Web Forge Challenge Walkthrough

> **Web Forge Challenge Walkthrough**
>
> **Category:** Web
> **Tags:** SSRF, SSTI, Unix
> **Author:** Francesco Vecchi (0xGassin)

Problem Description

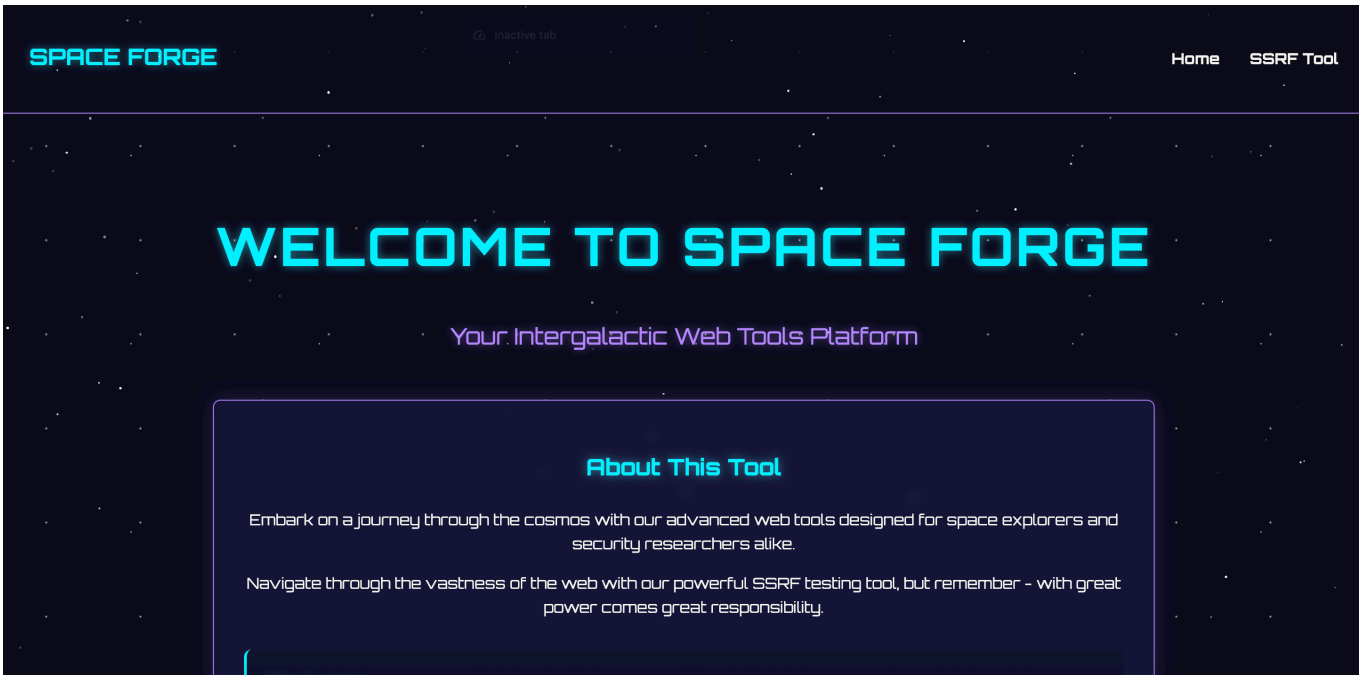
`Web Forge` is a chained web challenge combining a very basic **Authorisation bypass**, **SSRF** and **Server-Side Template Injection (SSTI)**. The challenge mentioned that fuzzing is allowed.

```
NOTE FROM ADMINS: Use of automated fuzzing tools are allowed for this
challenge. Fuzzing. Not Crawling. All endpoints aside from one are rate
limited.
```

