# SQL-Injektion

### aus Wikipedia, der freien Enzyklopädie

**SQL-Injektion** (engl. *SQL Injection*) bezeichnet das Ausnutzen einer Sicherheitslücke in Zusammenhang mit SQL-Datenbanken, die durch mangelnde Maskierung oder Überprüfung von Metazeichen in Benutzereingaben entsteht. Der Angreifer versucht dabei, über die Anwendung, die den Zugriff auf die Datenbank bereitstellt, eigene Datenbankbefehle einzuschleusen. Sein Ziel ist es, Daten in seinem Sinne zu verändern oder Kontrolle über den Server zu erhalten.

## Inhaltsverzeichnis



## Auftreten

SQL-Injektionen sind dann möglich wenn Daten wie beispielsweise Benutzereingaben in den SQL-Interpreter gelangen. Denn Benutzereingaben können Zeichen enthalten, die für den SQL-Interpreter Sonderfunktion besitzen und so Einfluß von außen auf die ausgeführten Datenbankbefehle ermöglichen. Solche Metazeichen in SQL sind zum Beispiel der umgekehrte Schrägstrich, das Anführungszeichen, der Apostroph und das Semikolon.

Oft sind solche Lücken in CGI-Scripten und in Programmen zu finden, die Daten wie Webseiteninhalte oder E-Mails in SQL-Datenbanken eintragen. Nimmt ein solches Programm die Maskierung nicht korrekt vor, kann ein Angreifer durch den gezielten Einsatz von Funktionszeichen weitere SQL-Befehle einschleusen oder die Abfragen so manipulieren, dass zusätzliche Daten verändert oder ausgegeben werden. In einigen Fällen besteht auch die Möglichkeit, Zugriff auf eine Shell zu erhalten, was im Regelfall die Möglichkeit zur Kompromittierung des gesamten Servers bedeutet.

# Vorgang

## Veränderung von Daten

Auf einem Webserver befindet sich das Script find.cgi zum Anzeigen von Artikeln. Das Script akzeptiert den Parameter „ID“, welcher später Bestandteil der SQL-Abfrage wird. Folgende Tabelle soll dies illustrieren:

| | Erwarteter Aufruf |
|---|---|
| **Aufruf** | `http://webserver/cgi-bin/find.cgi?ID=42` |
| **Erzeugtes SQL** | `SELECT author, subjekt, text FROM artikel WHERE ID=42` |
| | **SQL-Injektion** |
| **Aufruf** | `http://webserver/cgi-bin/find.cgi?ID=42;UPDATE+USER+SET+TYPE="admin"+WHERE+ID=23` |
| **Erzeugtes SQL** | `SELECT author, subjekt, text FROM artikel WHERE ID=42; UPDATE USER SET TYPE="admin" WHERE ID=23` |

Dem Programm wird also ein zweiter SQL-Befehl untergeschoben, der die Benutzertabelle modifiziert.

## Datenbank-Server verändern

Auf einem Webserver findet sich das Script search.aspx zum Suchen nach Webseiten. Das Script akzeptiert den Parameter „keyword“, welcher später Bestandteil des SQL-Abfrage wird. Folgende Tabelle soll dies illustrieren:

| | Erwarteter Aufruf |
|---|---|
| **Aufruf** | `http://webserver/search.aspx?keyword=sql` |
| **Erzeugtes SQL** | `SELECT url, title FROM myindex WHERE keyword LIKE '%sql%'` |
| | **SQL-Injektion** |
| **Aufruf** | `http://webserver/search.aspx?keyword=sql'+;GO+EXEC+cmdshell('format+C')+--` |
| **Erzeugtes SQL** | `SELECT url, title FROM myindex WHERE keyword LIKE '%sql' ;GO EXEC cmdshell('format C') --%'` |

Hier wird der eigentlichen Abfrage ein weiterer Befehl angehängt. Die zwei Bindestriche (`--`) kommentieren das Hochkomma als Überbleibsel der eigentlichen Anfrage aus, womit es ignoriert wird. Die nun generierte Abfrage ermöglicht so das Formatieren der Festplatte. Aber auch Downloads oder Ähnliches lassen sich dadurch erzeugen (am Beispiel Microsoft SQL Server).

