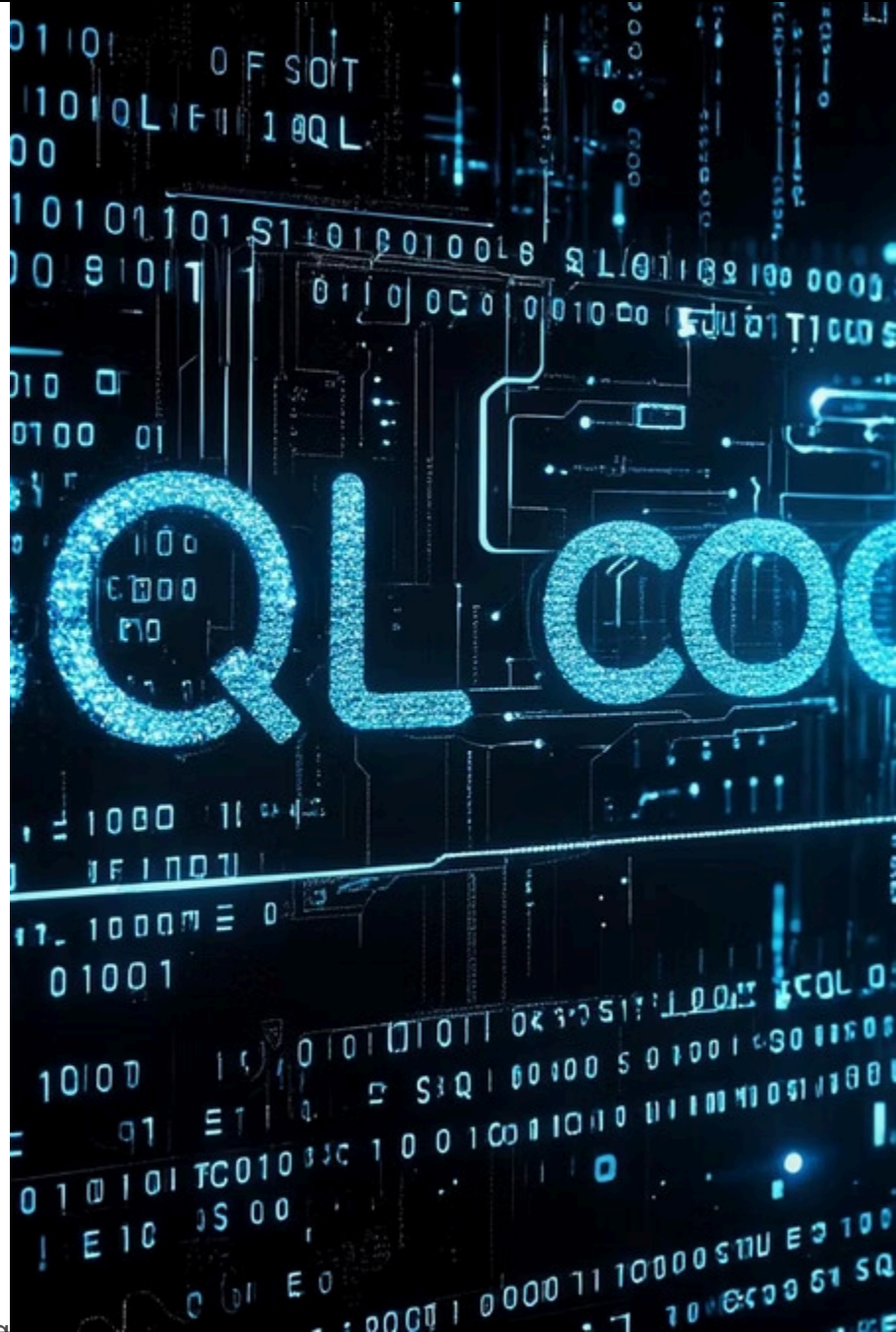
# SQL, Queries & SQL Injection

Eine technische Einführung für Sicherheitsbeauftragte und IT-Fachleute



CloudCommand GmbH chr.schumacher@g

# Agenda

1

### Grundlagen von SQL

Geschichte, Anwendungsbereiche und Datenbanktypen

2

### SQL-Befehle und Syntax

Kernbefehle und praktische Beispiele

3

### SQL Injection: Konzept

Funktionsweise und Angriffsszenarien

4

### Typen von SQL Injection

Klassisch, Blind, Fehlerbasiert, Second-Order

5

### Präventionsmaßnahmen

Technische und organisatorische Schutzmaßnahmen

6

### Praktische Übungen

Realistische Anwendungsfälle und Lösungsansätze

# Geschichte und Bedeutung von SQL

SQL (Structured Query Language) wurde in den 1970er Jahren von IBM entwickelt. Ursprünglich bekannt als SEQUEL (Structured English Query Language), wurde es später als ANSI- und ISO-Standard etabliert.

