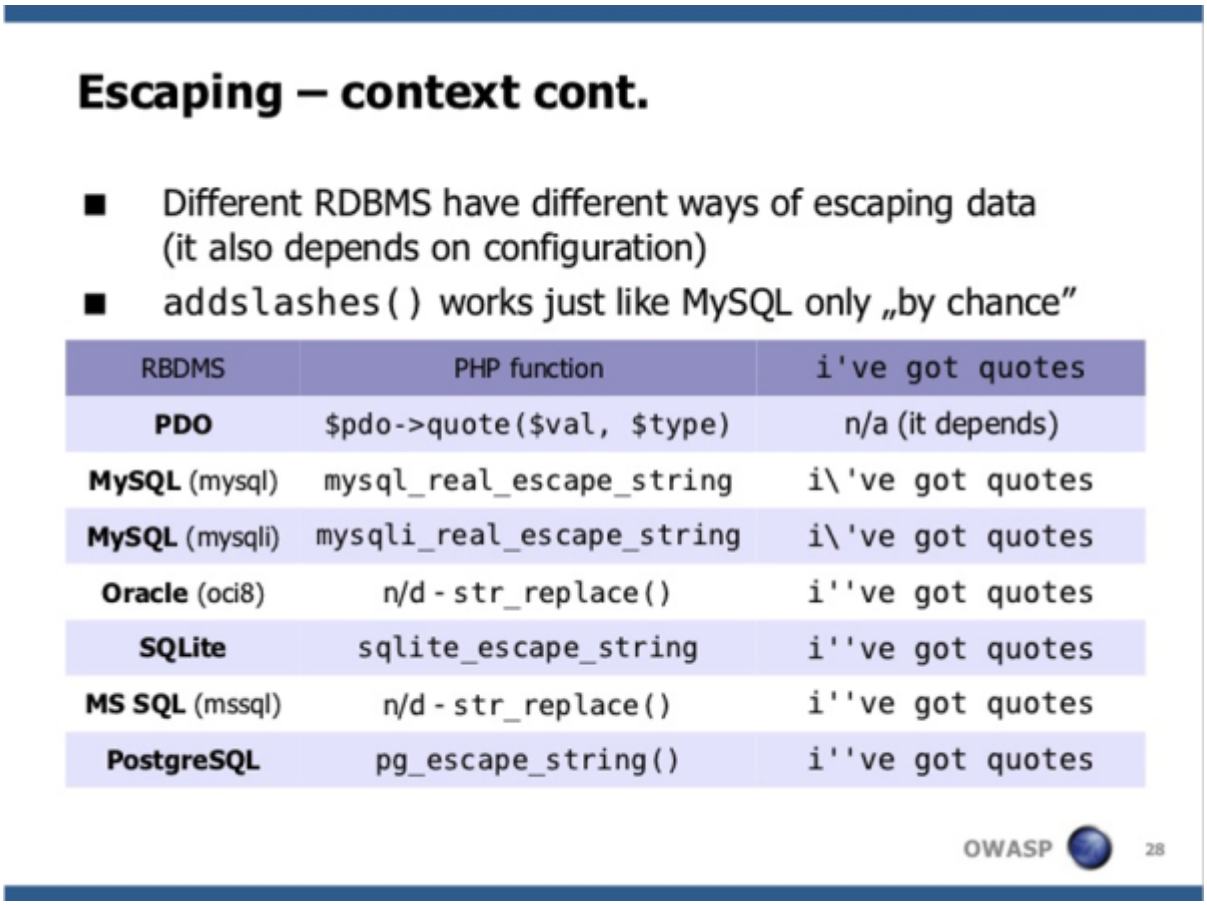
# LU11c - SQLi Gegenmassnahmen

Glücklicherweise gibt gegen alle SQLi-Varianten entsprechende Gegenmassnahmen. Die hier vorliegende Liste ist daher nur ein Ausschnitt mit den prominentesten.

