SQL-Injection

aus Wikipedia, der freien Enzyklopädie

SQL-Injection (dt. *SQL-Einschleusung*) 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 auszuspähen, in seinem Sinne zu verändern, die Kontrolle über den Server zu erhalten oder einfach größtmöglichen Schaden anzurichten.

Inhaltsverzeichnis

Auftreten

SQL-Injections 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 (Backslash), das Anführungszeichen, der Apostroph und das Semikolon.

Oft sind solche Lücken in CGI-Skripten 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 der 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, subject, text FROM artikel WHERE ID=42;`
	SQL-Injection
Aufruf	`http://webserver/cgi-bin/find.cgi?ID=42;UPDATE+USER+SET+TYPE="admin"+WHERE+ID=23`
Erzeugtes SQL	`SELECT author, subject, 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“, dessen Wert später Bestandteil der 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-Injection
Aufruf	`http://webserver/search.aspx?keyword=sql'+;GO+EXEC+cmdshell('shutdown+/s')+--`
Erzeugtes SQL	`SELECT url, title FROM myindex WHERE keyword LIKE '%sql' ;GO EXEC cmdshell('shutdown /s') --%'`

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 das Ausführen eines Windows-Prozesses, hier illustriert durch das erzwungene Herunterfahren des Servers (sofern der Prozess Administratorrechte hat). Aber auch Daten 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, subject, text FROM artikel WHERE ID=42;`
	SQL-Injection
Aufruf	`http://webserver/cgi-bin/find.cgi?ID=42+UNION+SELECT+login,+password,+'x'+FROM+user`
Erzeugtes SQL	`SELECT author, subject, 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 kann die Anzahl der Spalten herausfinden, indem er `ID=42 order by x--` anhängt. Wenn die Seite beispielsweise bei x = 8 normal lädt, aber bei x = 9 eine Fehlermeldung oder andere Seite zeigt, dann ist die Spaltenanzahl 8.

Ist der Datenbankserver fehlerhaft konfiguriert und hat beispielsweise ein aktuell mit der Datenbank verbundener Benutzer, über den die SQL-Injection 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.

Bei MySQL-Datenbanksystemen werden diese Systeminformationen in der Datenbank `information_schema`[1] verwaltet. Das nachfolgende Beispiel zeigt, wie bei einer Abfrage mit 3 Ergebnisspalten die Struktur sämtlicher zugreifbarer Datenbanken in Erfahrung gebracht werden kann.

	Erwarteter Aufruf
Aufruf	`http://webserver/cgi-bin/find.cgi?ID=42`
Erzeugtes SQL	`SELECT author, subject, text FROM artikel WHERE ID=42;`
	SQL-Injection
Aufruf	`http://webserver/cgi-bin/find.cgi?ID=42+UNION+SELECT+'Datenbank',+'Tabelle',+'Spalte'+ UNION+SELECT+table_schema,+table_name,+column_name+FROM+information_schema.columns+ WHERE+NOT+table_schema='information_schema';#%20`
Erzeugtes SQL	`SELECT author, subject, text FROM artikel WHERE ID=42 UNION SELECT 'Datenbank', 'Tabelle', 'Spalte' UNION SELECT table_schema, table_name, column_name FROM information_schema.columns WHERE NOT table_schema='information_schema';# ;`

Einige Datenbanksysteme bieten weiterhin die Möglichkeit, Dateien über eine Anfrage zurückzugeben. Hierüber können in Kombination mit oben genannten Techniken und soweit der Pfad bekannt ist, beliebige Dateien, auf die der Datenbankprozess Zugriff hat, ausgelesen werden.

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 bis zur Version 2000, wurden *Stored Procedures* wie Xp_cmdshell automatisch angeboten, die unter anderem dazu missbraucht werden können, Kommandos mit den Rechten des SQL-Serverprogramms auszuführen. Neuere Versionen des Microsoft SQL Server haben diese Funktion standardmäßig deaktiviert. Diese Möglichkeit konnte dazu benutzt werden, um zum Beispiel eine Shell auf dem angegriffenen Rechner zu starten.

