VectraOps

Knowledge Library

CHECKLIST

Python Security Checklist for Safe File and Data Handling

A practical checklist for reviewing Python file and data handling before production. Verify input validation, path safety, serialization controls, logging, permissions, and rollback criteria.

Programming - July 06, 2026

Purpose

Unsafe file and data handling is one of the fastest ways to turn otherwise clean Python code into an operational risk. A script can appear correct in unit tests and still expose directory traversal, deserialization abuse, file overwrite, data leakage, or corrupted output when it processes untrusted paths, payloads, archives, uploads, or configuration data.

Use this checklist to verify whether a Python component handles files and data safely before production use. After completing it, you should be able to decide whether the implementation is suitable, apply a practical review workflow, and confirm what evidence must be collected before release.

How to use this checklist

Review the checklist in order, because each phase builds on the previous one. For every item, confirm the behavior with code review, automated tests, config inspection, or operational evidence. Do not mark an item complete unless you can point to a concrete artifact such as a test case, log sample, permission setting, or signed-off review note.

Use the acceptance criteria as a gate, not as guidance. If a check fails, treat it as a deployment blocker unless the risk is explicitly accepted and documented.

Phase 1: Define data trust boundaries

This phase establishes what data the code can trust and where protection controls must exist. It prevents teams from assuming that local files, API payloads, queue messages, or user-supplied paths are safe by default.

Checklist items

- ? Confirm which inputs are untrusted, partially trusted, and fully trusted.
- ? Review every file source, path input, archive, and payload field that influences file access or parsing.
- ? Validate that externally supplied paths are normalized before use.
- ? Document which code paths handle secrets, credentials, personal data, or regulated records.
- ? Assign a named owner for each data flow that crosses a trust boundary.

Evidence to capture

Capture a short data-flow note, an input inventory, and references to the functions or modules that consume each input. If the component accepts serialized objects or JSON payloads, align the review with the same control discipline used in a Python Checklist: Production Readiness Review so the file-handling review does not miss deployment-sensitive behavior.

Acceptance criteria

- Every input that can affect file location, file content, or parsing behavior is identified.
- No trust boundary is implied without a documented control.
- The code owner for each boundary is named and reachable.

Owner and review cadence

Owner: application engineer or module maintainer. Review cadence: on every significant code change and before production deployment.

Common mistakes

- Assuming internal queue messages are safe because they are not user-facing.
- Forgetting that configuration values can become attack inputs when they control file paths or formats.
- Documenting trust boundaries at a high level but not mapping them to specific code.

Phase 2: Validate path and filename handling

This phase checks whether the code can be tricked into reading, writing, or overwriting files outside the intended location. Path manipulation issues are especially common when code joins user input to base directories without strict validation.

Checklist items

- ? Confirm that all user-controlled paths are resolved against an approved base directory.
- ? Review whether path normalization happens before any read, write, move, or delete operation.
- ? Validate that traversal sequences such as ../ cannot escape the intended directory.

- ? Test that symbolic links, hard links, and mounted paths do not bypass restrictions.
- ? Document the exact rules for allowed extensions, prefixes, and filename characters.
- ? Assign a review for any code that constructs filesystem paths dynamically.

Evidence to capture

Capture test cases covering safe and unsafe inputs, including absolute paths, traversal attempts, and malformed names. Record the final resolved path for each case and the expected allow or deny outcome.

Acceptance criteria

- Only approved paths are reachable.
- Unsafe path patterns are rejected before file access occurs.
- The code does not rely on filename filtering alone to prevent escape from the base directory.

Owner and review cadence

Owner: application engineer with review from security or platform engineering. Review cadence: every change that touches file I/O or upload handling.

Common mistakes

- Checking for .. in the raw string but not validating the resolved path.
- Using string concatenation for filesystem paths.
- Allowing archive extraction to write files without destination containment checks.

Phase 3: Control file creation, overwrite, and permission behavior

This phase ensures files are created with safe permissions and that existing files are not overwritten unintentionally. It is critical for logs, exports, reports, caches, and temporary data.

Checklist items

- ? Confirm that new files are created with explicit, least-privilege permissions.
- ? Validate that sensitive outputs are not world-readable by default.
- ? Review whether the code prevents accidental overwrite of existing files.
- ? Test that temporary files are created securely and cleaned up reliably.
- ? Document any exception cases where overwrite is expected and approved.
- ? Assign ownership for filesystem permissions in the runtime environment.

