Verteidigung gegen SQL Injection Attacks

Semesterarbeit SS 2003

Daniel Lutz

danlutz@watz.ch

Inhalt

- Motivation
- Demo-Applikation
- Beispiele von Attacken
- Massnahmen zur Verteidigung
- Schlussfolgerungen

Motivation

- Aktuelles Problem
  - Web-Applikationen
  - Allgemein: Datenbank-basierte Applikationen
- Auch kommerzielle Applikationen/Frameworks betroffen
  Beispiel: *Xpressions Software: Multiple SQL Injection Attacks To Manage WebStore* (Juni 2003)

Ziele der Semesterarbeit

- Problem-Analyse
- "Anleitung" zur Entwicklung einer sicheren Applikation
- Framework: Java-Klassen zur Unterstützung

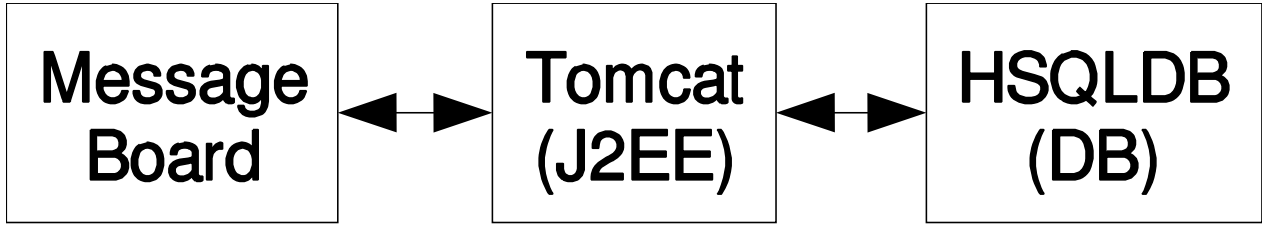
Demo-Applikation

- Praktischer Nachvollzug der Attacken
- 2 Versionen:
  - unsichere Version
  - sichere Version

- Message Board

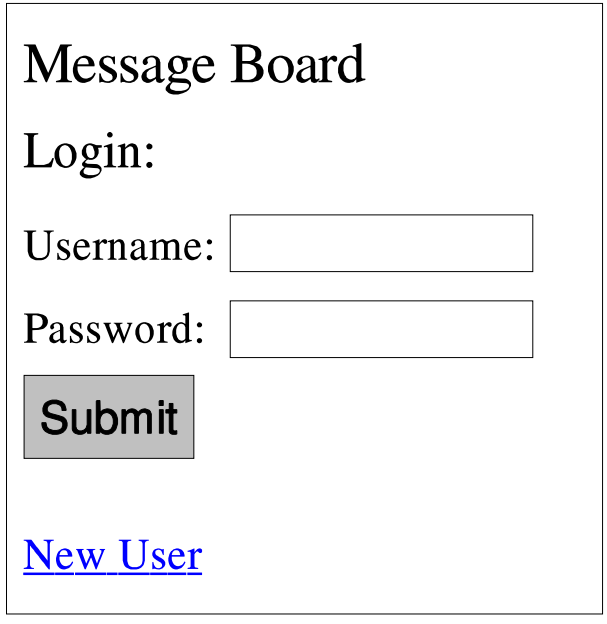
Architektur:

- Java 2 Enterprise Edition (J2EE)
- JSP, JavaBeans
- HSQL Database Engine (HSQLDB)



Beispiele von Attacken (1)

Authentifizierung:



Felder: *username*
password

Beispiele von Attacken (1)

Authentifizierung:

`authenticate.jsp:`

```
<%-- check if user is available --%>
<sql:query var="result">
  SELECT * FROM users
  WHERE username = '<c:out value="${param.username}" escapeXml="false" />'
  AND password = '<c:out value="${param.password}" escapeXml="false" />'
</sql:query>

<c:if test="${result.rowCount == 0}">
  <c:set var="errorMsg" scope="session">Invalid Username or Password</c:set>
  <c:redirect url="index.jsp" />
</c:if>
```

Beispiele von Attacken (1)

Authentifizierung:

```
SELECT * FROM users
WHERE username = '${username}'
AND password = '${password}'
```

username:	*(leer)*
password:	`' or " = '`

```
SELECT * FROM users
WHERE username = ''
AND password = '' or '' = ''
```

Bedingung des Statements ist für alle Datensätze erfüllt!
$\rightarrow$ ResultSet enthält alle Datensätze, der erste Datensatz wird verwendet.