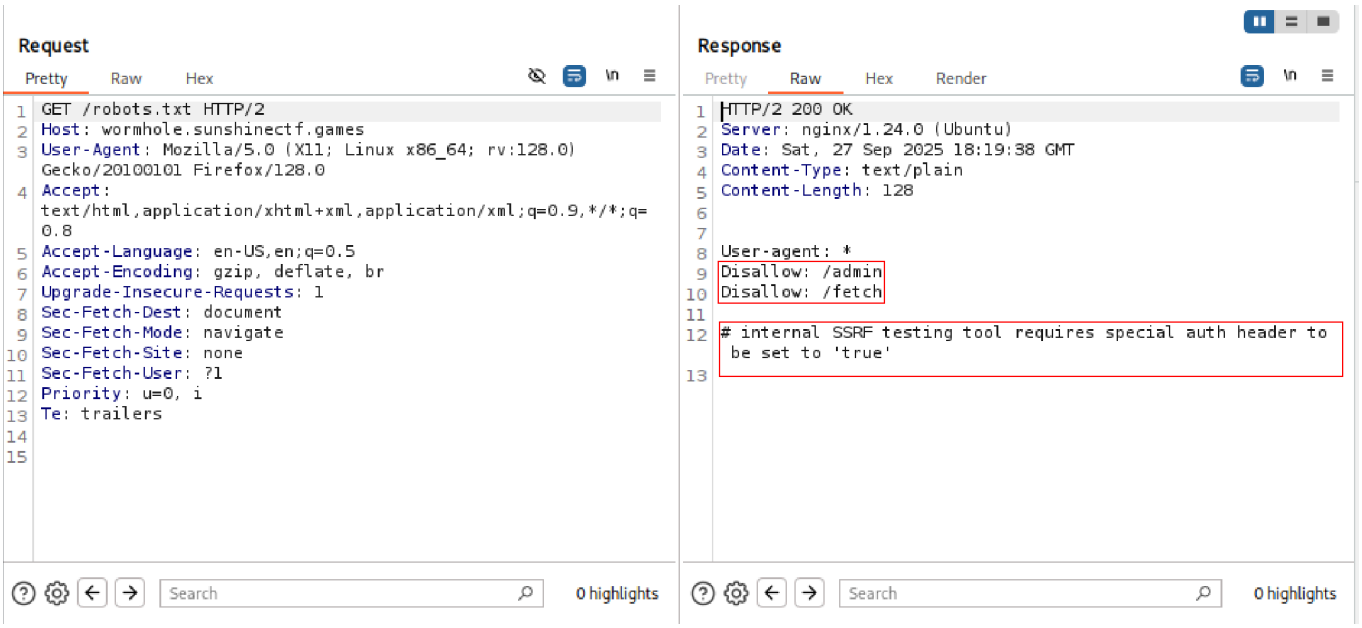
The general flow to pwn this machine was the following:

- discover a secret header
- use it to access `/fetch`
- use `/fetch` to SSRF into `127.0.0.1:8000`
- trigger SSTI on `/admin?template=`
- bypass character blacklist with hex escapes and read `flag.txt`.

Part 1 — Authorisation bypass (discovering the secret header)



Upon landing on the website (https://wormhole.sunshinectf.games/) we see mention of an SSRF tool somewhere on the website. Navigating to this tool, we reach the `/fetch` endpoint which displays a basic error message `403 Forbidden: missing or incorrect SSRF access header`. Great, we have to somehow find out the name of the header and the correct value.



A little more lurking around leads us to the `robots.txt` file, letting us know:

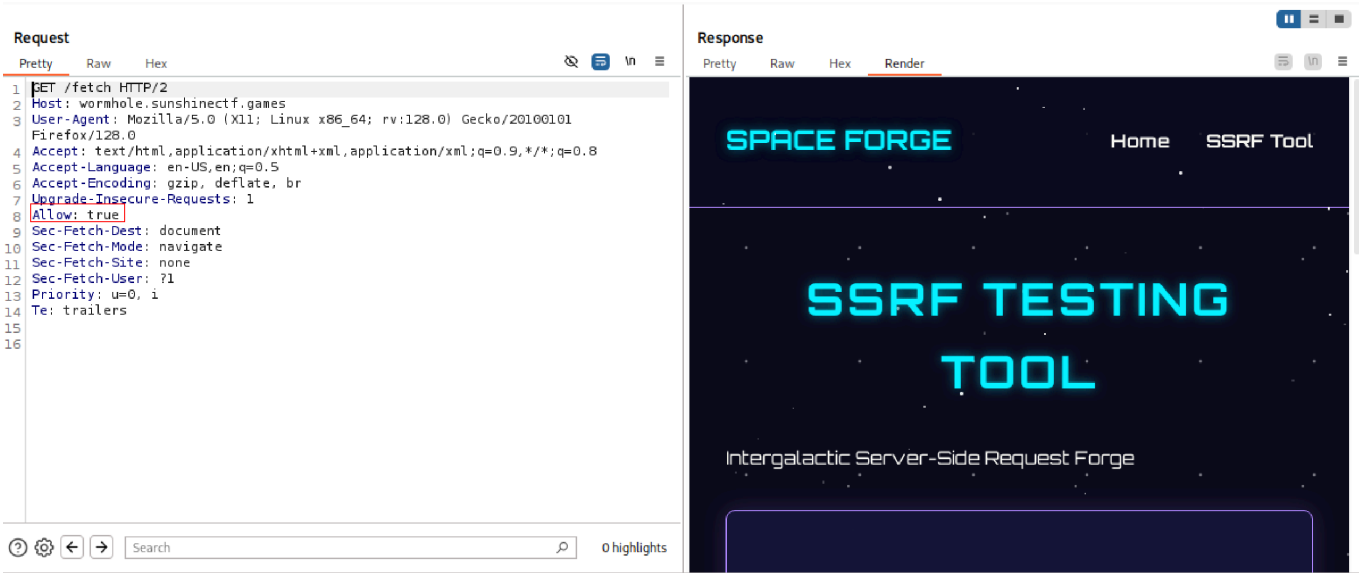
1. That there exists an `/admin` endpoint probably useful for later on in the challenge
2. That whatever header we're looking for must be set to `true`

With this extra knowledge we can now go back to the `/fetch` and enumerate possible request header names until a header is accepted. We can do so using intruder as follows:

1. Send a `GET /fetch` request to Intruder.

2. Set a payload position for the header *name* instead of a body parameter.
    - Create a header like: `{{FUZZ}}: true` (set header value to literal `true`).
3. Use `seclists` request-headers wordlist as the payload set (common header names).
4. Inspect responses — look at `Content-Length` / response code differences.

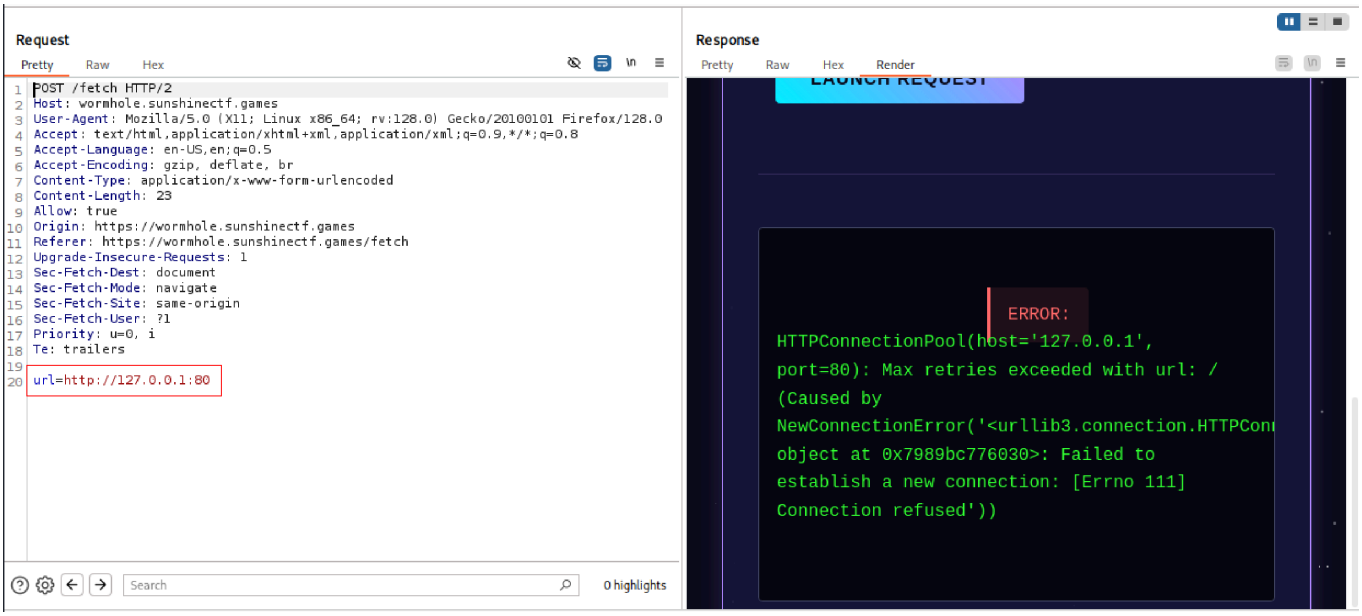
After running burpsuite, we can see that the request containing the header `Allow: true` changed the response revealing the secret header name is `Allow` and that its value must be `true`. If we now append that value to our request on the browser, the page renders.