Evidence to capture

Capture runtime permission settings, file mode expectations, and test results for create, overwrite, and cleanup cases. Record how the component behaves when the target file already exists.

Acceptance criteria

- File permissions match the data sensitivity level.
- Overwrite behavior is deliberate, not accidental.
- Temporary files are not left behind after success or failure.

Owner and review cadence

Owner: platform engineer or service owner. Review cadence: at deployment review and after any storage or runtime change.

Common mistakes

- Relying on the operating system default umask without confirming it in production.
- Writing sensitive exports to shared directories.
- Leaving partial files behind when a process crashes mid-write.

Phase 4: Review parsing and deserialization safety

This phase verifies that Python code does not accept formats or loaders that can execute unintended behavior or consume excessive resources. Parsing and deserialization should be deliberately constrained, especially when dealing with external API data, archives, config files, or serialized application state.

Checklist items

- ? Confirm that the chosen parser matches the expected data format.
- ? Validate that unsafe deserializers are not used for untrusted input.
- ? Review whether the code enforces size, depth, and type limits before parsing.
- ? Test failure handling for malformed, truncated, and oversized payloads.
- ? Document allowed schemas, fields, and fallback behavior for invalid data.
- ? Assign a code owner for every custom parser or transformation layer.

Evidence to capture

Capture parser selection, schema validation logic, and negative test cases. If the component reads external JSON, ensure the invalid-response handling is explicit and predictable, following the same safe-failure discipline used in a Python JSONDecodeError Troubleshooting for Invalid API Responses.

Acceptance criteria

- Untrusted input is parsed only by a safe, intended parser.
- Invalid or malformed data fails closed with a controlled error.
- Resource limits prevent a single payload from exhausting memory or CPU.

Owner and review cadence

Owner: application engineer, with security review for high-risk formats. Review cadence: when new formats are added and during incident-driven revalidation.

Common mistakes

- Deserializing untrusted objects because it is more convenient than validating plain data.
- Accepting any structure and validating it only after parsing completes.
- Treating parser exceptions as harmless without checking for partial writes or retries.

Phase 5: Validate content integrity and type expectations

This phase checks that the code verifies content type, file type, and basic integrity before processing. A file extension alone does not prove the content is safe or well-formed.

Checklist items

- ? Confirm that file content is validated independently of the filename.
- ? Review whether MIME type, magic bytes, or schema checks are used where appropriate.
- ? Test that the code rejects mismatched extensions and content signatures.
- ? Validate checksum or signature checks for files that require integrity guarantees.
- ? Document the expected encoding and newline behavior for text files.
- ? Assign ownership for any integrity verification performed at ingest.

Evidence to capture

Capture examples of valid and invalid files, plus the expected rejection reason for each invalid case. Include checksum, signature, or schema verification results when those controls apply.

Acceptance criteria

- The component does not rely on extension matching alone.
- Invalid or tampered content is rejected before it reaches business logic.
- Encoding assumptions are explicit and testable.

Owner and review cadence

Owner: application engineer or data pipeline owner. Review cadence: whenever file formats, sources, or ingest rules change.

Common mistakes

- Trusting .json, .csv, or .txt as proof of safe content.
- Allowing encoding errors to be silently repaired in ways that change meaning.
- Ignoring checksum verification for files that are expected to be tamper-evident.

Phase 6: Protect secrets and sensitive data in memory, logs, and exports

This phase focuses on preventing accidental disclosure during file reads, writes, logging, or error handling. Even safe parsing can leak sensitive values if debug output, tracebacks, or export files are not controlled.

Checklist items

- ? Confirm that secrets are never written to plain-text files unless explicitly required and approved.
- ? Review whether logs redact credentials, tokens, personal data, and session identifiers.
- ? Validate that error messages avoid exposing file paths, raw payloads, or sensitive field values.
- ? Test that backup, temp, and cache files are excluded from sensitive-data retention gaps.
- ? Document retention, deletion, and encryption requirements for sensitive outputs.
- ? Assign an owner for redaction rules and secure storage settings.

Evidence to capture

Capture sample log lines, exported file samples, and the redaction policy in effect. Verify that error traces and debug output do not reveal secrets or sensitive paths.

Acceptance criteria

- Sensitive data is minimized in files, logs, and diagnostics.
- Redaction and retention rules are consistent across environments.
- Error handling reveals enough to diagnose issues without exposing confidential content.