Verwundbarkeiten innerhalb des Datenbankservers

Manchmal existieren Verwundbarkeiten auch innerhalb der Datenbanksoftware selbst. So erlaubte zum Beispiel die PHP-Funktion mysql_real_escape_string() im MySQL Server einem Angreifer, SQL Injection-basierende Angriffe basierend auf Unicode-Zeichen selbst dann auszuführen, wenn die Benutzereingaben korrekt maskiert wurden. Dieser Fehler wurde in der Version 5.0.22 am 24. Mai 2006 behoben.

Blinde SQL-Injection

Von einer *blinden SQL-Injection* 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

Um SQL-Injections zu verhindern, dürfen vom Benutzer eingegebene Daten nicht ohne Weiteres in eine SQL-Anweisung eingebaut werden. Durch das Escapen der Eingabe werden Metazeichen wie z. B. Anführungszeichen maskiert und somit vom SQL-Interpreter nicht beachtet.

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 erwarteter 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 SQL-Interpreter fernzuhalten. Dabei kann man auf das Kappen der Eingabe verzichten. Die Technik dazu sind gebundene Parameter in Prepared Statements. Dabei werden die Daten als Parameter an einen bereits kompilierten Befehl übergeben. Die Daten werden somit nicht interpretiert und eine SQL-Injection verhindert. Als positiven Nebeneffekt bekommt man bei bestimmten Datenbanken (z. B. Oracle) außerdem eine Steigerung der Performance. Stored Procedures bieten dagegen keinen generellen Schutz vor SQL-Injection, insbesondere 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-Injection-Schwachstellen ausgenutzt werden können. Unabhängig von der Kontrolle über die Anwendungen kann ein Betreiber eines Webservers durch den gezielten Einsatz einer WAF die Sicherheit zusätzlich erhöhen, da viele WAFs neben abwehrenden auch prophylaktische Maßnahmen anbieten.

Es ist nicht schwer, bestehende Programme so umzubauen, dass SQL-Injections 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
  .CommandText = "SELECT spalte1 FROM tabelle WHERE spalte2 = ?"
  .Parameters.Append .CreateParameter("paramSp2", adVarChar, adParamInput, 25, spalte2Wert) '25 ist die max. länge
  Set rs = .Execute
End With
```

Microsoft .NET Framework - C# (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-Injection kann leicht durch eine bereits vorhandene Funktion verhindert werden. In Java wird zu diesem Zweck die PreparedStatement-Klasse verwendet (JDBC-Technologie) und die Daten unsicherer Herkunft werden als getrennte Parameter übergeben. Um die Daten von der SQL-Anweisung zu trennen wird der Platzhalter „?“ verwendet.

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 kann sich außerdem durch einen Performancegewinn auszahlen, wenn das Programm das PreparedStatement-Objekt mehrfach verwendet.

PHP

In PHP steht zu diesem Zweck für fast jede verwendete Datenbank eine Escape-Funktion bereit. Nur die Oracle-Anbindung besitzt keine solche Escape-Funktion, hingegen können dort Prepared Statements verwendet werden, was beispielsweise bei der MySQL-Datenbank erst mit den Funktionen von MySQLi[2] möglich geworden ist.

Ein Beispiel für MySQL: 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!");
```

Eine weitere Möglichkeit ist die Typumwandlung von Übergabeparametern.

```
$abfrage = "SELECT spalte1
            FROM tabelle
            WHERE spalte2 = " . intval($_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 Skripte setzen eigenständig Funktionen wie etwa `addslashes()`[3] oder das bereits weiter oben genannte `mysql_real_escape_string()`[4] ein. Das heißt, dass bereits allen relevanten Zeichen in den Benutzereingaben durch so genannte Magic Quotes[5] ein Backslash vorangestellt wurde und nun durch die Escape-Funktion erneut ein Backslash vorangestellt wird. Somit werden die Benutzereingaben verfälscht und man erhält beispielsweise 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. Darüber hinaus sollte `addSlashes()` nicht zum Maskieren von Datenbank-Eingaben benutzt werden, da es keine ausreichende Sicherheit gegenüber `mysql_real_escape_string()` gewährleistet.[6]

Perl

Mit dem datenbankunabhängigen Datenbankmodul DBI, welches normalerweise in Perl verwendet wird:

Anstatt

