LangChain Core Prompt Loader File-Read Report
Executive summary

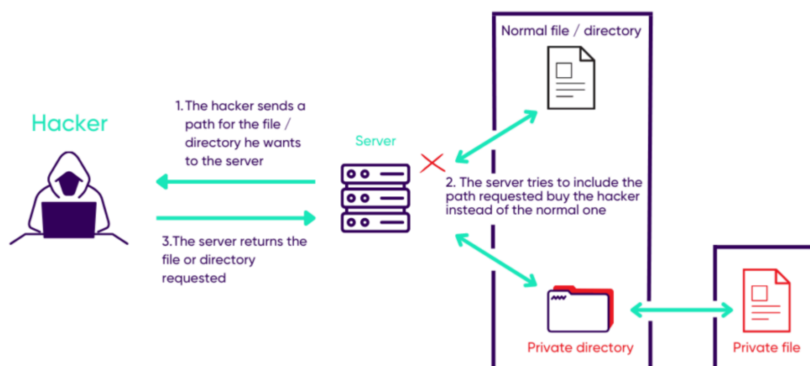
This report documents a file-path handling issue in langchain_core prompt loading. The core security problem class is path traversal / arbitrary file read, where attacker-influenced path values can cause the application to read files outside the intended trust boundary. Path traversal attacks commonly rely on ../ sequences or absolute paths, and they can expose configuration, source code, credentials, and other sensitive files.

My lab notes show that load_prompt_from_config() was exercised with several path shapes, including relative files, traversal-style input, and symlink redirection. The older test notes describe successful disclosure from a symlink target and traversal-style paths, while the newer lab run shows the current version blocking absolute paths and .. traversal.

Affected component

The relevant component is the prompt-loading path inside langchain_core, especially load_prompt_from_config() and related file-backed fields such as template_path, examples, and example_prompt_path. My notes show that these inputs are treated as file paths and that the loader reads file contents directly into the prompt object.

The security concern is not “a file loader exists.” It is that path input is security-sensitive, and safe handling must account for canonicalization, symlinks, and directory boundaries. Python’s Path.resolve() is specifically the operation used to resolve symlinks and eliminate .. components, which is why canonical-path validation is essential in path-sensitive code.



What the lab proves

Safe relative file loading works

My lab created a controlled file in /tmp/lab, pointed link.txt at it, and load_prompt_from_config() returned the file contents successfully:

normal.txt → link.txt → [SUCCESS] 'NORMAL_FILE\n'
secret.txt → link.txt → [SUCCESS] 'SECRET_TEST_DATA\n'

This proves the loader accepts relative file paths and follows symlinks when the final target remains inside the allowed scope. That is expected file-loading behavior, but it is also the behavior that must be carefully bounded in security-sensitive contexts.

Directory traversal is rejected in the newer lab

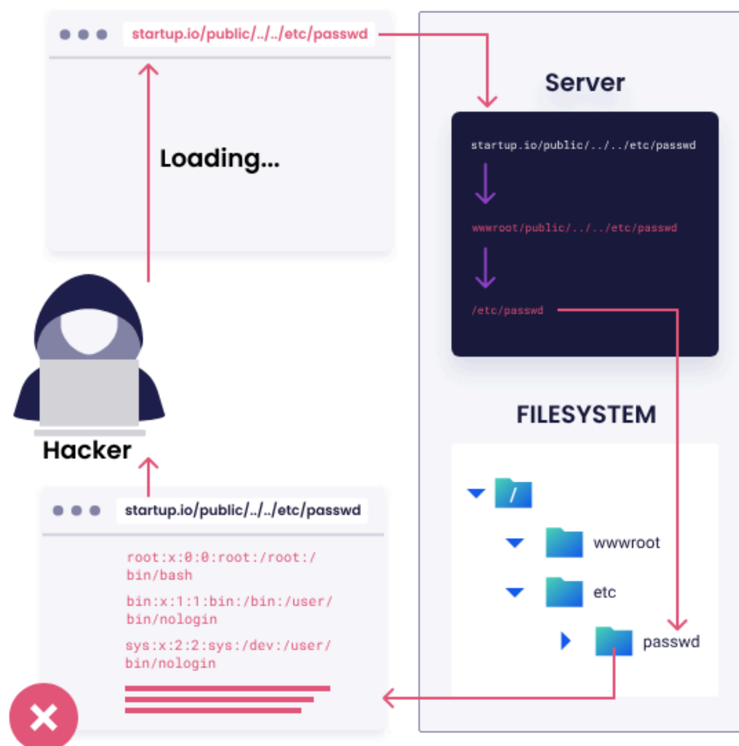
The current lab rejected traversal-style input:

