load_empty_dictionary` method and `load_stop` method are used, which are included in the pickle module [7]. The following is the source code of `load` method. The `load` method repeats the procedure of reading an opcode from a pickled data file and calling the method corresponding to the opcode.

Source Code 1 load method

```
def load(self):
 """Read a pickled object representation
 from the open file.

 Return the reconstituted object hierarchy
 specified in the file.
 """
 # Check whether Unpickler was initialized
 # correctly. This is
 # only needed to mimic the behavior of
 # _pickle.Unpickler.dump().
 if not hasattr(self, "_file_read"):
   raise UnpicklingError(
```

```
    "Unpickler.__init__() was not called by "
    "%s.__init__()" % (self.__class__.__name__,))
self._unframer = _Unframer(self._file_read,
                           self._file_readline)
self.read = self._unframer.read
self.readline = self._unframer.readline
self.metastack = []
self.stack = []
self.append = self.stack.append
self.proto = 0
read = self.read
dispatch = self.dispatch
try:
  while True:
    key = read(1)
    if not key:
      raise EOFError
    assert isinstance(key, bytes_types)
    dispatch[key[0]](self)
except _Stop as stopinst:
  return stopinst.value
```

The `load` method reads an opcode from the pickled data file and sets it to `key`. Then it calls the method corresponding to `key`. In this example, the `load` method first reads `}` opcode and calls the following `load_empty_dictionary` method.

Source Code 2 load_empty_dictionary method

```
def load_empty_dictionary(self):
  self.append({})
dispatch[EMPTY_DICT[0]] = load_empty_dictionary
```

The `load_empty_dictionary` method pushes an `empty dictionary` onto the stack. Next, The `load` method reads '`.`' opcode and calls the following `load_stop` method.

Source Code 3 load_stop method

```
def load_stop(self):
  value = self.stack.pop()
  raise _Stop(value)
dispatch[STOP[0]] = load_stop
```

The `load_stop` method pops an object on the stack top, and the `load` method returns the object and finishes unpickling. In this example, the `load` method returns an `empty dictionary` object "{}".

4 DoS Attacks Using Unpickling

This section presents memory consumption DoS attacks that make PM deserialize specially-crafted pickled data. The pickled data forms a sequence of opcodes that push Python objects onto the stack in PM. During unpickling, since the pickled data continues to push Python objects onto the stack, the stack continues to grow deeper until `STOP` opcode, and finally consumes huge memory.

The opcode that pushes an `empty dictionary` onto the stack produces quite efficient attacks. The opcode that pushes an `empty dictionary` is one byte.

On other hand, the `empty dictionary` pushed onto the stack consumes 224 bytes memory. Pushed objects continue remaining on the stack until PM reads `STOP` opcode.

4.1 A DoS Attack

This subsection describes the details of a DoS attack using unpickling. This following code creates pickled data specially-crafted for DoS attack.

Source Code 4 creation-attack-code.py

```
#!/usr/bin/python3

num = 5 * 024 * 1024
pickle_data = b'}' * num
pickle_data += b'.'
with open('malicious.pickle','wb') as f:
  f.write(pickle_data)
```

This code generates a pickled data file `malicious.pickle`. It consists of 5 * 1024 * 1024 repeats of the same opcode that pushes an `empty dictionary` object onto the stack and of one opcode `STOP`. When PM unpickles `malicious.pickle`, since an `empty dictionary` consumes 224 bytes memory in the stack, all generated `empty dictionary` objects consume about one gigabyte memory. These `empty dictionary` objects continue remaining in the stack until unpickling is finished.

4.2 Memory Consumption of Python Objects in the Stack

The script verification.py investigates memory consumption of Python objects in the stack in Python 3.5.2.

Source Code 5 verification.py

```
#!/usr/bin/python3

import tracemalloc
import logging
maxi = 5 * 1024 * 1024
cnt = 0
lis = []
logging.basicConfig(filename = 'tracemalloc.log',
                    level = logging.DEBUG)
tracemalloc.start()
snapshot1 = tracemalloc.take_snapshot()
while cnt < num:
  lis.append({})
  cnt = cnt + 1
  snapshot2 = tracemalloc.take_snapshot()
  top_stats = snapshot2.compare_to(snapshot1,
                                   "lineno")
  logging.debug("{0}{1}".format('='*50,
                                len(lis)))
  logging.debug("[Top 10 differences]")
  for stat in top_stats[:10]:
      logging.debug(stat)
```

This code reproduces the behavior of unpickling. Table 2, Table 3 and Table 4 show the results of running this code in Python 3.5.2 on Windows 10, Ubuntu 16.04 and CentOS 7, respectively.

