SECURE CODING REVIEW — PASSWORD-SECURITY-TOOL

Target: password-security-tool (Python CLI/tool)

Scope: source code (password handling, I/O), third-party dependencies, logging & error handling, cryptography usage, build/CI.

Note / assumptions: I did this review as a static conceptual audit (no repository clone). I assumed typical patterns found in small password tools: CLI input or file input, optional password storage or caching, hashing functionality, and use of third-party libs. If you want repository-specific findings, paste key files (or allow me to fetch) and I'll adapt.

EXECUTIVE SUMMARY

The project is a strong candidate for a secure-coding review because it handles sensitive secrets (passwords). That raises risk if input handling, storage, hashing, logging, or error handling are weak.

Typical high-risk issues to check and remediate:

Plaintext password logging or printing (high).

Using weak or unsalted hashing (high).

Storing passwords to disk without encryption (critical).

Hard-coded secrets or keys (high).

Insecure dependency versions (medium).

Unsanitized input used in shell commands or file paths (medium).

Priority: Fix logging & storage first, then crypto algorithm choices, then dependency hygiene and CI checks.

FINDINGS (LIKELY / COMMON PATTERNS) — PRIORITIZE TRIAGE

I list likely findings relevant to small password tools. Mark each with risk and recommended remediation.

1) PASSWORDS PRINTED TO CONSOLE OR LOGS

Risk: Passwords or derived values may appear in logs, CI output, or terminal — total compromise if log files are exfiltrated.

Recommendation: Never print or log raw passwords or any reversible secret. Use `getpass.getpass()` for interactive input. Mask any UI echoes. Remove any print(password) or logging lines that include password strings.

Example fix:

```
import getpass password = getpass.getpass("Enter password: ") # do not print(password) or log it
```

2) PLAINTEXT STORAGE / WRITING TO DISK WITHOUT ENCRYPTION

Risk: If the tool caches passwords or writes results to a file (e.g., wordlists, temporary files), they may be accessible.

Recommendation: Avoid storing plain passwords. If you must, use OS keystore (keyring) or encrypt files with a key that is not in source. Use secure temp files and remove them immediately. Prefer hashing over storage.

Example fix (avoid file storage):

If only checking strength, do not store the password. If persistent storage needed, encrypt with a user password-derived key (see crypto recommendations below).

3) WEAK HASHING OR INCORRECT USE OF HASHING FUNCTIONS

Risk: Using MD5, SHA1, or plain SHA256 for passwords is weak for password storage (fast hashes).

Recommendation: Use a slow, adaptive hashing function with salt: bcrypt, scrypt, or argon2 (Argon2id preferred). Use well-maintained libs: argon2-cffi or bcrypt.

Example using argon2 (python):

```
from argon2 import PasswordHasher ph = PasswordHasher() hash = ph.hash(password) # store hash only ph.verify(hash, provided_password) # verify
```

4) HARD-CODED SECRETS OR CREDENTIALS IN SOURCE CODE

Risk: API keys, test credentials, or encryption keys in code may leak (especially if repo is public).

Recommendation: Remove secrets. Use environment variables, config files excluded via .gitignore, or a secret manager. Scan repo for strings that resemble secrets.

Quick scan tip: run `git grep -I --line-number -n "password\|secret\|api_key\|token\|credentials"` and review results.

5) INSECURE DEPENDENCY VERSIONS / SUPPLY-CHAIN RISK

Risk: Dependency vulnerabilities could be exploited to exfiltrate data or run arbitrary code.

Recommendation: Use pip-audit or safety to scan for known insecure packages. Pin dependencies in requirements.txt. Add dependency scanning in CI.

Commands:

```
pip install pip-audit pip-audit # or pip install safety safety check
```

6) UNHANDLED EXCEPTIONS THAT LEAK STACK TRACES (DEBUG INFO)

Risk: Detailed stack traces printed to console or logs can leak file paths, secrets, or internal details.