sub/../link.txt → ValueError: contains '..' components
/tmp/lab/secret.txt → ValueError: absolute path not allowed

This shows the current version is applying lexical path checks against obvious traversal and absolute-path cases. That is an improvement over the vulnerable behavior documented in my older notes.

Few-shot loading also accepts normal relative file input

My examples.json test succeeded, showing that the same trust model applies to additional prompt-loading inputs, not only template_path. The loader still accepts relative files when they are syntactically allowed and present in the sandboxed test area.



Why this matters

A path traversal issue becomes dangerous when user-controlled data reaches a file-loading function. OWASP describes path traversal as using ../, absolute paths, or similar techniques to escape the intended directory and access files elsewhere on the filesystem. CWE-22 describes the same class as improper limitation of a pathname to a restricted directory.

That matters here because prompt loaders often sit in application paths where configuration can come from:

API input
database fields
uploaded files
orchestration metadata
agent/tool configuration

If an application passes untrusted path data into a prompt loader, the file contents can become application data, prompt data, or even secret material. That is the real risk: the file system becomes a data source controlled by attacker-influenced input.

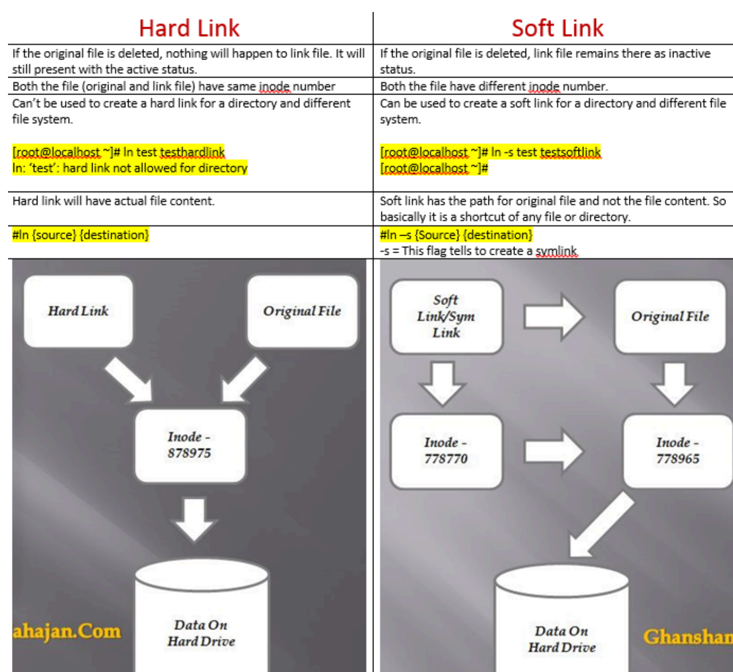
Root cause analysis

The underlying issue is a mismatch between:

what the application thinks it is reading, and
what the filesystem actually resolves

That mismatch is the classic symlink problem. A path that looks harmless may resolve to a different file after canonicalization. Python's documentation explicitly notes that Path.resolve() is the operation used to resolve symlinks and eliminate .. components, which is why canonical checks must be performed on the resolved path, not only the string provided by the caller.