These results say that the DoS attacks depend only on python environments and almost not on operating systems.

Next, the script estimate-deserializer.py investigates consumed memory of simple built-in objects in the stack by estimating memory consumption for 5 * 1024 * 1024 objects pushed onto the stack in Python 3.5.2.

Table 2 Windows 10

The number of pushed empty dictionaries	Memory size (B)	Memory block
10	2432	10
11	2656	11
12	2880	12
13	3104	13
14	3328	14
15	3552	15

Table 3 Ubuntu 16.04

The number of pushed empty dictionaries	Memory size (B)	Memory block
10	2368	10
11	2592	11
12	2816	12
13	3040	13
14	3264	14
15	3488	15

Table 4 CentOS 7

The number of pushed empty dictionaries	Memory size (B)	Memory block
10	2368	10
11	2592	11
12	2816	12
13	3040	13
14	3264	14
15	3488	15

Source Code 6 estimate-deserializer.py

```
#!/usr/bin/python3

import os
import sys
import subprocess
import shlex
import logging

logging.basicConfig(filename = 'output.log',
                    level = logging.DEBUG)

def make_malicious_file():
  NUM = 5 * 1024 * 1024
  head = b'\x8f'
  head = head * NUM
  tail = b'.'
  code = b''.join([head,tail])
  with open('malicious.pickle','wb') as f:
      f.write(code)

if __name__ == '__main__':
  make_malicious_file()
  p = subprocess.Popen('./deserializer.py',
                       stdout=subprocess.PIPE)
  command = "pmap -x " + str(p.pid) + " | tail -n 1"
  while p.poll() == None:
    monitor = subprocess.Popen(command,
                               stdout=subprocess.PIPE,
                               stderr=subprocess.PIPE,
                               shell=True)
    stdout_data,stderr_data = monitor.communicate()
    logging.debug(stdout_data)
  sys.exit(0)
```

Source Code 7 deserializer.py

```
#!/usr/bin/python3

import pickle

with open("malicious.pickle",mode="rb") as f:
  pickle.load(f)
```

The script estimate-deserializer.py makes a pickled data file `malicious.pickle` and runs the script deserializer.py which unpickles `malicious.pickle`. It estimates memory consumption in deserializer.py. The result is shown in Table 5, which says that an `empty set` object has the largest amplification for memory consumption. An `empty set` object consumes 241 byte memory for one byte opcode '`\x8f`'.

However the opcode '`\x8f`' of pushing an `empty set` object onto the stack works only in pickle protocol 4 and is not included in ASCII. In an application that supports only pickle protocol with its version less than 4, or allows only human-readable ASCII characters, the opcode '`\x8f`' cannot be used for the DoS attack. On the other hand, since the opcode '`}`' of pushing an `empty dictionary` object onto the stack works in all versions of pickle protocols and is an ASCII character, the opcode '`}`' is available in wider environment than the opcode '`\x8f`'.

Table 5 Opcode object

Opcode name	Object type	Opcode string	Consuming memory (kb)	Pickle protocol
BININT	For-byte signed int	J\x00\x00\x00\x00	74,140	0
BININT1	1-byte unsigned int	K\x00	74,144	0
BINSTRING	Counted binary string	T\x01\x00\x00\x00a	74,144	0
SHORT_ BINSTRING	Counted binary string less 256 bytes	U\x01a	74,148	0
BINUNICODE	Unicode string; counted UTF-8 string	X\x01\x00\x00\x00a	32,176	0
EMPTY_DICT	empty_dictionary	}	1,197,300	0
EMPTY_LIST	empty_list	]	404,388	0
EMPTY_TUPLE	empty_tuple	)	74,148	0
TRUE	Boolean Object:TRUE	I01\n	74,148	0
FALSE	Boolean Object:FALSE	I00\n	74,148	0
NEWTRUE	Boolean Object:TRUE	\x88	74,148	2
NEWFALSE	Boolean Object:FALSE	\x89	74,144	2
EMPTY_SET	empty_set	\x8f	1,236,772	4

4.3 Impact

This subsection discusses the threat of the DoS attacks using pickle by considering the following attack scenario of a service that manages Python objects using the pickle module. At a scheduled maintenance time, the service pickles Python objects, stores the pickled data to a file and halts. At the scheduled restart time, it restarts and unpickles the pickled data from the file to restore the Python objects. If the file that includes the pickled data is replaced with a specially-crafted data file `malicious.pickle` during the maintenance, the service at the restart time unpickles `malicious.pickles`. It causes huge memory consumption and may fail to restart.

