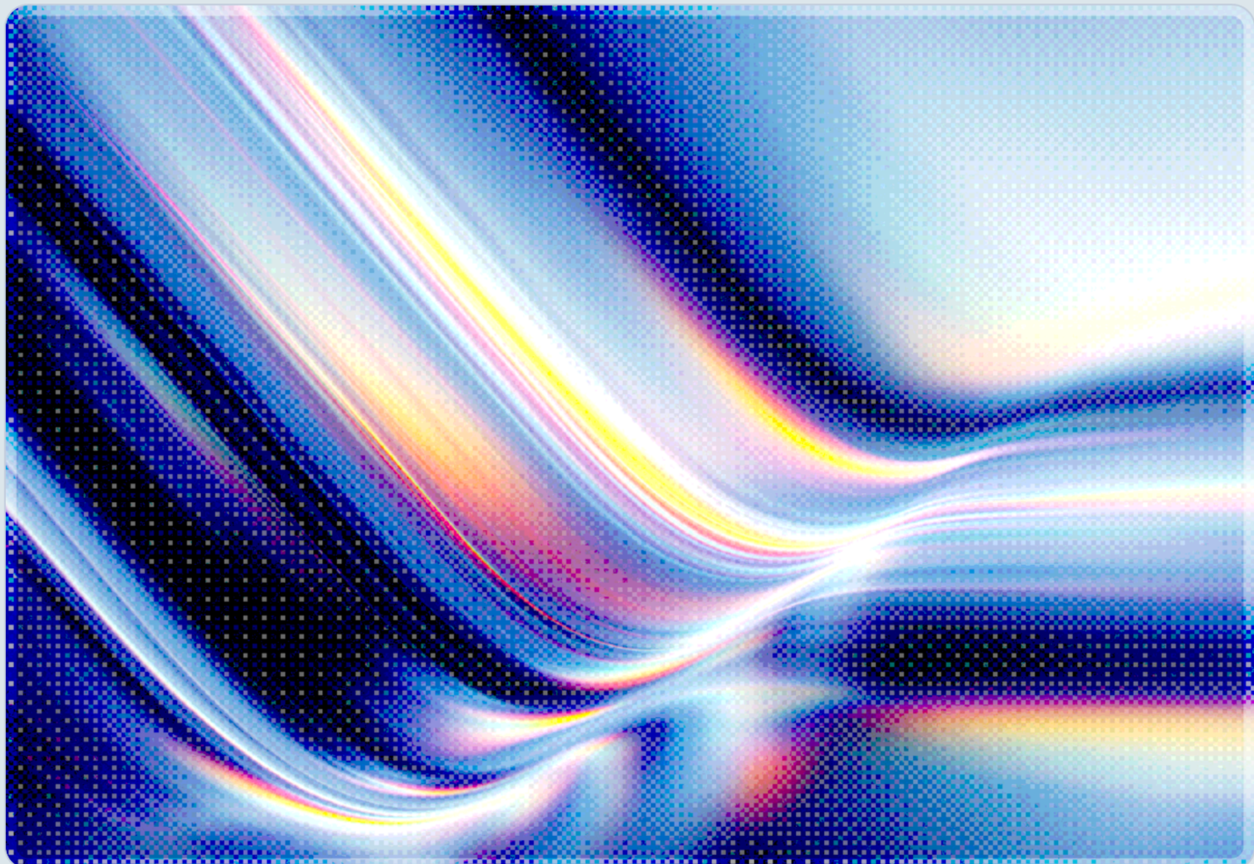
CSAI Foundation | Cloud Security Alliance

LangGraph RCE Chain: Checkpointer Flaw Enables Server Takeover

SQL Injection and Unsafe Deserialization in Self-Hosted AI Agent Deployments

2026-06-14



CSAI   CSA cloud security alliance®

© 2026 Cloud Security Alliance. Some rights reserved.

You may download, store, display, view, print, redistribute, and link to this document in its original, unmodified form, provided that attribution to the Cloud Security Alliance is maintained and all trademark and copyright notices remain intact.

This document may not be modified or altered. You may quote portions of the document as permitted by the Fair Use provisions of the United States Copyright Act, provided that attribution is given to the Cloud Security Alliance.