Heute ist SQL die Standardsprache für relationale Datenbanken und bildet das Rückgrat zahlreicher Unternehmensanwendungen, von Finanzsystemen bis hin zu E-Commerce-Plattformen.

# Datenbanktypen im Überblick

### Relationale Datenbanken

- MySQL, MariaDB, MSSQL, PostgreSQL, Oracle
- Tabellenstruktur mit definierten Beziehungen
- ACID-Eigenschaften (Atomarität, Konsistenz, Isolation, Dauerhaftigkeit)
- Umfassende SQL-Unterstützung

### NoSQL-Datenbanken

- MongoDB (dokumentenbasiert), Redis (Key-Value), Cassandra (spaltenbasiert)
- Schemalos, flexibel und hochskalierbar
- Eigene Abfragesprachen oder eingeschränkte SQL-Unterstützung
- Andere Sicherheitsherausforderungen als klassische SQL-Injection

### Dateibasierte Systeme

- Btrieve und ähnliche Systeme
- Meist in Legacy-Anwendungen
- Eingeschränkte Sicherheitskonzepte
- Keine vollständige SQL-Unterstützung

# Das relationale Datenbankmodell

Der Begriff "relational" bezieht sich auf das von Edgar F. Codd 1970 entwickelte Konzept der Datenrelationen - mathematisch definierte Beziehungen zwischen Tabellen.

- Daten in Tabellen (Relationen) organisiert
- Jede Tabelle besteht aus Zeilen (Tupel) und Spalten (Attribute)
- Beziehungen über Schlüssel definiert (Primär- und Fremdschlüssel)
- Normalisierung reduziert Redundanz und verbessert Datenintegrität

# Grundlegende SQL-Befehle: Daten abfragen

## SELECT-Anweisung

Die SELECT-Anweisung ist der Grundbaustein für Datenabfragen:

```
-- Alle Spalten auswählen
SELECT * FROM Kunden;

-- Bestimmte Spalten auswählen
SELECT Name, Email FROM Kunden;

-- Mit Filterbedingung
SELECT Name, Telefon
FROM Kunden
WHERE Umsatz > 5000;

-- Mit Sortierung
SELECT * FROM Kunden
ORDER BY Name ASC;
```

Der SELECT-Befehl ist der am häufigsten für SQL-Injection missbrauchte Befehl, da er in fast allen datengetriebenen Anwendungen verwendet wird.

CloudCommand GmbH chr.schumacher@gmx.tm

# Grundlegende SQL-Befehle: Daten manipulieren

## INSERT

```
-- Neuen Datensatz einfügen
INSERT INTO Kunden (Name, Email)
VALUES ('Max Mustermann',
'max@example.com');

-- Mehrere Datensätze einfügen
INSERT INTO Kunden (Name, Email)
VALUES
('Anna Schmidt',
'anna@example.com'),
('Peter Müller', 'peter@example.com');
```

## UPDATE

```
-- Datensatz aktualisieren
UPDATE Kunden
SET Umsatz = 7500
WHERE KundenID = 10;

-- Mehrere Spalten aktualisieren
UPDATE Kunden
SET Umsatz = Umsatz + 500,
    Status = 'Premium'
WHERE Jahresumsatz > 10000;
```

## DELETE

```
-- Datensatz löschen
DELETE FROM Kunden
WHERE KundenID = 10;

-- VORSICHT: Alle Datensätze löschen!
DELETE FROM Kunden;

-- Mit JOIN (Beispiel)
DELETE K FROM Kunden K
JOIN Blacklist B ON K.Email = B.Email;
```

# SQL Injection: Das Konzept

SQL Injection (SQLi) ist eine Angriffstechnik, bei der schädlicher SQL-Code in Eingabefelder eingeschleust wird, um nicht autorisierte Datenbankoperationen auszuführen.

