Vulnerability Assessment Report: pygoat

Repository: Akaolisangwu-projects/pygoat
Analyst: Akaolisa Ngwu

Summary

PyGoat is an intentionally vulnerable Django web application used for OWASP Top-10 training. Its purpose means many dangerous patterns exist by design and the repository must never be deployed as-is to production.

Key confirmed issues include:

- Public demo credentials published in the README.
- Docker/requirements files that allow trivial reproduction of the vulnerable environment.
- Use of dependency versions that are likely outdated.

These upgrades reduce container attack surface and demonstrate good security hygiene by integrating automated Snyk scans.I have already taken important remediation steps by merging two Snyk pull requests that upgrade the Docker base images to a more secure Python release, reducing OS- and runtime-level CVEs.

Changes Already Addressed (via Snyk)

Changes	From	To	Benefit
1	python:3.12.0a5-slim	python:3.14.0rc2-slim-trixie	Moves from an alpha image to a later patched release candidate, eliminating multiple Debian/CPython CVEs.
2	python:3.7.5-buster	python:3.14.0rc2-slim-trixie	Replaces an end-of-life 3.7 base (no fixes since 2023) with a maintained 3.14 release, removing known Debian-Buster vulnerabilities.

Confirmed Vulnerabilities & Risks

Category	Evidence	Risk	Recommended Remediation
Public demo credentials	README shows username: user / password: user12345	Critical	Remove or clearly sandbox these credentials; rotate any reused secrets immediately.
Reproducible vulnerable environment	Presence of requirements.txt, Dockerfile, and docker-compose.yml	High	Keep these files, but pin safe package versions and mark the repo as training only.
outdated Python dependencies	Requirements file exists but versions not audited	High	Run pip-audit/safety and update packages to patched versions.

Probable High-Risk Issues

These are common to intentionally vulnerable Django apps; confirm with static/dynamic scans:

- SQL Injection: raw SQL queries or unsafe `raw()` calls.
- Cross-Site Scripting (XSS): unescaped template output or use of `mark_safe`.
- CSRF Disabled: missing middleware or `{% csrf_token %}`.
- Weak Authentication: hard-coded or weak passwords, missing rate limits.
- Sensitive Data Exposure: debug mode enabled, hard-coded `SECRET_KEY`.
- Insecure Deserialization: use of `pickle` or unsafe YAML.
- Insufficient Logging & Monitoring.

Recommended Remediation Plan

Immediate (0-2 days)

1. Remove/rotate demo credentials in the README or ensure they point only to a hardened sandbox.
2. Clearly mark the repo and Docker images as *training only* to prevent accidental production deployment.
3. Set `DEBUG = False`, move `SECRET_KEY` to environment variables, and configure `ALLOWED_HOSTS`.

Short Term (1 week)

1. Run dependency and container scans
2. Patch or upgrade all high/critical CVEs found.
3. Add pre-commit hooks for secret detection (`detect-secrets`, `git-secrets`).

Medium Term (2–4 weeks)

1. Perform **static code analysis** (`bandit -r .`) and **dynamic scanning** (OWASP ZAP) to confirm SQLi/XSS/CSRF.
2. Harden Docker images further: use a non-root user, minimal base image, and multistage builds.
3. Implement CI/CD gates that fail on high severity findings.

Long Term

1. Maintain a secure production branch and keep intentionally vulnerable code only in a clearly labeled training branch.
2. Add structured logging and monitoring to detect exploitation attempts.

Key Hardening Checklist

- ☐ Remove credentials from README or rotate them.
- ☐ `DEBUG=False` in production settings.
- ☐ `SECRET_KEY` loaded from environment variable.
- ☐ `SESSION_COOKIE_SECURE`, `CSRF_COOKIE_SECURE`, `SECURE_HSTS_SECONDS` configured.
- ☐ Run `pip-audit/safety` regularly.
- ☐ Integrate Snyk scans into CI for both Python dependencies and Docker images.