Owner and review cadence

Owner: security or application owner, depending on the data class. Review cadence: every logging change and every data-classification update.

Common mistakes

- Logging raw request bodies during parse failures.
- Storing credentials in local scratch files for convenience.
- Forgetting that debug mode often changes file and error output behavior.

Phase 7: Test failure handling, rollback, and safe recovery

This phase verifies that the component fails safely when file access, parsing, or validation breaks. A secure implementation should stop, report, and recover without leaving corrupted or partially processed data behind.

Checklist items

- ? Confirm that failed reads and writes do not leave partial artifacts in the final location.
- ? Review whether retries are safe for idempotent and non-idempotent operations.
- ? Validate that rollback removes temporary outputs and restores previous state where required.
- ? Test behavior when disk space, permissions, or locks are unavailable.
- ? Document which failures must stop processing immediately.
- ? Assign an operational owner for recovery actions and incident handling.

Evidence to capture

Capture simulated failure results for permission denial, missing files, malformed inputs, and storage exhaustion. Record whether the component retries, aborts, or rolls back as designed.

Acceptance criteria

- The component fails closed when safety checks fail.
- Partial work is not exposed as valid output.
- Recovery steps are documented and practical for operators.

Owner and review cadence

Owner: service owner and on-call operations lead. Review cadence: during pre-production testing and after incident review.

Common mistakes

- Retrying a bad payload without changing the cause of failure.

- Leaving temporary files in a shared directory after exceptions.
- Treating partial output as acceptable because the process exited successfully.

Phase 8: Verify automation, monitoring, and review evidence

This phase makes the checklist operational. Security controls that cannot be observed, tested, or audited are easy to bypass unintentionally.

Checklist items

- ? Confirm that file-handling tests run in CI for safe and unsafe cases.
- ? Review whether static analysis or linters flag risky filesystem and serialization patterns.
- ? Validate that alerts exist for repeated parse failures, permission errors, and unexpected file writes.
- ? Document the approval record for any accepted exceptions.
- ? Test that dashboards or logs can distinguish validation failures from system failures.
- ? Assign responsibility for ongoing control monitoring.

Evidence to capture

Capture CI output, test reports, and alert samples. Include evidence that security-sensitive checks are part of the normal release workflow rather than an ad hoc review.

Acceptance criteria

- The checks are repeatable and visible to reviewers.

- Failures are observable before production impact.
- Exceptions are documented, time-bounded, and owned.

Owner and review cadence

Owner: engineering lead with support from security and operations. Review cadence: every release and during periodic control audits.

Readiness scoring

Use the scoring below to decide whether the component is ready for production use. Score each phase from 0 to 2:

- 0 = not implemented or not evidenced
- 1 = partially implemented, but evidence or coverage is incomplete
- 2 = implemented and evidenced with passing tests or review artifacts

A total score of 14 to 16 indicates strong readiness. A score of 10 to 13 indicates conditional readiness with documented follow-up actions. A score below 10 means the component is not ready for production until the gaps are closed.

Interpretation rules

- If any phase scores 0 in path safety, parsing safety, or secret handling, treat the review as failed regardless of total score.
- If evidence exists but cannot be reproduced by another reviewer, cap that phase at 1.
- If an exception is accepted, it must have an owner, expiration date, and compensating control, or it counts as a failure.

Pass/fail criteria

Pass the review only when every high-risk control has passing evidence and no blocking exceptions remain. The implementation should demonstrate safe path resolution, controlled parsing, explicit file permissions, no secret leakage, tested failure handling, and observable monitoring.

Fail the review if any of the following are true:

- untrusted input can influence file location without strict validation
- unsafe deserialization or unconstrained parsing is used for external data
- sensitive information is exposed in logs, errors, temp files, or exports
- partial writes or recovery behavior are not tested
- required evidence is missing or inconsistent

Concrete follow-up actions

When the review is incomplete, document the gaps, assign an owner, and set a verification deadline. If a risk is found in production, disable the unsafe code path, rotate any exposed secrets, and re-run the relevant tests before restoring normal operation.

If the component passes, keep the controls in the release workflow so the same checks apply to future changes, not just the current implementation. That is what keeps safe file and data handling reliable after the initial review is finished.

Use this guidance together with Hadoop cluster hardening checklist and Spark shuffle failures to connect the workflow with related operational context already available on the site.