Warning: The pickle module is not secure against erroneous or maliciously constructed data. Never unpickle data received from an untrusted or unauthenticated source.

Fig. 1 Pickle module warning

Table 6 Pickle and pickletools results

Consumed memory of unpickling (kb)	Consumed memory of disassembling (kb)
1,236,060	154,348

4.4 Mitigations

This subsection discusses mitigations for the DoS attacks. The most simple mitigation is not to unpickle data received from an untrusted or unauthenticated source as described Fig. 1.

Pickletools module defines some functions for analyzing pickled data. The `pickletools.dis` function disassembles and analyzes pickled data and does not unpickle it, so it consumes less memory than actual unpickling.
When `pickletools.dis` function reads `STOP` opcode, if there remain two or more objects in the stack, `pickletools.dis` function determines the pickled data invalid and outputs error code. For example, if `pickletools.dis` function reads `malicious.pickle`, it outputs "ValueError: stack not empty after STOP: [dict]". The DoS attacks can be mitigated by using the `pickletools.dis` function before unpickling. Table 6 compares consumed memory of unpickling `malicious.pickle` and that of disassembling `malicious.pickle` using pickletools.

5 Conclusion

This paper reported memory consumption DoS attacks in unpickling in the pickle module. In the DoS attack, PM pushes a lot of `empty dictionary` objects onto the stack and continues remaining the objects in the stack until `STOP` opcode. An `empty set` object consumes 241 byte memory in the stack. An opcode of pushing an `empty set` object onto the stack is one byte. So, the DoS attacks amplification is 241. The DoS attacks can be mitigated by using the `pickletools.dis` function before unpickling.

Appendix

Reply from Python Team

This subsection describes Python team's reply to our report on the DoS attacks (Fig. 2).

Changelog of Python Built-in Objects

This subsection describes the change history of Python built-in objects. Our DoS attacks were found in Python 3.5.2 and reported to Python team. However the changelog of the version 3.6.0 says as follows (Fig. 3).

This changelog is in python 3.6.0. An `empty dictionary` has reduced consumption memory with this change. However, an `empty set` consumes huge memory as in Python 3.5.2. The DoS attacks are still possible.

Thanks for your report.

I do not believe this is a vulnerability – a program that reads pickles from untrusted sources has bigger problems (it's easy to send a pickle that runs arbitrary code).

Fig. 2 Python team's reply

New dict implementation

The dict type now uses a "compact" representation based on a proposal by Raymond Hettinger which was first implemented by PyPy. The memory usage of the new dict() is between 20% and 25% smaller compared to Python 3.5.

The order-preserving aspect of this new implementation is considered an implementation detail and should not be relied upon (this may change in the future, but it is desired to have this new dict implementation in the language for a few releases before changing the language spec to mandate order-preserving semantics for all current and future Python implementations; this also helps preserve backwards-compatibility with older versions of the language where random iteration order is still in effect, e.g. Python 3.5).

(Contributed by INADA Naoki in bpo-27350. Idea originally suggested by Raymond Hettinger.)

Fig. 3 Dict object

References

1. Collections—Commons Collections Security Reports. https://commons.apache.org/proper/commons-collections/security-reports.html. Accessed 31 March 2018
2. Tomáš Polešovský. http://topolik-at-work.blogspot.jp/2016/04/java-deserialization-dos-payloads.html. Accessed 23 March 2018
3. Java-Deserialization-Cheat-Sheet. https://github.com/topolik/ois-dos/. Accessed 23 March 2018
4. Marco Slaviero—Sour Pickle. https://media.blackhat.com/bh-us-11/Slaviero/BH_US_11_Slaviero_Sour_Pickles_Slides.pdf. Accessed 23 March 2018
5. 12.1. Pickle Python object serialization—Python 3.5.5 documentation. https://docs.python.org/3.5/library/pickle.html#module-pickle. Accessed 23 March 2018
6. cpython/pickletools.py at master—python/cpython. https://github.com/python/cpython/blob/master/Lib/pickletools.py. Accessed 23 March 2018
7. cpython/pickle.py at master—python/cpython. https://github.com/python/cpython/blob/master/Lib/pickle.py. Accessed 23 March 2018