This document may be shared on professional and social media platforms in its original form with attribution.

This document was generated with AI assistance and has not undergone official CSA review and approval processes.

Key Takeaways

- Three now-patched CVEs in LangGraph—CVE-2025-67644, CVE-2026-28277, and CVE-2026-27022—create a chained attack path from SQL injection to full remote code execution on self-hosted AI agent servers, discovered by Yarden Porat of Check Point Research and disclosed in November 2025 [1][2]. The 2026 prefixes on two CVE identifiers reflect MITRE's formal assignment timeline rather than a separate discovery or disclosure event.
- The RCE chain exploits an unsanitized filter parameter in LangGraph's `get_state_history()` endpoint to inject malicious checkpoint rows into a SQLite database; when LangGraph loads the forged checkpoint, its unsafe msgpack deserialization handler executes arbitrary Python callables specified in the payload [1].
- Self-hosted deployments using SQLite or Redis checkpointers with user-controlled filter input are at risk; LangChain's managed LangSmith Deployment platform uses PostgreSQL and is not affected [2].
- Operators must update immediately: `langgraph-checkpoint-sqlite` to version 3.0.1 or later, `langgraph` to version 1.0.10 or later, and `@langchain/langgraph-checkpoint-redis` to version 1.0.2 or later [3][4].
- A successful exploit gives attackers runtime code execution on the agent server, directly exposing LLM API keys, conversation histories, credentials for connected data systems, and any network-accessible resource within the agent's operating environment [2].
- As of publication, no confirmed exploitation in the wild has been reported; a technical proof-of-concept is publicly available through Check Point Research's disclosure [1][6].

Background

LangGraph is an open-source framework developed by LangChain for building stateful, multi-actor AI agent applications. Its central innovation is the ability to model agent workflows as directed graphs with persistent state—enabling agents to pause mid-task, retrieve historical reasoning steps, branch across parallel execution paths, and coordinate with other agents. This architecture has made LangGraph one of the dominant orchestration frameworks in production AI systems, with approximately 46.5 million monthly downloads [2]; the package is publicly available on the Python Package Index [7]. Organizations

across financial services, healthcare, and enterprise software deploy LangGraph to build agents that interact with databases, internal APIs, code repositories, and customer data, frequently configured with access to sensitive systems depending on the agent's defined task scope.

The persistence mechanism that enables these capabilities is the checkpointer. When an agent executes a step in a LangGraph workflow, the checkpointer serializes the full graph state—including metadata, conversation context, tool call results, and agent configuration—and writes it to a backing store. On the next invocation, the framework retrieves the stored checkpoint and restores the agent's execution context, allowing stateful multi-session behavior without storing everything in memory. LangGraph supports multiple checkpointer backends, each suited to different deployment contexts: SQLite for lightweight local and development deployments, Redis for distributed caching scenarios, and PostgreSQL for production use cases requiring durability and concurrent access.

The choice of checkpointer backend has proven to carry substantial security implications. Check Point Research disclosed three vulnerabilities in November 2025 affecting LangGraph's SQLite and Redis checkpointer implementations. Two of the three can be chained to achieve remote code execution on any server running a vulnerable self-hosted LangGraph deployment where users can supply input to a state history retrieval endpoint. Patches were issued between December 2025 and March 2026 [1][3][4].

Security Analysis

The Three Vulnerabilities

CVE-2025-67644 (CVSS 7.3) is a SQL injection vulnerability in the SQLite checkpointer's internal `_metadata_predicate()` function, which constructs WHERE clauses to filter checkpoint lookups by metadata fields. Rather than using parameterized queries, the function interpolates user-controlled dictionary keys directly into SQL using Python f-string formatting—producing fragments such as `json_extract(CAST(metadata AS TEXT), '$.{query_key}') {operator}` without any validation or escaping. An attacker who controls the filter parameter can inject arbitrary SQL through a crafted metadata key. The `get_state_history()` method, which retrieves a history of checkpoints for a given thread, passes this filter parameter directly to the vulnerable function, making it the primary exposure point for network-accessible deployments. All versions of `langgraph-checkpoint-sqlite` before 3.0.1 are affected [1][3].