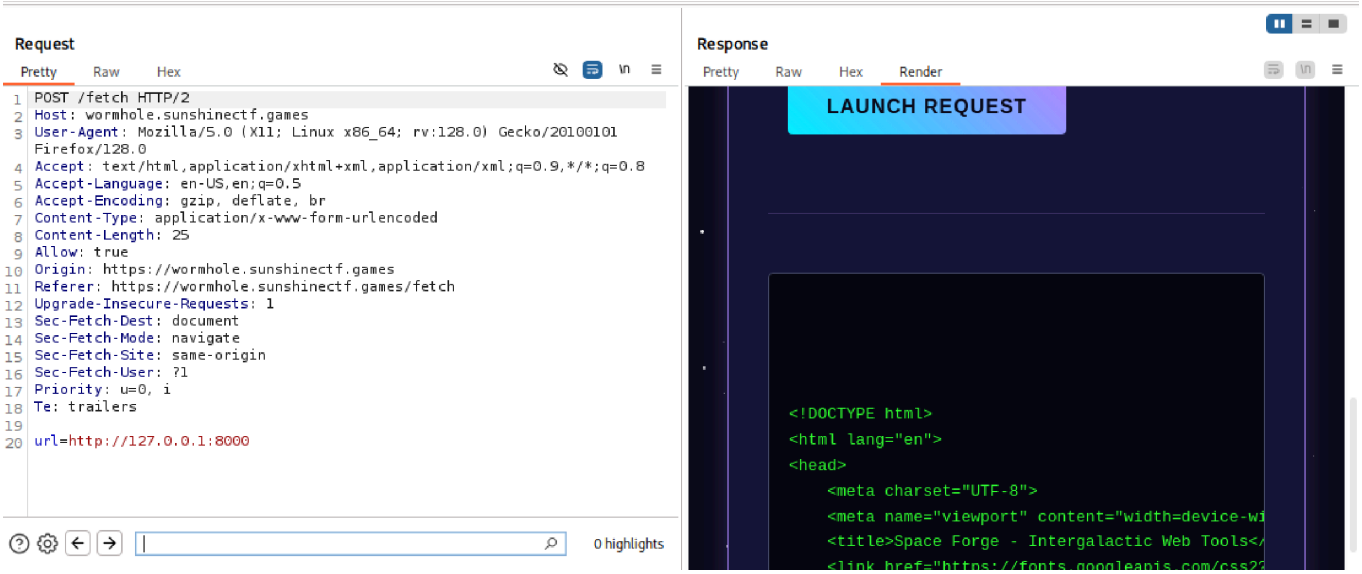
Part 2 — SSRF (finding an internal service)

Now that we have access to the tool, we can play around with the input to see what we can find. Since we have an input that takes in a URL, and SSRF is written all over the place, we can safely assume thats what it's going to be.