## Ausspähen von Daten

Auf manchen SQL-Implementationen ist die `UNION`-Klausel verfügbar. Diese erlaubt es, mehrere `SELECT`s gleichzeitig abzusetzen, die dann eine gemeinsame Ergebnismenge zurückliefern. Durch eine geschickt untergeschobene `UNION`-Klausel können beliebige Tabellen und Systemvariablen ausgespäht werden.

| | Erwarteter Aufruf |
|---|---|

| | |
|---|---|
| **Aufruf** | `http://webserver/cgi-bin/find.cgi?ID=42` |
| **Erzeugtes SQL** | `SELECT author, subjekt, text FROM artikel WHERE ID=42` |

**SQL-Injektion**

| | |
|---|---|
| **Aufruf** | `http://webserver/cgi-bin/find.cgi?ID=42+UNION+SELECT+login,+password,+'x'+FROM+user` |
| **Erzeugtes SQL** | `SELECT author, subjekt, text FROM artikel WHERE ID=42 UNION SELECT login, password, 'x' FROM user` |

Das „x“ beim `UNION SELECT` ist nötig, weil alle mit `UNION` verknüpften `SELECT`s die gleiche Anzahl von Spalten haben müssen. Der Angreifer muss also wissen, wie viele Spalten die ursprüngliche Abfrage hat.

Ist der Datenbankserver fehlerhaft konfiguriert und hat beispielsweise ein aktuell mit der Datenbank verbundener Benutzer, über den die SQL-Injektion abgesetzt werden soll, Zugriff auf Systemdatenbanken, so kann der Angreifer über eine einfache SQL-Syntax wie `Systemdatenbank.SystemtabelleMitTabellenAuflistung` auf die Systemtabellen zugreifen und sämtliche Tabellen einer bestimmten Datenbank auslesen. Dadurch kann er wichtige Informationen erhalten, um weitere Angriffe durchzuführen und tiefer in das System einzudringen.

## Einschleusen von beliebigem Code

Eine weniger bekannte Variante stellt gleichzeitig die potenziell gefährlichste dar. Wenn der Datenbankserver die Kommandos `SELECT … INTO OUTFILE` beziehungsweise `SELECT … INTO DUMPFILE` unterstützt, können diese Kommandos dazu benutzt werden, Dateien auf dem Dateisystem des Datenbankserver abzulegen. Theoretisch ist es dadurch möglich, falls das Bibliotheksverzeichnis des Betriebssystems oder des Datenbankservers für denselben beschreibbar ist (wenn dieser zum Beispiel als root läuft), einen beliebigen Code auf dem System auszuführen.

## Zeitbasierte Angriffe

Wenn der Datenbankserver Benchmark-Funktionen unterstützt, kann der Angreifer diese dazu nutzen, um Informationen über die Datenbankstruktur in Erfahrung zu bringen. In Verbindung mit dem if-Konstrukt sind der Kreativität des Angreifers kaum Grenzen gesetzt.

Das folgende Beispiel benötigt auf einem MySQL-Datenbankserver mehrere Sekunden, falls der gegenwärtige User root ist:

```
SELECT IF( USER() LIKE 'root@%', BENCHMARK(100000,SHA1('test')), 'false');
```

## Erlangen von Administratorrechten

Bei bestimmten Datenbankservern, wie dem Microsoft SQL Server, werden *Stored Procedures* mitgeliefert, die unter anderem dazu missbraucht werden können, einen neuen Benutzer auf dem angegriffenen System anzulegen.

Diese Möglichkeit kann dazu benutzt werden, um zum Beispiel eine Shell auf dem angegriffenen Rechner zu starten.

## Blinde SQL-Injektion

Von einer *blinden SQL-Injektion* wird gesprochen, wenn ein Server keine deskriptive Fehlermeldung zurückliefert, aus der hervorgeht, ob der übergebene Query erfolgreich ausgeführt wurde oder nicht. Anhand verschiedenster Kleinigkeiten wie etwa leicht unterschiedlicher Fehlermeldungen oder charakteristisch unterschiedlicher Antwortzeiten des Servers kann ein versierter Angreifer häufig dennoch feststellen, ob ein Query erfolgreich war oder einen Fehler zurückmeldet.

# Gegenmaßnahmen

Eine Maßnahme besteht darin, Daten die Bedeutung für den SQL-Interpreter zu nehmen. Zu diesem Zweck wird oft zum Ausfiltern oder Maskieren (sogenanntem Escapen) von Metazeichen in Benutzereingaben gegriffen. Hierdurch kann die Gefahr der Injektion gemildert oder beseitigt werden.

Generell ist die Webanwendung für die korrekte Prüfung der Eingabedaten verantwortlich, so dass vor allem die Metazeichen des betreffenden Datenbanksystems entsprechend zu maskieren sind, die für Ausnutzung dieser Sicherheitslücke verantwortlich sind. Weitergehend können auch die Eingabedaten auf die Eigenschaften erwarteten Werte geprüft werden. So bestehen deutsche Postleitzahlen beispielsweise nur aus Ziffern. Geeignete Schutzmaßnahmen sind in erster Linie dort zu implementieren.