## Escaping

Beim *escaping* werden Sonderzeichen nicht direkt in die Datenbank weitergeleitet. Die vom User eingegebenen Werte werden durch eine entsprechende Funktion/Methode, die es in vielen Programmiersprachen gibt vor der Weiterleitung gefiltert. Sonderzeichen, die den Angriffspunkt darstelle, werden von *gefährlichen* Steuerzeichen in *harmlose* Characters umgewandelt.Das „Escapen“ kann dabei auf der Client- und Serverseite geschehen.



**Escaping – context cont.**

- Different RDBMS have different ways of escaping data (it also depends on configuration)
- `addslashes()` works just like MySQL only „by chance”

| RBDMS | PHP function | i've got quotes |
|---|---|---|
| **PDO** | `$pdo->quote($val, $type)` | n/a (it depends) |
| **MySQL** (mysql) | `mysql_real_escape_string` | i\'ve got quotes |
| **MySQL** (mysqli) | `mysqli_real_escape_string` | i\'ve got quotes |
| **Oracle** (oci8) | n/d - `str_replace()` | i''ve got quotes |
| **SQLite** | `sqlite_escape_string` | i''ve got quotes |
| **MS SQL** (mssql) | n/d - `str_replace()` | i''ve got quotes |
| **PostgreSQL** | `pg_escape_string()` | i''ve got quotes |

OWASP 28

Das Hauptproblem bei SQLI besteht darin, dass Werte ungeprüft in die Datenbank weitergegeben werden, wobei Platzhalter mit vom User zur Laufzeit eingegebenen Werten angereichert werden. Enthält eine solcher zur Laufzeit generierte SQL-Anweisung Injektion, wird diese ungeprüft und ungepuffert an die DB weitergeleitet.

## Parametrisierte Werteübergabe

Die *parametrisierte Werteübergabe*, kurz *Parametrisierung* genannt, kann eine mögliche Gegenmassnahme gegen SQLI sein. Dabei werden die von User eingegebenen Werte in lokale

Parameter geschrieben. Diese Werte werden dann einen Schritt später durch Platzhalter in das SQL-Statement eingefügt.

**Beispiel mit JavaScript/MySQL-Beispiel**

```
// 1. Schritt: Zusammenbau des SQL-Statements mit Platzhaltern
const sql1 = `
  UPDATE Production.ProductInventory
  SET Quantity = @qty
  WHERE ProductID = @productId
`;
```

```
// 2. Schritt: Parametrisieren der Übergabewerte
const qty = 10;
const productId = 709;
```

```
// 3. Schritt: Zusammenbauen und Ausführen mit Laufzeitwerten
const sql = require('mssql');
async function updateInventory() {
  try {
    // Verbindung herstellen (Beispielconfig anpassen!)
    const pool = await sql.connect({
      user: 'username',
      password: 'password',
      server: 'localhost',
      database: 'AdventureWorks'
    });
    //
    const request = pool.request();
    request.input('qty', sql.Int, qty);
    request.input('productId', sql.Int, productId);
    //
    const result = await request.query(sql1);
    console.log('Update erfolgreich:', result.rowsAffected);
  } catch (err) {
    console.error('Fehler beim Update:', err);
  }
}
updateInventory();
```

# Input Validation

**Input Validation** bedeutet, dass alle Eingaben aus unsicheren Quellen (User, Formulare, API-Aufrufe, Cookies, Query-Strings, etc.) müssen überprüft werden, bevor sie in die Anwendung oder die Datenbank gelangen. Das Ziel ist es nur Werte zulassen, die dem erwarteten Format entsprechen. Alles andere wird abgelehnt oder bereinigt.

**Oder kurz:** Traue nie dem Input – prüfe ihn, bevor du ihn anfasst.

# Prepared Statement

Ein *Prepared Statement* trennt klar Code (das SQL) von Daten (Benutzereingaben). Dadurch kann ein Angreifer die Struktur der Abfrage nicht durch manipulierte Eingaben verändern — und genau das ist der Kern jeder SQL-Injection.