Laut OWASP zählt SQL Injection seit Jahren zu den gefährlichsten Sicherheitslücken in Webanwendungen. Erfolgreiche Angriffe können zu:

- Unberechtigtem Datenzugriff führen
- Datenmanipulation oder -löschung ermöglichen
- Authentifizierungsmechanismen umgehen
- Im schlimmsten Fall vollständige Serverübernahme ermöglichen

# Funktionsweise von SQL Injection

### Ursprünglicher Code

```
// PHP-Code (anfällig)
$suche = $_GET['suche'];
$query = "SELECT * FROM Produkte
        WHERE Name LIKE
'%$suche%'";
$result = mysqli_query($conn,
$query);
```

Entwickler erwartet normale Suchbegriffe wie "Laptop" oder "Drucker"

### Angreifer-Eingabe

```
' OR '1'='1

// Alternative mit Kommentar
' OR '1'='1' --
```

Der Angreifer schleust eine logische Bedingung ein, die immer wahr ist

### Resultierende Abfrage

```
SELECT * FROM Produkte
WHERE Name LIKE '%' OR '1'='1'%'
```

Die Bedingung '1'='1' ist immer wahr, daher werden ALLE Datensätze zurückgegeben

# Gefährliche SQL Injection Beispiele

| Angriffsziel | Injizierter Code | Potenzielle Auswirkung |
|---|---|---|
| Login umgehen | ' OR '1'='1' -- | Anmeldung ohne Passwort |
| Daten zerstören | '; DROP TABLE Kunden; -- | Vollständiger Datenverlust |
| Datenexfiltration | ' UNION SELECT Kreditkartennummer, Name FROM Zahlungen -- | Diebstahl sensibler Daten |
| Datenbankschema auslesen | ' UNION SELECT table_name, column_name FROM information_schema.columns -- | Aufdeckung der Datenbankstruktur |
| Betriebssystem-Befehle | '; EXEC xp_cmdshell 'net user hacker password /ADD'; -- | Erstellung eines Systemnutzers (bei MSSQL) |

Diese Angriffe können besonders in Legacy-Systemen ohne moderne Sicherheitsmaßnahmen erfolgreich sein.