Der simplere und sicherere Weg ist jedoch, die Daten überhaupt vom Interpreter fernzuhalten. Dabei lässt sich auch ohne Verstümmelung der Eingabe auskommen. Die Technik dazu sind gebundene Parameter Prepared Statements. Dabei werden die Daten als Parameter an einen bereits kompilierten Befehl übergeben. Die Daten werden somit nicht interpretiert und eine Injektion verhindert. Stored Procedures bieten dagen keinen generellen Schutz vor SQL-Injection, insbesonder dann nicht, wenn der SQL-Code der Funktion nicht bekannt ist.

Doch auch auf Seiten des Datenbankservers lassen sich Sicherheitsvorkehrungen treffen. So sollten die Benutzer, mit denen sich eine Webanwendung beim Datenbankserver authentifiziert, nur die Privilegien besitzen, die er tatsächlich benötigt. So können zumindest einige der möglichen Angriffe unwirksam werden.

Hat ein Betreiber eines Webservers keine Kontrolle über die Anwendungen kann durch Einsatz von Web Application Firewalls (WAF) zumindest teilweise verhindert werden, dass SQL-Injektion-Schwachstellen ausgenutzt werden können.

Es ist nicht schwer, bestehende Programme so umzubauen, dass SQL-Injektionen nicht mehr möglich sind. Das hauptsächliche Problem der meisten Programmierer ist fehlendes Wissen über diese Art von Angriffen. Nachfolgend einige Beispiele, um die Angriffe abzuwehren.

## Visual Basic (ADOdb)

In Visual Basic gibt es einfache *Command*-Objekte, mit denen diese Probleme vermieden werden können.

Anstatt

```
cn.Execute "SELECT spalte1 FROM tabelle WHERE spalte2 = '"
          & spalte2Wert & "'"
```

sollte Folgendes verwendet werden:

```
Dim cmd As ADODB.Command, rs as ADODB.Recordset
With cmd
  Set .ActiveConnection = cn
  Set .CommandType = adCmdText
  Set .CommandString = "SELECT spalte1 FROM tabelle WHERE spalte2=paramSp2"
  .Parameters.Append .CreateParameter("paramSp2", adVarChar, adParamInput, 25, spalte2Wert) '25 ist die max. län
  Set rs = .Execute
End With
```

## Microsoft .NET Framework (ADO.NET)

Im .NET Framework gibt es einfache Objekte, mit denen solche Probleme umgangen werden können.

Anstatt

```
SqlCommand cmd = new SqlCommand("SELECT spalte1 FROM tabelle WHERE spalte2 = '"
                + spalte2Wert + "';");
```

sollte Folgendes verwendet werden:

```
string spalte2Wert = "Mein Wert";
SqlCommand cmd = new SqlCommand("SELECT spalte1 FROM tabelle WHERE spalte2 = @spalte2Wert;");
cmd.Parameters.AddWithValue("spalte2Wert", spalte2Wert);
```

## Java (JDBC)

Eine SQL-Injektion kann leicht durch bereits vorhandene Funktion verhindert werden. In Java wird zu diesem Zweck die PreparedStatement-Klasse verwendet (JDBC-Technologie).

Anstatt

```
Statement stmt = con.createStatement();
ResultSet rset = stmt.executeQuery("SELECT spalte1 FROM tabelle WHERE spalte2 = '"
                + spalte2Wert + "';");
```

sollte Folgendes verwendet werden:

```
PreparedStatement pstmt = con.prepareStatement("SELECT spalte1 FROM tabelle WHERE spalte2 = ?");
pstmt.setString(1, spalte2Wert);
ResultSet rset = pstmt.executeQuery();
```

Der Mehraufwand an Schreibarbeit durch die Verwendung der PreparedStatement-Klasse zahlt sich außerdem durch einen Performancegewinn aus, da durch die Angaben bereits im Voraus eine Optimierung durchgeführt werden kann.

## PHP

In PHP steht zu diesem Zweck die Funktion mysql_real_escape_string()[1] zur Verfügung, die jedoch lediglich für eine MySQL-Verbindung benutzbar ist. Wird eine andere Datenbank benutzt, so ist diese Funktion nicht geeignet. Doch PHP hält für fast jede Datenbank eine solche Escape-Funktion bereit. Die PHP-Oracle-Funktionen besitzen jedoch beispielsweise keine Escape-Funktion, hingegen können

dort Prepared Statements verwendet werden, was bei der beliebten MySQL-Datenbank erst mit den Funktionen von MySQLi[2] möglich geworden ist.