If we want to see what the tool does in the backend, we can simply enter a URL we control (such as collaborator). In doing so we can see that it makes HTTP requests to the provided resource and returns the contents of the GET request it makes. Entering the URL `http://127.0.0.1:80` for instance, appears to work, however it returns a connection error:



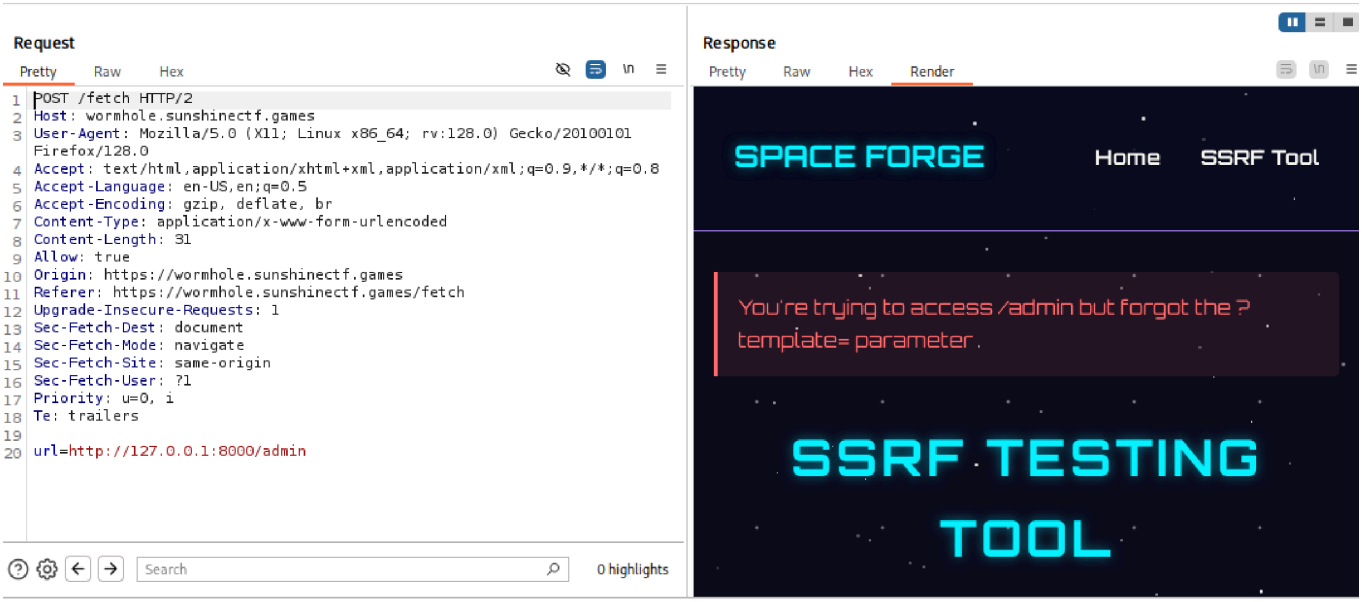
This confirms the presence of SSRF, and gives us a new goal, which is to discover which localhost port is used by the local app so we can reach the internal admin panel. Once again we can use bupsuite intruder on the post parameter `url=http://127.0.0.1:{{FUZZ}}/` with a wordlist of common ports. We can use nmaps wordlist or any wordlist of our choice really.