```
$arrayref = $databasehandle->selectall_arrayref("SELECT spalte1 FROM tabelle WHERE spalte2 = $spalte2Wert");
```

sollte Folgendes verwendet werden:

```
$arrayref = $databasehandle->selectall_arrayref('SELECT spalte1 FROM tabelle WHERE spalte2 = ?',{},$spalte2Wert);
```

Perls DBI-Modul unterstützt außerdem eine „prepare“-Syntax ähnlich der aus dem Java-Beispiel.

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

Alternativ können über das Datenbankhandle auch Eingabe-Werte sicher maskiert werden, dabei achtet der DB-Treiber auf die für diese Datenbank typischen Sonderzeichen, der Programmierer muss keine tiefergehenden Kenntnisse darüber haben.

```
$arrayref = $databasehandle->selectall_arrayref("SELECT spalte1 FROM tabelle WHERE spalte2 = " .
                        $databasehandle->quote($spalte2Wert) );
```

Im sogenannten "tainted mode" verwendet Perl starke Heuristiken, um nur sichere Zugriffe zu erlauben. Zeichenketten, die vom Benutzer übergebene Parameter enthalten werden zunächst als "unsicher" behandelt, bis die Daten explizit validiert wurden, und dürfen vorher nicht in unsicheren Befehlen verwendet werden.

ColdFusion Markup Language

Unter ColdFusion kann das <cfqueryparam>-Tag verwendet werden, welches sämtliche notwendigen Validierungen übernimmt:[7]

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

MS-SQL

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

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

Siehe auch

- Sicherheit von Webanwendungen

Weblinks

- SQL-Injections - Analyse (http://www.erich-kachel.de/?p=223)
- Abusing Poor Programming Techniques in Webserver Scripts via SQL Injection (http://www.derkeiler.com/Mailing-Lists/securityfocus/secprog/2001-07/0001.html) (englisch)
- Chris Anley: *Advanced SQL Injection In SQL Server Applications* (http://www.nextgenss.com/papers/advanced_sql_injection.pdf) (PDF; 291 kB) NGSSoftware Insight Security Research (englisch)
- Maßnahmenkatalog und Best Practices für die Sicherheit von Webanwendungen (https://www.bsi.bund.de/cae/servlet/contentblob/476464/publicationFile/30642/WebSec_pdf.pdf) (PDF; 884 kB) Bundesamt für Sicherheit in der Informationstechnik (BSI)
- Web Security Threat Classification (http://www.webappsec.org/projects/threat/v1/WASC-TC-v1_0.de.pdf) (PDF, 432 kB)
- Steve Friedl: *SQL Injection Attacks by Example*. (http://unixwiz.net/techtips/sql-injection.html)
- Advanced SQL Injection in MySQL (http://www.alirecaiyekta.com/uploads/Advanced-SQL-Injection-in-MySQL-GERMAN.pdf) (PDF; 935 kB)

Einzelnachweise und Ressourcen

1. Die Datenbank `INFORMATION_SCHEMA`. (http://dev.mysql.com/doc/refman/5.1/en/information-schema.html) MySQL 5.1 Referenzhandbuch, Kapitel 20
2. Verbesserte MySQL-Erweiterung (MySQLi) (http://www.php.net/manual/de/mysqli.overview.php) PHP Manual
3. addslashes (http://de.php.net/manual/de/function.addslashes.php) PHP Manual
4. mysql_real_escape_string (http://de.php.net/manual/de/function.mysql-real-escape-string.php) PHP Manual
5. Magic Quotes (http://www.php.net/manual/de/security.magicquotes.php) PHP Manual
6. Chris Shiflett: addslashes() Versus mysql_real_escape_string() (http://shiflett.org/blog/2006/jan/addslashes-versus-mysql-real-escape-string) (englisch)
7. ColdFusion Online Hilfe (http://livedocs.macromedia.com/coldfusion/7/htmldocs/00000317.htm#1102474)

Kategorien: Wikipedia:Lesenswert | Sicherheitslücke