Anstatt

```
$abfrage = "SELECT spalte1 FROM tabelle WHERE spalte2 = '".$_POST['spalte2Wert']."'";
$query = mysql_query($abfrage) or die("Datenbankabfrage ist fehlgeschlagen!");
```

sollte Folgendes verwendet werden:

```
$abfrage = "SELECT spalte1 FROM tabelle WHERE
            spalte2 = '".mysql_real_escape_string($_POST['spalte2Wert'])."'";
$query = mysql_query($abfrage) or die("Datenbankabfrage ist fehlgeschlagen!");
```

Ab PHP 5.1 sollten PHP Data Objects für Datenbankabfragen verwendet werden.

```
$dbh->exec("INSERT INTO REGISTRY (name, value)
            VALUES (".$dbh->quote($name,PDO::PARAM_STR).", ".$dbh->quote($value,PDO::PARAM_INT).")");
```

Oder als Prepared Statement:

```
$stmt = $dbh->prepare("INSERT INTO REGISTRY (name, value) VALUES (:name, :value)");
$stmt->bindParam(':name', $name);
$stmt->bindParam(':value', $value);
```

Häufig wird aus Bequemlichkeit einfach die Konfigurationsoption „magic_quotes_gpc“ auf „on“ gestellt, mit der von außen kommende Benutzereingaben automatisch maskiert werden. Doch dies ist nicht empfehlenswert. Denn manche nicht selber programmierte Scripte setzen eigenständig Funktionen wie etwa addslashes()[3] oder das bereits weiter oben genannte mysql_real_escape_string()[1] ein. D. h. dass bereits allen relevanten Zeichen in den Benutzereingaben durch Magic Quotes[4] ein Backslash vorangestellt wurde und nun durch die Escape-Funktion erneut ein Backslash vorangestellt wird. Somit werden die Benutzereingaben verfälscht, und man erhält anstatt eines einfachen Anführungszeichens ein Anführungszeichen mit vorangestelltem Backslash (\"). Auch aus Gründen der Portabilität sollte bei der Entwicklung von Anwendungen auf diese Einstellung verzichtet und stattdessen alle Eingaben manuell validiert und maskiert werden, da nicht davon ausgegangen werden kann, dass auf allen Systemen dieselben Einstellungen vorherrschen oder möglich sind.

## Perl

Das datenbankunabhängige Datenbankmodul DBI unterstützt eine „prepare“-Syntax ähnlich der aus dem Java-Beispiel.

```
$statementhandle = $databasehandle->prepare("SELECT spalte1 FROM tabelle WHERE spalte2 = ?");
$returnvalue = $statementhandle->execute( $spalte2Wert );
```

## ColdFusion

Unter ColdFusion kann das <cfqueryparam>-Tag verwendet werden, welches sämtliche notwendigen

Validierungen übernimmt[5]):

```
SELECT *
   FROM courses
   WHERE Course_ID = <cfqueryparam value = "#Course_ID#" CFSQLType = "CF_SQL_INTEGER">
```

### MS-SQL

Über parametrisierte Kommandos kann die Datenbank vor SQL-Injektionen geschützt werden:

```
SELECT COUNT(*) FROM Users WHERE UserName=? AND UserPassword=?
```

## Siehe auch

- Cross-Site Scripting

## Weblinks

- *SQL Injection – Are your web applications vulnerable?*, Kevin Spett, SPI Labs (http://www.spidynamics.com/support/whitepapers/WhitepaperSQLInjection.pdf) (englisch, PDF)
- Abusing Poor Programming Techniques in Webserver Scripts via SQL Injection (http://www.derkeiler.com/Mailing-Lists/securityfocus/secprog/2001-07/0001.html) (englisch)
- *Advanced SQL Injection In SQL Server Applications*, Chris Anley, NGSSoftware Insight Security Research (http://www.nextgenss.com/papers/advanced_sql_injection.pdf) (englisch, PDF)
- Bundesamt für Sicherheit in der Informationstechnik (BSI): Maßnahmenkatalog und Best Practices für die Sicherheit von Webanwendungen (http://www.bsi.de/literat/studien/websec/WebSec.pdf) (PDF)
- MS Access SQL Injection Cheat Sheet (http://www.webapptest.org/ms-access-sql-injection-cheat-sheet-EN.html) (englisch)
- Web Security Threat Classification (http://www.webappsec.org/projects/threat/v1/WASC-TC-v1_0.de.pdf) (PDF, 432kb)

## Einzelnachweise und Ressourcen

1. ↑ *a b* PHP Manual: mysql_real_escape_string (http://www.php.net/manual/de/function.mysql-real-escape-string.php)
2. ↑ PHP Manual: Verbesserte MySQL-Erweiterung (MySQLi) (http://www.php.net/manual/de/ref.mysqli.php)
3. ↑ PHP Manual: addslahes (http://www.php.net/manual/de/function.addslashes.php)
4. ↑ PHP Manual: Magic Quotes (http://www.php.net/manual/de/security.magicquotes.php)
5. ↑ ColdFusion Online Hilfe (http://livedocs.macromedia.com/coldfusion/7/htmldocs/00000317.htm#1102474)

---

Dieser Artikel wurde in die Liste der lesenswerten Artikel aufgenommen.