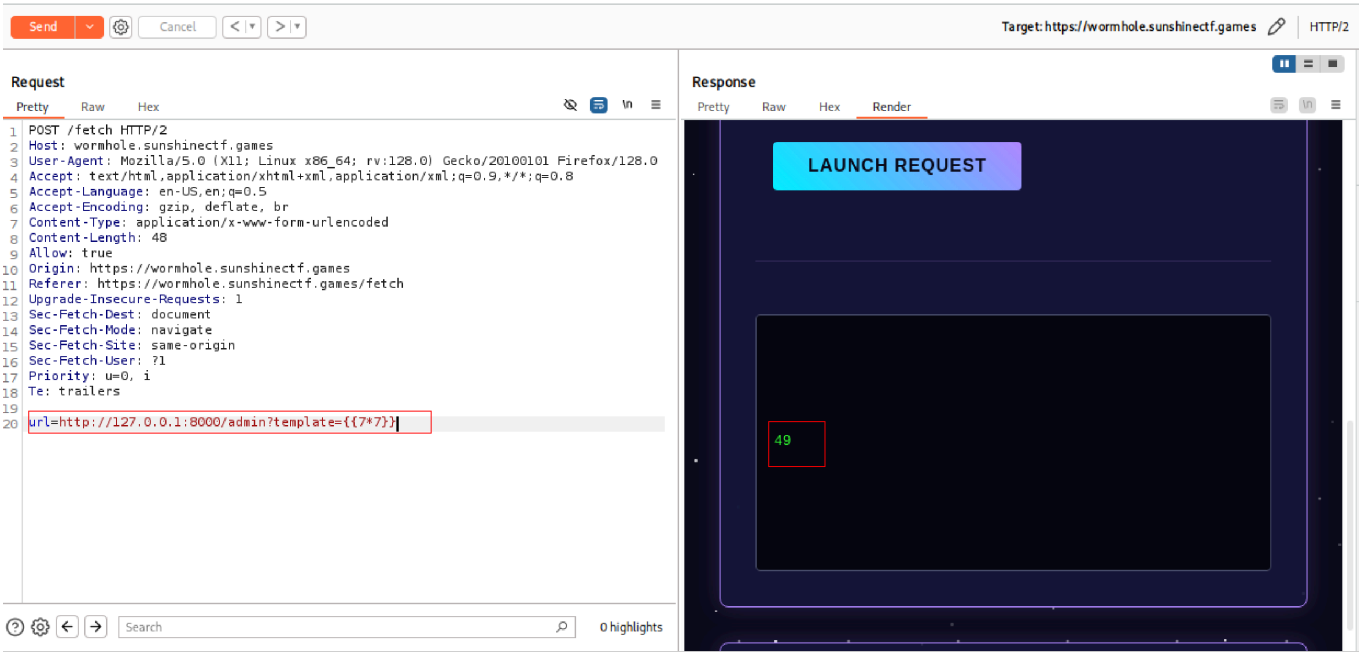
Port `8000` responded as an active web server. So this confirms that the web server is running internally on that port.

Part 3 — SSTI (admin template parameter)

When accessing `/admin` on the internal service, the app expects a query parameter `template=` if a directory is specified. The name suggests server-side template rendering (SSTI).



If we specify this parameter, we can see that whatever value we specify gets reflected in the server response. Adding the payload `?template={{7*7}}` returns `49` which is our proof of concept for the SSTI.



Now comes the hard part, we have to somehow weaponise this to ultimately get the flag. We can immediatly see that the template engine is likely to be `Jinja`, so payloads such as `{{config}}` and `{{cycler}}`) return valid (and useful) data, however the flag wasn't in any of these. With SSTIs the typical path to RCE is by importing `os` and running `popen`, however, the standard payloads didn't work and the server returns the message `Nope.`.

The challenge enforced a blacklist/WAF that blocked many characters used in exploitation: `.`, `[ ]` and others. This prevents directly referencing things like `__globals__`, `__builtins__`, `__import__`, etc. This now means that we have to try throw whatever bypass method we know at the serve and hope that one sticks.