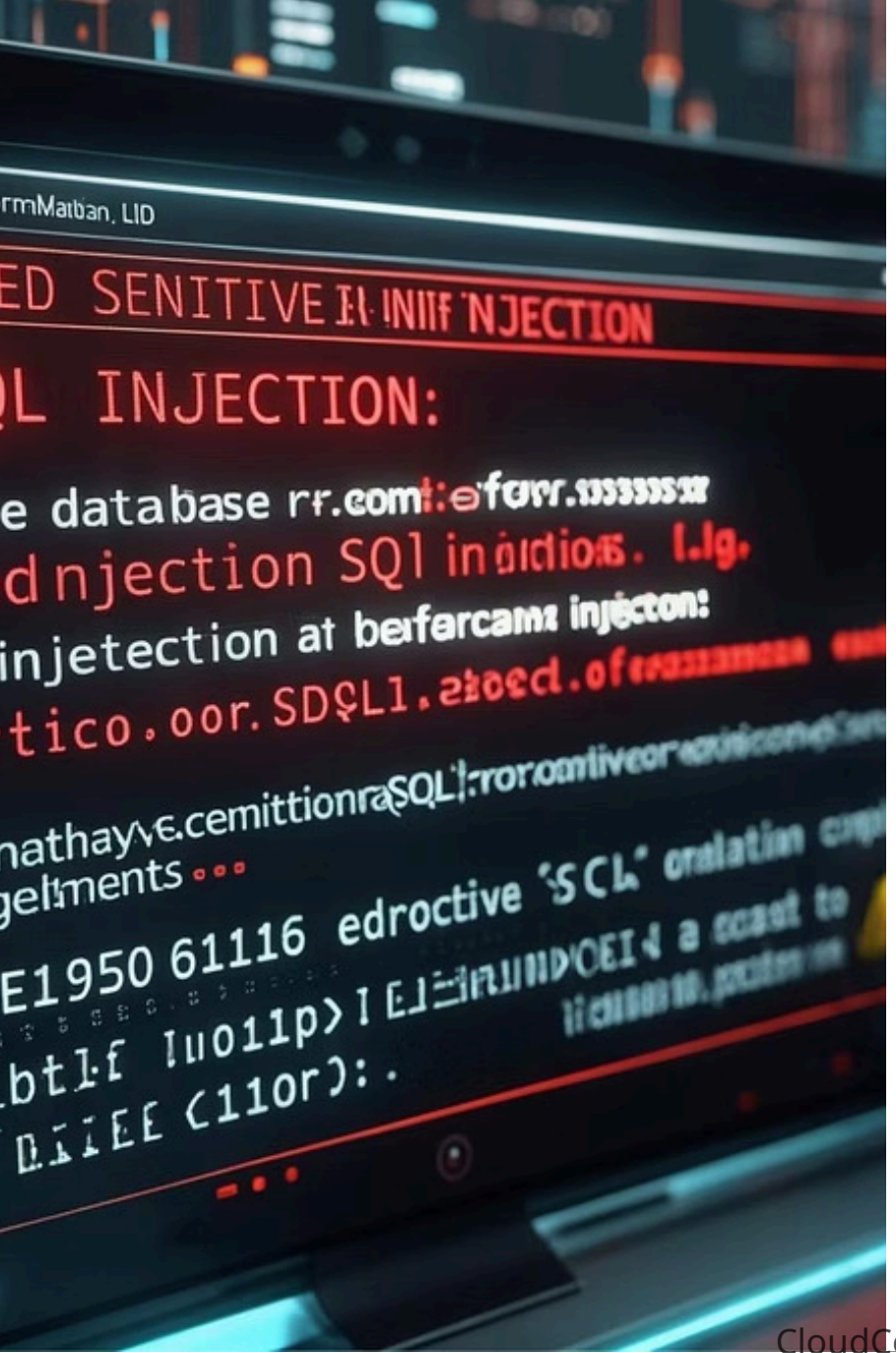
# Typen von SQL Injection: Klassisch und Blind

## Klassische SQL Injection

- Ergebnisse werden direkt in der Anwendung angezeigt
- Fehler und Ausgaben sind für den Angreifer sichtbar
- Schnelles Feedback ermöglicht effizientes Testen
- Beispiel: ' UNION SELECT username, password FROM users --

## Blind SQL Injection

- Keine direkten Ergebnisse oder Fehlermeldungen sichtbar
- Angriff basiert auf indirekten Indikatoren (Boolean/Time-based)
- Beispiel Boolean-based: ' AND (SELECT 1 FROM admin WHERE username='admin' AND SUBSTRING(password,1,1)='a')=1 --
- Beispiel Time-based: ' AND IF(SUBSTRING(user(),1,1)='r',SLEEP(5),0) --