CVE-2026-28277 (CVSS 6.8) is an unsafe deserialization vulnerability in LangGraph's msgpack checkpoint decoder. When deserializing extension types from a checkpoint BLOB, the decoder reconstructs Python objects using a pattern equivalent to `getattr(importlib.import_module(module_name), callable_name)(arguments)`. This mechanism allows any importable Python callable to be invoked with attacker-supplied arguments—including `os.system`, `subprocess.Popen`, and other standard library functions suitable for executing shell commands. Exploiting this flaw in isolation requires write access to the checkpoint store. However, when chained with CVE-2025-67644, that prerequisite is eliminated: SQL injection creates the necessary write path without requiring direct database access. This vulnerability affects `langgraph` versions before 1.0.10 [1][4][5].

CVE-2026-27022 (CVSS 6.5) affects the Redis checkpointer implementation in `@langchain/langgraph-checkpoint-redis` [1][2][6]. Filter input passed to the Redis checkpoint listing method is incorporated into RediSearch queries without adequate escaping, enabling injection that can bypass access controls and return checkpoint data belonging to other users or sessions. While CVE-2026-27022 does not independently enable code execution, it creates unauthorized read access to checkpoint state and, in combination with the msgpack deserialization vulnerability, could support a similar exploit chain in Redis-backed deployments. Versions before 1.0.2 are affected [1][2].

The RCE Chain

The attack requires an application that exposes `get_state_history()` with a user-controlled `filter` parameter—a configuration that naturally arises in multi-tenant agent applications where users can query the history of their own agent threads. From that starting point, the attack proceeds in three phases [1].

In the first phase, the attacker crafts a malicious msgpack payload encoding a Python callable with a command argument—for example, invoking `os.system` with a reverse shell instruction or `subprocess.Popen` to stage a file exfiltration. This payload is structured as a well-formed msgpack extension object, the same format LangGraph uses to serialize real checkpoint state. In the second phase, the attacker embeds this payload inside a SQL injection string targeting the filter parameter of a `get_state_history()` call. The injected SQL instructs the SQLite engine to insert a fabricated row into the checkpoint table, with the malicious msgpack payload occupying the checkpoint BLOB column alongside otherwise valid-looking metadata.

In the third phase, LangGraph processes the returned rows from the query as though they were legitimate checkpoints. The msgpack decoder encounters the forged BLOB and, following its extension handler logic, imports the specified module and calls the specified function with the supplied arguments. The result is arbitrary code execution running under the process identity of the LangGraph server—before any application-level validation of the checkpoint data occurs in standard LangGraph deployments [1].

From that foothold, attackers gain access to everything the server process can reach: LLM API keys and provider credentials stored in environment variables (directly usable and billable), database connection strings for systems the agent queries, authentication tokens for integrated services, and the full content of conversation histories and reasoning traces stored in the checkpoint database. AI agent servers may hold credentials for multiple backend systems to support their tool-use capabilities; where credential scoping has not been enforced, compromising the agent runtime can expose access comparable to a privileged service account. The actual blast radius depends on the breadth of credentials present in the agent's environment and the network reach available to the compromised process [2].

Why Self-Hosted Deployments Are Distinctly Exposed

LangChain's managed platform—LangSmith Deployment, formerly LangGraph Platform—uses PostgreSQL rather than SQLite or Redis and is not affected by either vulnerability. This asymmetry is instructive: organizations that self-host LangGraph to satisfy data residency requirements, reduce cost, or support custom configurations accept a security responsibility that the managed service abstracts away. Development and prototype deployments using SQLite are likely to carry the greatest exposure, because SQLite requires no separate database service and is trivially easy to start without the network controls and access restrictions that production systems typically receive. Teams that prototype with SQLite and then deploy the same code with minimal hardening to internal-facing production environments may inadvertently ship the vulnerable configuration.

The filter parameter at the center of CVE-2025-67644 is architecturally natural—it exists to let applications retrieve checkpoint subsets based on application-defined metadata. In a framework without AI-specific considerations, a similar filtering endpoint in a web service would be an ordinary candidate for input validation and parameterized query review. In a self-hosted AI agent deployment, it is also an entry point into the agent's memory system, and the rows returned from that system are subsequently deserialized in a way that treats them as trusted code. The combination of a conventional injection class with an AI-specific execution path means that teams performing traditional web application security assessments may identify the injection risk but miss its full severity.