### Wie funktionieren sie?

1. Die Anwendung schickt ein SQL-Template an die Datenbank, in dem Platzhalter statt konkreter Werte stehen.
    - Beispiel (pseudo): SELECT * FROM users WHERE id = ?
2. Die Datenbank parst und compiliert dieses Template, legt einen Ausfuehrungsplan an.
3. Zur Laufzeit werden die Platzhalter separat mit Werten gefuellt — diese Werte werden nie als Teil des SQL-Codes interpretiert, sondern rein als Daten behandelt.

Ergebnis: Eingaben wie 1; DROP TABLE users; koennen nicht als zusätzlicher SQL-Code ausgefuehrt werden — sie landen als sicherer Datenwert in der Abfrage.

### Warum das gegen SQLi hilft

- **Trennung von Code und Daten:** Eingaben koennen die SQL-Syntax nicht mehr verändern.
- **Schutz gegen die meisten Angriffsvarianten:** Union-, Error- und Blind-SQLi werden dadurch weitgehend unwirksam, weil die Datenbank die Struktur bereits fixiert hat.
- Performance-Bonus: Bei wiederholten Abfragen wird der vorbereitete Plan wiederverwendet — das ist ein netter Nebeneffekt.

**Beispiel JavaScript** Die Werte in [...] sind rein Daten — kein Code.

```
// Platzhalter $1, $2 – Parameter separat uebergeben
const sql = 'UPDATE Production.ProductInventory SET Quantity = $1 WHERE
ProductID = $2';
await client.query(sql, [quantity, productId]);
```

# Stored Procedure

Kurzfassung zuerst: Stored Procedures (SPs) verlagern SQL-Logik in die Datenbank. Richtig verwendet trennen sie Code (die prozedurale Logik in der DB) von Daten (die Parameter), reduzieren Angriffsflächen und erleichtern Zugriffssteuerung. Keine Zauberformel — sie helfen, sind aber kein Freifahrtschein.

```
-- ============================================
-- Template generated from Template Explorer using:
-- Create Procedure (New Menu).SQL
--
-- Use the Specify Values for Template Parameters
-- command (Ctrl-Shift-M) to fill in the parameter
-- values below.
--
-- This block of comments will not be included in
-- the definition of the procedure.
-- ============================================
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
-- =============================================
-- Author:      <Author,,Name>
-- Create date: <Create Date,,>
-- Description: <Description,,>
-- =============================================
CREATE PROCEDURE <Procedure_Name, sysname, ProcedureName>
   -- Add the parameters for the stored procedure here
   <@Param1, sysname, @p1> <Datatype_For_Param1, , int> = <Default_Value_For_Param1, , 0>,
   <@Param2, sysname, @p2> <Datatype_For_Param2, , int> = <Default_Value_For_Param2, , 0>
AS
BEGIN
   -- SET NOCOUNT ON added to prevent extra result sets from
   -- interfering with SELECT statements.
   SET NOCOUNT ON

    -- Insert statements for procedure here
   SELECT <@Param1, sysname, @p1>, <@Param2, sysname, @p2>
END
GO
```

### Wie stored procedures grundsätzlich schützen

- Die SQL-Struktur liegt in der Datenbank und nicht in applikationsseitig zusammengesetzten Strings.
- Parameter werden (wenn sie als Parameter benutzt werden) von der DB engine als Daten behandelt — nicht als Teil des SQL-Codes.
- DB-Accounts können nur EXECUTE-Rechte auf die SP erhalten, nicht direkte SELECT/UPDATE/DELETE-Rechte auf Tabellen. Damit wird least privilege einfacher durchsetzbar.
- SPs erlauben zentralisierte Validierung/Logging und vereinfachen Auditing.

**Kurz:** SP machen es einem Angreifer schwerer, per manipulierten Input die Abfrage-Struktur zu verändern — sofern diese nicht durch unsauberen Code zunichte gemacht wurde.

---