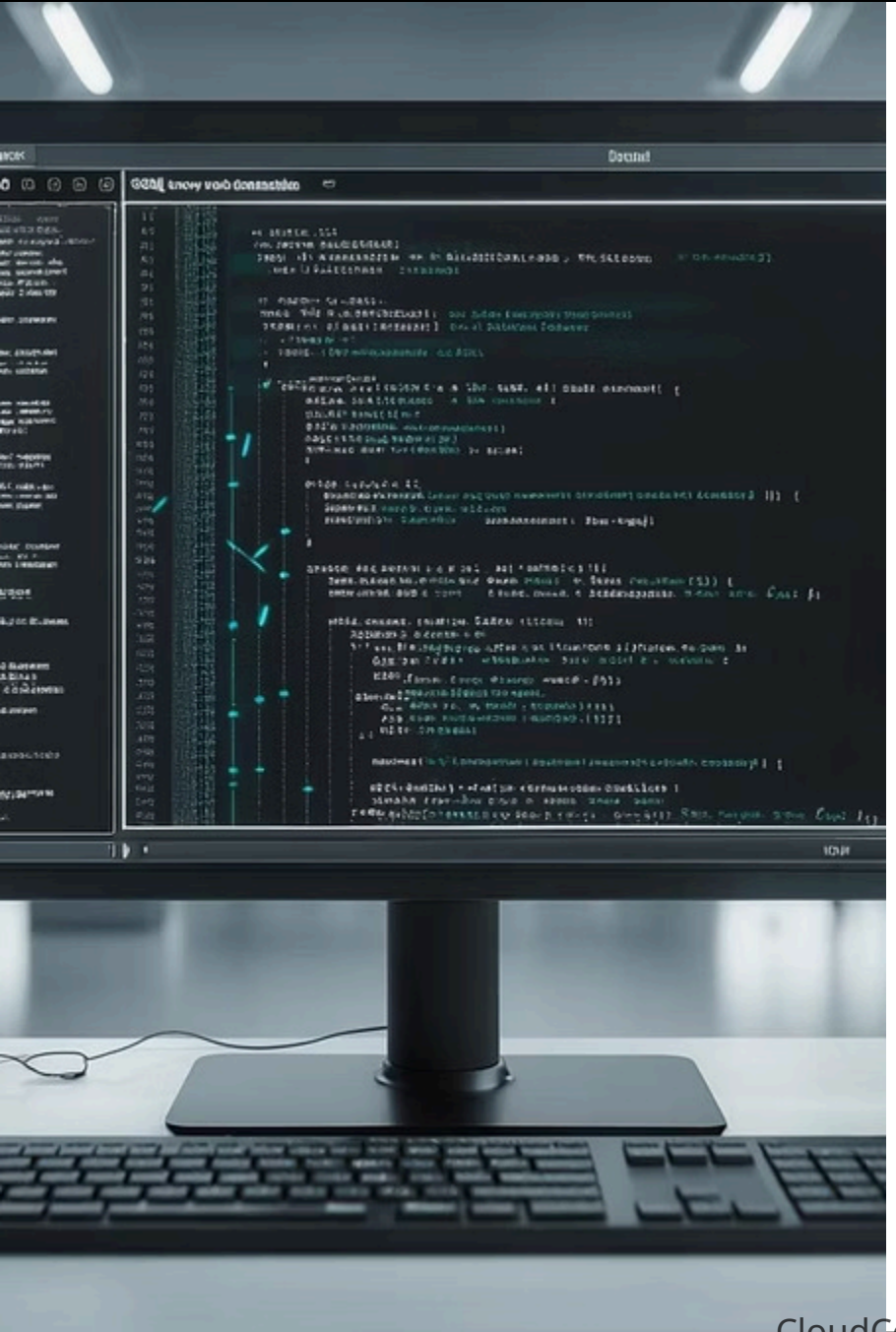

# Typen von SQL Injection: Fehlerbasiert und Second-Order

## Fehlerbasierte SQL Injection

- Nutzt Fehlermeldungen zur Informationsgewinnung
- Besonders effektiv bei detaillierten Fehlermeldungen
- Beispiel: ' AND (SELECT 1 FROM (SELECT COUNT(*), CONCAT(version(),FLOOR(RAND(0)*2))x FROM information_schema.tables GROUP BY x)a) --
- Kann Informationen extrahieren, selbst wenn die eigentliche Abfrage fehlschlägt

## Second-Order SQL Injection

- Injizierterer Code wird zunächst gespeichert und später ausgeführt
- Schwerer zu erkennen, da der Angriff zeitversetzt stattfindet
- Beispiel: Schädlicher Code wird in ein Nutzerprofil eingefügt und bei späterem Zugriff aktiviert
- Umgeht häufig Standard-Schutzmaßnahmen



# Praxisbeispiel: SQL Injection in einem ERP-System

## Ausgangssituation

Ein mittelständisches Unternehmen nutzt ein älteres ERP-System mit integriertem SQL-Editor für Berichterstellung

## Sicherheitslücke

Der SQL-Editor wird auch von Fachabteilungen genutzt und verfügt über zu weitreichende Berechtigungen (sysadmin)

## Angriffsszenario

Ein unzufriedener Mitarbeiter führt folgende Abfrage aus: SELECT * FROM Kunden; DROP TABLE Aufträge;

## Auswirkung

Vollständiger Verlust der Auftragsdaten, erhebliche Betriebsstörung, aufwändige Wiederherstellung

# Moderne Schutzmechanismen: Prepared Statements

## Funktionsweise

Prepared Statements trennen SQL-Code von Eingabedaten, indem sie Platzhalter verwenden. Die Datenbank kompiliert die Anweisung vor der Eingabe der Parameterwerte.