Persistence Mechanisms as High-Value Targets

Beyond the immediate RCE chain, this case illustrates a structural risk in AI agent architectures: the persistence mechanism is simultaneously essential to agent capability and potentially exploitable as a code execution channel. Because the checkpointer stores rich object graphs that must be faithfully reconstructed to restore agent state, the deserialization path is inherently expressive. Any deserialization format capable of reconstructing arbitrary Python objects from disk is a latent execution vector if an attacker can influence what gets deserialized.

The checkpointer is also a high-value target for integrity attacks that fall short of RCE. An attacker who can write to the checkpoint store—through CVE-2025-67644 or through a separate database compromise—can poison an agent's memory: altering prior reasoning traces, injecting false tool call results into historical context, or inserting persuasive content that the agent incorporates into future decisions. This threat to AI agent reasoning integrity—which persists even after organizations patch for RCE—merits dedicated attention in agentic threat models, where it does not yet receive systematic treatment.

Recommendations

Immediate Actions

Organizations running LangGraph in any configuration should audit installed versions of the three affected packages against the patched baselines. The required minimum versions are `langgraph-checkpoint-sqlite` 3.0.1 or later, `langgraph` 1.0.10 or later, and `@langchain/langgraph-checkpoint-redis` 1.0.2 or later. Since all three CVEs were disclosed simultaneously in November 2025 with coordinated patches completed by March 2026, any deployment that has not applied updates remains exposed to a publicly documented, chained exploit [1][3][4]. Version pinning in `requirements.txt` or `pyproject.toml` files should be reviewed to confirm that dependency resolution is not holding installations below the patched versions.

Any LangGraph application that exposes `get_state_history()` or equivalent state retrieval endpoints should be reviewed for user-controlled input reaching the filter parameter. Where such input exists, the library patch is the primary remediation; as a defense-in-depth measure, applications should validate and reject filter keys containing SQL metacharacters—quotes, comment sequences, semicolons, and parentheses—at the application boundary, independent of framework-level handling.

Short-Term Mitigations

Self-hosted deployments benefit substantially from network-layer controls that are straightforward to apply regardless of patch status. LangGraph servers should not be exposed as open endpoints: authentication should be enforced at a reverse proxy or API gateway before requests reach the application, and the `get_state_history()` endpoint should require authentication tokens scoped to specific users or tenants. The checkpointer's backing store—whether a SQLite file on disk or a Redis instance—should be treated as sensitive infrastructure. Filesystem permissions for SQLite database files should restrict write access to the agent process exclusively, and Redis instances should require authentication and bind only to localhost or a dedicated private interface.

For teams using SQLite in production, migration to the PostgreSQL checkpointer is the most structurally sound remediation. The SQLite backend is documented as appropriate for development and testing; the PostgreSQL implementation provides access control, audit logging, and process isolation that SQLite cannot offer. LangChain's own managed platform made this architectural choice, and organizations self-hosting LangGraph that handle sensitive data should align with it. LangGraph's official persistence documentation covers the available checkpointer backends and migration path [10].

LLM API keys and other credentials in the agent server environment should be evaluated for rotation following any window of confirmed or suspected exposure. Because LLM provider API keys grant billable, quota-consuming access to foundation model endpoints, their compromise has both security and direct financial consequences. Credential scoping should be reviewed to confirm that agents hold only the permissions their defined tasks require—overly broad credentials amplify the blast radius of any future compromise.

Strategic Considerations

LangGraph has released multiple versions per month throughout its history; security-relevant changes to serialization paths, query construction, and filter handling may not receive prominent placement in release notes. Security teams should integrate AI framework package updates into their patch management cadence and instrument systems to alert when AI framework versions fall below required baselines.

AI-specific threat modeling should be incorporated into the security review process for new agentic deployments. The attack surface of a self-hosted AI agent server extends beyond traditional web application concerns to include serialization paths, persistence mechanism trust boundaries, and credential concentration patterns that differ from conventional services. Standard SAST tools tuned for web injection patterns may identify the SQL injection surface while lacking rules for AI-specific

deserialization chains—a gap worth validating in any tool evaluation. Threat modeling that explicitly considers the agent's state management, memory system, and credential environment is necessary to capture the full exposure profile.