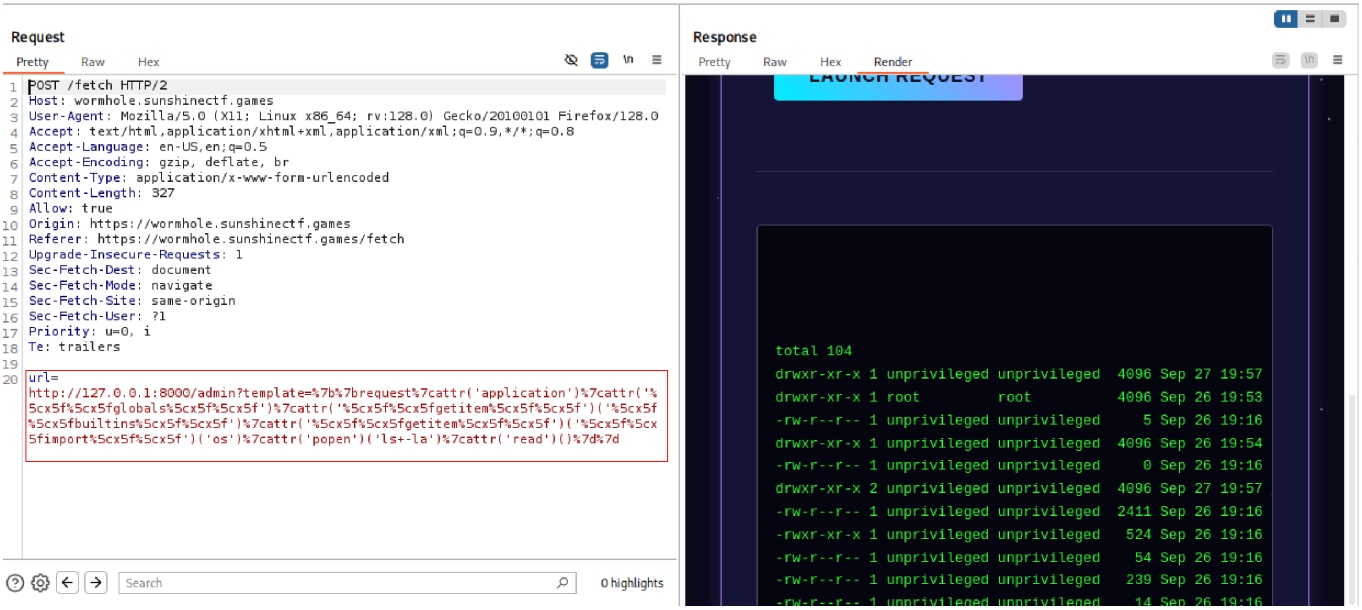
Bypass using hex escapes

The app allowed percent-encoded sequences and interpreted them inside the template engine. By replacing banned characters with hex escapes (`\xHH` sequences) — which after unescaping produce the required characters — we can reach `__globals__` , `__builtins__` , `__import__` , etc...

We can now use the `request` object -> `application` -> `__globals__` -> `__getitem__` -> `__builtins__` -> `__import__` to import `os` , call `popen('cat flag.txt')` and `.read()` .