My earlier notes show the dangerous version of this pattern: a trusted-looking relative file redirected through a symlink into a file that was not meant to be exposed. That is exactly the kind of "filesystem aliasing" risk that turns ordinary file reads into disclosure bugs.



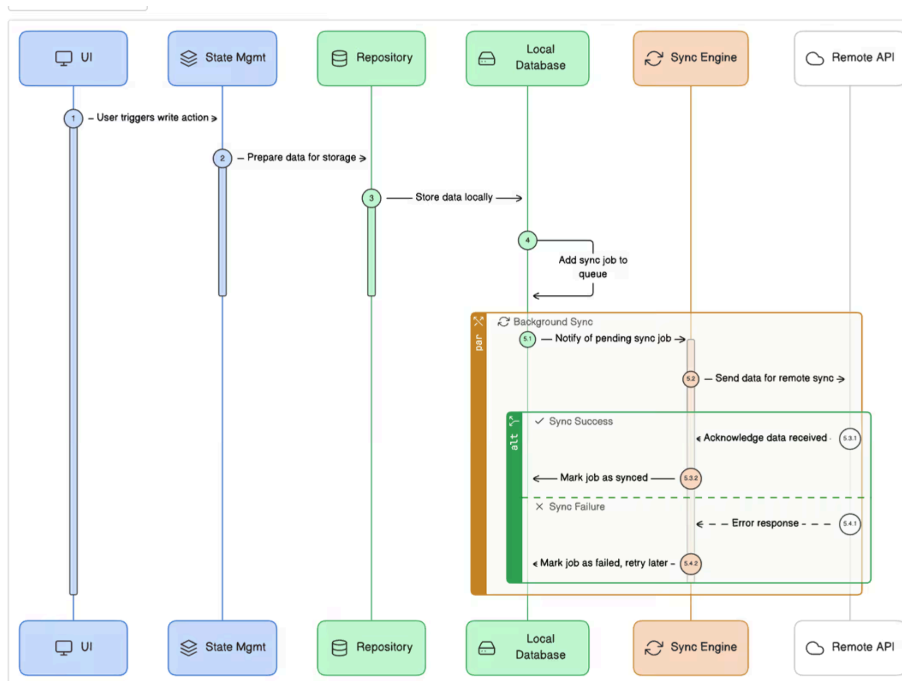
Hard Link	Soft Link
If the original file is deleted, nothing will happen to link file. It will still present with the active status.	If the original file is deleted, link file remains there as inactive status.
Both the file (original and link file) have same inode number	Both the file have different inode number.
Can't be used to create a hard link for a directory and different file system. [root@localhost ~]# ln test testhardlink ln: 'test': hard link not allowed for directory	Can be used to create a soft link for a directory and different file system. [root@localhost ~]# ln -s test testsoftlink [root@localhost ~]#
Hard link will have actual file content.	Soft link has the path for original file and not the file content. So basically it is a shortcut of any file or directory.
#ln {source} {destination}	#ln –s {Source} {destination} -s = This flag tells to create a symlink

Detailed findings

Finding 1: Untrusted file paths can be treated as prompt content sources

Severity: High in vulnerable versions, context-dependent in the current lab
Impact: Arbitrary file disclosure