At a program level, organizations deploying AI agents that process sensitive data or hold credentials for production systems should establish incident response runbooks that specifically address AI agent server compromise. The combination of credential exposure, data access, and potential for agent memory poisoning creates a response scenario that differs from a standard web application breach and that may require coordination across data privacy, cloud security, and AI governance teams.

CSA Resource Alignment

The LangGraph vulnerability chain maps directly to threat categories in CSA's MAESTRO framework for agentic AI threat modeling. MAESTRO organizes threats across seven architectural layers of agentic systems and identifies the persistence and state management layer as a distinct zone of risk. The RCE chain disclosed here illustrates how exploitation of the trust relationship between an agent orchestrator and its backing store can achieve code execution or integrity compromise—an attack pattern practitioners can map to MAESTRO's persistence-layer threat categories to identify analogous injection surfaces in their own checkpointer configurations [8].

The AI Controls Matrix (AICM) provides specific control requirements applicable to this vulnerability. For organizations acting as Orchestrated Service Providers—deploying AI agent infrastructure for internal or external use—AICM's input handling controls cover the SQL injection vulnerability directly, requiring that all external input to AI system components be validated and sanitized before processing. AICM's supply chain and dependency integrity controls apply to the msgpack deserialization vulnerability, encompassing review of serialization libraries and deserialization paths within AI framework dependencies. The AICM shared responsibility framework also clarifies the division of security obligations between infrastructure operators and application developers for self-hosted deployments, helping teams identify gaps that the patching of library code alone does not address [9].

CSA's Cloud Controls Matrix (CCM) provides complementary guidance through its Application and Interface Security domain, which requires secure coding practices including injection prevention across all system interfaces. The Identity and Access Management domain in CCM is directly applicable to the credential exposure risk that follows RCE: AI agent identities should be provisioned following least-privilege principles, with scoped credentials, credential rotation procedures, and detection capabilities for anomalous API key usage.

Zero Trust principles are particularly valuable for self-hosted AI agent deployments. A Zero Trust architecture treats the agent runtime as an untrusted execution environment that must authenticate for every downstream resource access, enforces micro-segmentation to limit lateral movement following a compromise, and applies continuous verification rather than implicit trust based on network location. For organizations running LangGraph in internal environments behind perimeter controls, Zero Trust network policies on the agent server's outbound access can meaningfully constrain the blast radius of a successful RCE—limiting what an attacker can reach even after obtaining code execution.

© 2026 Cloud Security Alliance. AI-assisted Rapid Research.

References

[1] Porat, Yarden. "From SQLi to RCE: Exploiting LangGraph's Checkpointer." Check Point Research, 2026.

[2] Check Point Blog. "When Your AI Agent's Memory Becomes a Security Liability." Check Point, 2026.

[3] GitLab Advisory Database. "CVE-2025-67644: LangGraph's SQLite is vulnerable to SQL injection via a metadata filter key in SQLite checkpointer list method." GitLab, 2025.

[4] GitLab Advisory Database. "CVE-2026-28277: LangGraph checkpoint loading has unsafe msgpack deserialization." GitLab, 2026.

[5] SentinelOne. "CVE-2026-28277: LangGraph SQLite Checkpoint RCE Flaw." SentinelOne Vulnerability Database, 2026.

[6] The Hacker News. "LangGraph Flaw Chain Exposes Self-Hosted AI Agents to Remote Code Execution." The Hacker News, June 2026.

[7] LangChain. "langgraph." Python Package Index (PyPI), accessed June 2026.

[8] Huang, Ken and CSA AI Safety Working Group. "Agentic AI Threat Modeling Framework: MAESTRO." Cloud Security Alliance, February 2025.

[9] Cloud Security Alliance. "AI Controls Matrix (AICM)." Cloud Security Alliance, 2025.

[10] LangChain. "LangGraph Persistence." LangGraph Documentation, accessed June 2026.

© 2026 Cloud Security Alliance. AI-assisted Rapid Research.