- Kompilierung der SQL-Anweisung vor Parameterbindung
- Sicheres Escaping der Eingabewerte durch die Datenbank
- Höhere Performance bei wiederholten Abfragen

```
// Unsicher (PHP)
$query = "SELECT * FROM Users
WHERE username = '$username'
AND password = '$password'";

// Sicher mit Prepared Statement (PHP/PDO)
$stmt = $pdo->prepare("SELECT * FROM Users
WHERE username = ? AND password = ?");
$stmt->execute([$username, $password]);

// Sicher mit benannten Parametern
$stmt = $pdo->prepare("SELECT * FROM Users
WHERE username = :user AND password = :pw");
$stmt->execute(['user' => $username,
 'pw' => $password]);
```

# Moderne Schutzmechanismen: ORM und Input Validation

### ORM (Object-Relational Mapping)

Frameworks wie Hibernate (Java), Entity Framework (.NET) oder Doctrine (PHP) abstrahieren die Datenbankinteraktion:

- Automatische Umwandlung von Objekten in Datenbankstrukturen
- Integrierte Schutzmaßnahmen gegen SQL Injection
- Beispiel: `User user = userRepository.findByUsername(username);`

### Input Validation & Sanitization

Validierung und Bereinigung aller Benutzereingaben:

- Whitelisting erlaubter Zeichen und Werte
- Typprüfungen (z.B. Integer für IDs)
- Entfernung oder Escaping potenziell gefährlicher Zeichen
- Beispiel: `$id = filter_input(INPUT_GET, 'id', FILTER_VALIDATE_INT);`

### Stored Procedures

Vordefinierte Datenbankfunktionen mit kontrollierten Parametern:

- Ausführung mit minimalen Berechtigungen
- Kein direkter Zugriff auf Tabellen
- Beispiel: CALL sp_GetUserByID(5);

# Erkennung von SQL Injection

## Manuelle Testmethoden

- Einfügen von Sonderzeichen (', ", ;, --)
- Testen von Logikoperatoren (OR 1=1)
- Verzögerungsbefehle (SLEEP, BENCHMARK)
- Syntaxfehler provozieren und Fehlermeldungen analysieren

## Automatisierte Tools

- sqlmap: Umfassendes Open-Source-Tool für SQLi-Tests
- OWASP ZAP: Integrierte SQLi-Scanner
- Burp Suite: Proxy mit SQLi-Erkennungsfunktionen

⚠ Die Verwendung von SQLi-Testtools ohne ausdrückliche Genehmigung kann rechtswidrig sein und sollte nur in kontrollierten Umgebungen oder mit entsprechender Autorisierung erfolgen!

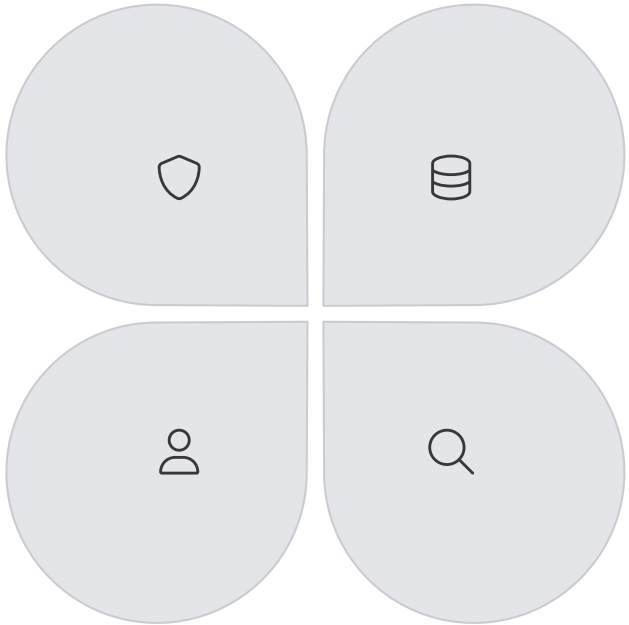
# Präventionsmaßnahmen im Überblick

## Technische Maßnahmen

- Prepared Statements / Parametrisierte Abfragen
- ORM-Frameworks
- WAF (Web Application Firewall)
- Aktuelle Patches für Datenbanken und Frameworks