Recommendation: Catch exceptions at top level and show user-friendly errors; log detailed errors only to secure logs and scrub secrets before logging.

Example:

```
try: do_sensitive_work() except Exception as e: logger.error("An error occurred (scrubbed)") # log scrubbed message # optionally log detailed trace to secure storage, not stdout
```

7) UNSAFE SUBPROCESS OR SHELL USAGE WITH USER INPUT

Risk: If user input is used inside `os.system()`, `subprocess` with `shell=True`, or file paths unsanitized, there's injection risk.

Recommendation: Use `subprocess.run([...])` with args list (no shell), or sanitize/validate inputs strictly.

CONCRETE REMEDIATION PLAN & CODE PATCHES

A. REPLACE ANY INPUT() FOR PASSWORDS WITH GETPASS.GETPASS()

```
# BAD password = input("Enter password: ")
```

```
# GOOD import getpass password = getpass.getpass("Enter password: ")
```

B. USE ARGON2 FOR HASHING — EXAMPLE

Add dependency: `argon2-cffi`

```
pip install argon2-cffi
```

```
from argon2 import PasswordHasher, exceptions ph = PasswordHasher() # Create hash (store only this) password_hash = ph.hash(password) # Verify try: ph.verify(password_hash, given_password) except exceptions.VerifyMismatchError: # invalid password pass
```

C. AVOID WRITING RAW PASSWORDS TO FILES / LOGS

If generating a report, mask passwords (e.g., show only last 2 characters or length).

Use secure temp files: `tempfile.NamedTemporaryFile(delete=True)`.

D. SECRETS REMOVAL / USE ENVS

Move any keys to environment variables; do not commit .env. Example using python-dotenv for dev only.

`.env` (dev) should be .gitignored. Use `os.environ.get('SECRET_KEY')` in code.

E. ADD LINTERS & SECURITY SCANNERS

Add bandit for static security linting: `pip install bandit` and run `bandit -r .`

Add pip-audit to check dependency vulns: `pip-audit`

Add flake8 / pylint for code quality.

CI INTEGRATION (RECOMMENDED)

Add a GitHub Actions workflow that runs on PRs and enforces checks:

`python -m pip install -r requirements-dev.txt`

Run unit tests (pytest)

Run bandit -r . (fail on high severity)

Run pip-audit (fail if vulnerabilities found)

Optionally run safety check --file=requirements.txt

Example .github/workflows/security.yml (short sketch):

```
name: Security Checks on: [pull_request, push] jobs: security: runs-on: ubuntu-latest steps: - uses: actions/checkout@v4 - name: Set up Python uses: actions/setup-python@v4 with: python-version: 3.11 - name: Install run: pip install -r requirements-dev.txt - name: Bandit run: bandit -r . - name: pip-audit run: pip-audit - name: Run tests run: pytest -q
```

TESTING RECOMMENDATIONS

Add unit tests for hashing and verification, ensuring invalid attempts fail and salts are randomized.

Add tests for CLI to ensure passwords are never echoed (use pexpect or pytest capsys).

Add a test that asserts no secrets are present in the repo (use detect-secrets or a simple grep in CI).

TOOLS & COMMANDS — QUICK STARTER LIST

static scanning

```
pip install bandit pip-audit
```

run scans

```
bandit -r path/to/project pip-audit
```

dependency check (alt)

```
pip install safety safety check
```

search for obvious secrets

```
git grep -I --line-number -n "password\|secret\|api_key\|token"
```

EXAMPLE REMEDIATION PR DESCRIPTION (YOU CAN USE THIS TEXT)

Fix: secure password handling + dependency scan

Replace input() with getpass.getpass() for password entry to prevent terminal echo.

Replace legacy hash usage with Argon2 via argon2-cffi and update verification logic.

Remove prints/logs that contained raw password data; mask sensitive info in output.

Add bandit & pip-audit to dev requirements and GitHub Actions workflow to run security checks on PRs.

Added unit tests for hashing/verification and CLI behavior.