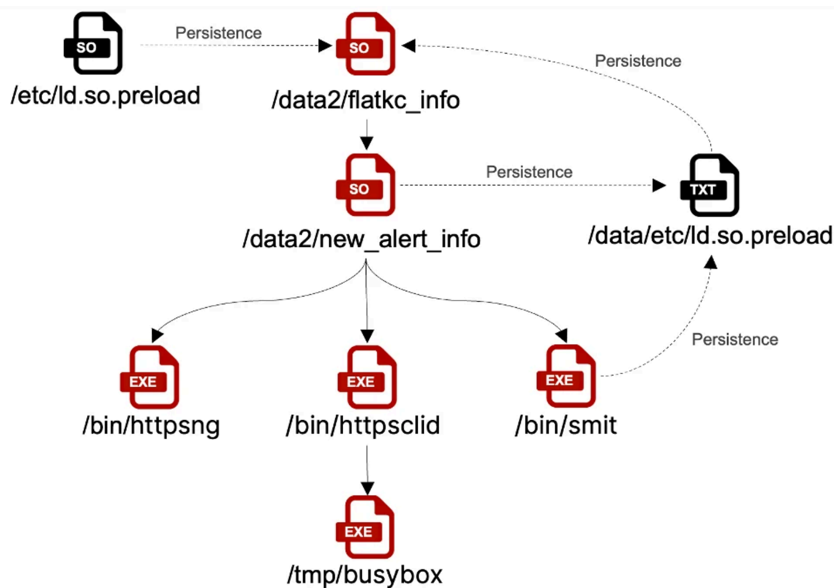
When the loader accepts a file path from configuration and reads it directly, the path becomes a trust boundary. If the caller can influence that path, the application may disclose file contents that were never meant to be exposed. My older test notes show exactly this pattern with the prompt loader reading attacker-influenced file targets.



Finding 2: Symlink redirection is the key bypass mechanism in the older behavior

Severity: High
Impact: File disclosure through pathname aliasing

My older lab notes show a symlink-based redirection from a trusted-looking path to another file, with the target contents successfully returned. That is the practical bypass that converts path-loading functionality into a disclosure primitive.



Finding 3: Traversal-style inputs are now rejected in the newer lab

Severity: Mitigated in current version
Impact: Obvious ../ and absolute-path cases are blocked

My current lab run shows a better posture: absolute paths and .. traversal are rejected. That is the right direction, and it means the old proof does not automatically carry over to the current package version.

Evidence summary

My lab evidence is strong because it covers multiple path shapes:

safe relative file: allowed
symlink to controlled file: allowed
traversal-style input: blocked
absolute path: blocked
few-shot examples file: allowed

That mix is useful because it separates expected behavior from dangerous behavior. It shows the loader is not blindly accepting everything, but it also shows how much trust is still placed in the filesystem inputs that are allowed.

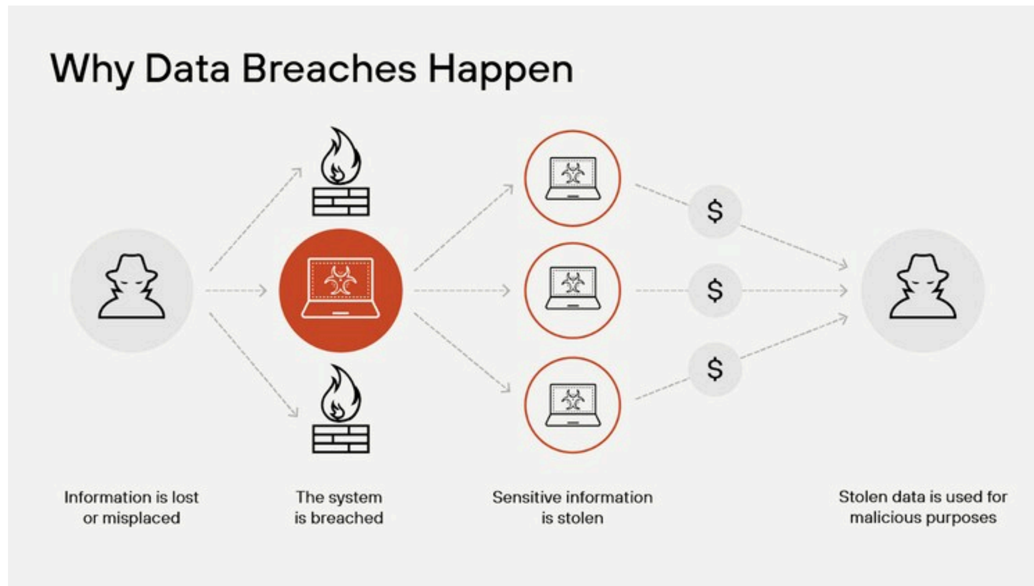
Impact assessment

If this issue exists in a real application where untrusted input can reach the loader, the likely consequences are:

disclosure of .env-style secrets
leakage of API keys or tokens
disclosure of application configuration

leakage of prompt material and internal instructions
possible cross-tenant data exposure in shared environments