## Organisatorisch

- Regelmäßige Sicherheitsschulungen
- Secure Coding Guidelines
- Code-Reviews mit Sicherheitsfokus
- Penetrationstests durch externe Experten



## Datenbankdesign

- Principle of Least Privilege
- Trennung von Lese- und Schreibzugriffen
- Stored Procedures statt direkter Tabellenzugriff
- Keine DBA-Rechte für Anwendungskonten

## Monitoring

- Logging aller Datenbankaktivitäten
- Real-time Anomalieerkennung
- Audit-Trails für kritische Daten
- Überwachung verdächtiger Abfragemuster

# Praktische Übung: SQL Injection erkennen

## Aufgabe

Identifizieren Sie die SQL Injection-Schwachstellen in folgenden Code-Beispielen:

```
// Beispiel 1: PHP
$id = $_GET['id'];
$query = "SELECT * FROM Produkte WHERE id = $id";

// Beispiel 2: Java
String query = "SELECT * FROM users WHERE name = '"
          + request.getParameter("username")
          + "' AND password = '"
          + request.getParameter("password") + "'";
Statement stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery(query);

// Beispiel 3: C#
string sql = "UPDATE users SET password = '"
         + txtNewPassword.Text
         + "' WHERE id = " + userId;
```

## Lösungsansätze

```
// Korrektur Beispiel 1: PHP mit PDO
$id = filter_input(INPUT_GET, 'id', FILTER_VALIDATE_INT);
if ($id === false) {
 die("Ungültige ID");
}
$stmt = $pdo->prepare("SELECT * FROM Produkte
 WHERE id = ?");
$stmt->execute([$id]);

// Korrektur Beispiel 2: Java mit PreparedStatement
String query = "SELECT * FROM users WHERE name = ?
 AND password = ?";
PreparedStatement pstmt = connection.prepareStatement(query);
pstmt.setString(1, request.getParameter("username"));
pstmt.setString(2, request.getParameter("password"));
ResultSet rs = pstmt.executeQuery();
```

# Praktische Übung: SQL Injection simulieren

## 1 Aufbau einer Testumgebung

- DVWA (Damn Vulnerable Web Application) oder WebGoat installieren
- Virtualisierte Testumgebung ohne Netzwerkverbindung nutzen
- Schwierigkeitsgrad niedrig wählen für erste Tests

## 2 Manuelle Injection durchführen

- Einfache Teststrings eingeben: ' OR '1'='1
- Datenbank-Version ermitteln: ' UNION SELECT version(), NULL #
- Tabellennamen auslesen: ' UNION SELECT table_name, NULL FROM information_schema.tables #

## 3 Automatisierte Tests mit sqlmap

- Grundlegende Erkennung: sqlmap -u "http://localhost/dvwa/vulnerabilities/sqli/?id=1"
- Datenbank-Enumeration: sqlmap -u "..." --dbs --tables
- Datenextraktion: sqlmap -u "..." -D dvwa -T users --dump

## 4 Analyse und Dokumentation

- Erfolgreiche Angriffsvektoren dokumentieren
- Extrahierte Daten protokollieren
- Sicherheitslücken klassifizieren nach Schweregrad
- Gegenmaßnahmen für jede Lücke definieren

# Zusammenfassung und nächste Schritte

## Kernpunkte

- SQL Injection bleibt eine der gefährlichsten Sicherheitslücken in Webanwendungen
- Verschiedene Angriffstypen erfordern unterschiedliche Erkennungs- und Abwehrstrategien
- Parametrisierte Abfragen und ORM sind die wirksamsten technischen Schutzmaßnahmen
- Sicherheit erfordert einen ganzheitlichen Ansatz: Technik, Design, Monitoring und Schulung

## Empfohlene nächste Schritte

1. Sicherheitsaudit der eigenen Anwendungen durchführen
2. Legacy-Systeme identifizieren und Modernisierungsstrategie entwickeln
3. Entwicklerteam in sicherer Programmierung schulen
4. Regelmäßige Penetrationstests etablieren
5. Incident-Response-Plan für Datenbankbezogene Sicherheitsvorfälle erstellen

Für weiterführende Informationen empfehlen wir die OWASP SQL Injection Prevention Cheat Sheet sowie die aktuellen Sicherheitshinweise des BSI zu Datenbankangriffen.