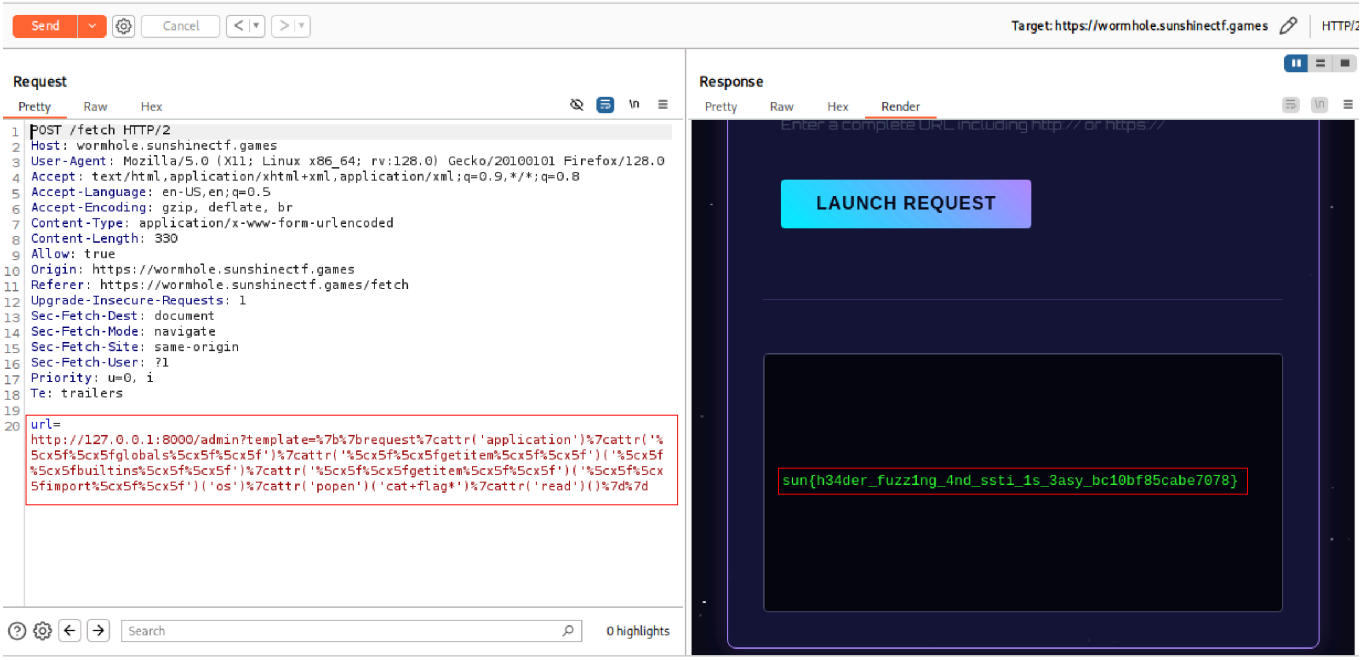
This leads us to our final (PoC) payload which performs a simple directory listing in the current directory:

```
http://127.0.0.1:8000/admin?template=
{{request|attr('application')|attr('\x5f\x5fglobals\x5f\x5f')|attr('\x5f\x5f
getitem\x5f\x5f')
('\x5f\x5fbuiltins\x5f\x5f')|attr('\x5f\x5fgetitem\x5f\x5f')
('\x5f\x5fimport\x
```



This (although hard to see from the screenshot) shows us the command `ls -la` , which is great news! If we change the command to cat out the flag (using the hex bypass for the dot once again) and URL encode it, we get the following final payload:

```
http://127.0.0.1:8000/admin?
template=%7b%7brequest%7cattr('application')%7cattr('%5cx5f%5cx5fglobals%5cx
5f%5cx5f')%7cattr('%5cx5f%5cx5fgetitem%5cx5f%5cx5f')
('%5cx5f%5cx5fbuiltins%5cx5f%5cx5f')%7cattr('%5cx5f%5cx5fgetitem%5cx5f%5cx5f
')('%5cx5f%5cx5fimport%5cx5f%5cx5f')('os')%7cattr('popen')
('cat+flag\x2Etxt')%7cattr('read')()%7d%7d
```



FLAG: `sun{h34der_fuzz1ng_4nd_ssti_1s_3asy_bc10bf85cabe7078}`

Final Notes - Why the payload works (brief explanation)

- Jinja (or the template engine in use) templates can be abused to access Python internals via `__globals__`, `__builtins__`, and `__import__`.
- The chain: `request → application → __globals__ → __getitem__('__builtins__') → __getitem__('__import__')('os') → os.popen('cat flag.txt').read()` executes a shell command and returns its output.
- The challenge had a blacklist blocking characters like `.` `[` `]` etc., so we encoded underscores and other problematic tokens using hex escapes (`\x5f` → `_`, `\x2E` → `.`), then percent-encoded the entire string for transport in the URL. The server unescaped and processed the resulting template, allowing the exploit.