Those are the realistic impacts that should be emphasized in a report. Claims about immediate root compromise are too speculative unless a separate, fully verified chain exists. A file-read bug is serious on its own; it does not need theatrical embellishment to be worth fixing.

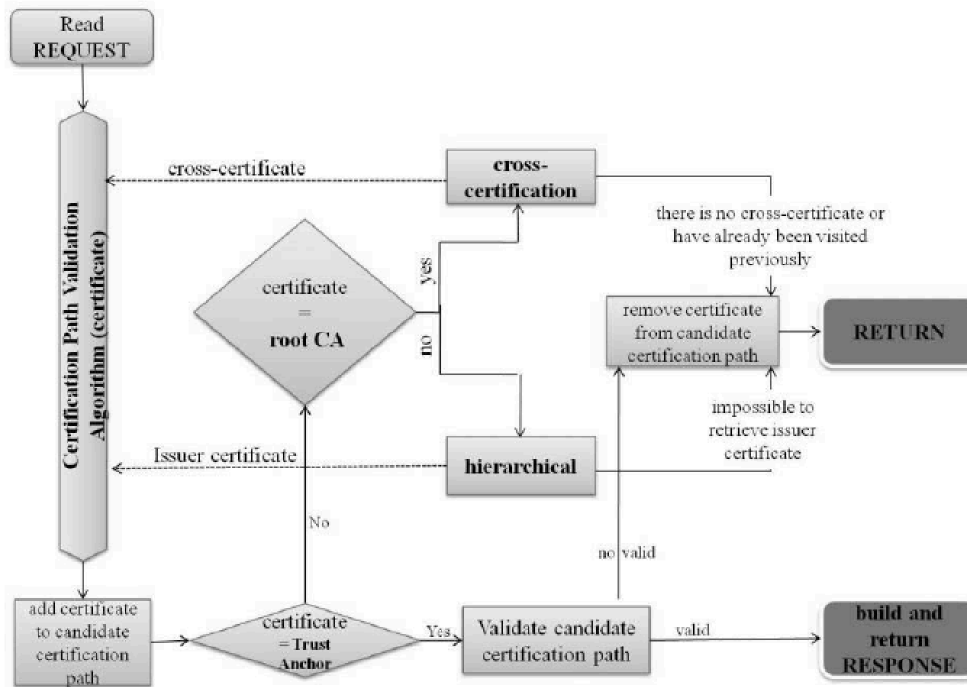


Remediation

The fix should be framed around canonical path enforcement and strict trust boundaries:

Resolve the path before opening it.
Enforce that the resolved path stays inside an explicit allowed directory.
Reject symlinks that escape the allowed root.
Avoid accepting raw filesystem paths from untrusted input where possible.
Prefer file IDs, allowlisted resource names, or in-memory prompt content instead of arbitrary file paths.

Python's Path.resolve() exists exactly to canonicalize paths and eliminate symlink and .. confusion, which is the right primitive to use before validating the result.


Read
REQUEST
Certification Path Validation
Algorithm (certificate)
cross-certificate
cross-
certification
there is no cross-certificate or
have already been visited
previously
certificate
=
root CA
yes
no
remove certificate
from candidate
certification path
RETURN
impossible to
retrieve issuer
certificate
Issuer certificate
hierarchical
No
no valid
add certificate
to candidate
certification
path
certificate
= Trust
Anchor
Yes
Validate candidate
certification path
valid
build and
return
RESPONSE