Beispiele von Attacken (2)

Beliebiger SQL-Code:

`showmsg.jsp:`

```
SELECT * FROM messages
WHERE messageid = ${messageid}
```

Tabelle löschen:

Ursprünglicher URL:

```
https://localhost:8443/insecure/showmsg.jsp ➥
  ?messageid=1
```

URL mit SQL-Injection:

```
https://localhost:8443/insecure/showmsg.jsp ➥
  ?messageid=1;drop+table+users
```

```
SELECT * FROM messages
WHERE messageid = 1;drop table users
```

Tabelle `users` wird gelöscht!

Weitere Attacken

Nur 2 Beispiele von mehreren möglichen Attacken.
Weitere Attacken:

- Einloggen als spezifischer Benutzer
- Daten einfügen
- Datenbank-Konto erstellen
- Daten anzeigen
- Systemvariablen anzeigen
- Tabellenstruktur ermitteln
- ...

(Vgl. Semesterarbeit)

Massnahmen zur Verteidigung

1. Überprüfung der Eingabewerte
2. Prepared Statements
3. Auslagerung der Business-Logik in Beans
4. Spezielles Konto für Datenbankzugriff
5. Passwörter nicht als Plaintext speichern
6. Fehlermeldungen vermeiden
7. Code-Review

$\rightarrow$ Defence in Depth

1. Überprüfung der Eingabewerte

- String, Zahl (Integer, Float), Datum
- Regular Expressions (Einschränkung der zulässigen Werte)

Hilfsmittel: Verifier-Klassen für String, Integer, Float und Date

```
String username = request.getParameter("username");
StringVerifier usernameVerifier = new StringVerifier(username, "[a-zA-Z0-9]+", true);

if (!usernameVerifier.verify()) {
  // error
}

try {
  username = usernameVerifier.stringValue();
}
catch (VerifierException e) {
  // error
}
```

2. Prepared Statements

- Daten nicht direkt in SQL-Code einbetten
- Daten separat an Datenbank-Server senden
- Überprüfung der Datentypen durch Datenbank-Server

```
String sql = "SELECT * FROM users"
  + " WHERE username = ?"
  + " AND password = ?";

PreparedStatement ps = conn.prepareStatement(sql);
ps.setString(1, username);
ps.setString(2, passwordDigest);
ResultSet rs = ps.executeQuery();
```

Daten werden als Zeichenkette/Zahl, nicht als SQL-Code interpretiert
$\rightarrow$ Typen-Fehler werden vom Datenbank-Server erkannt

3. Auslagerung der Business-Logik in Beans

- Trennung von Business-Logik und Präsentation
- Schutz vor Fehlkonfiguration des Application-Servers (Anzeige des Quellcodes)

`authenticate.jsp`

↓

`authenticate.jsp`, `UserAuthenticator.java`

4. Spezielles Konto für Datenbankzugriff

- Konto mit möglichst wenigen Rechten
  - Konto darf nur Datensätze lesen, einfügen, ändern und löschen können
  - Konto darf keine Tabellen löschen können!
  - Konto darf keine Benutzer erstellen können!
  - etc.

Statt Konto “sa” z. B. Konto “messageboard”.

- Für jede Applikation eigenes Konto

(Applikation funktioniert mit “sa”, Massnahme bringt aber zusätzliche Sicherheit.)

5. Passwörter nicht als Plaintext speichern

- Schutz, falls Passwörter gelesen werden können
- Zwei Möglichkeiten:
  - Passwörter als Hash speichern (z. B. SHA)
  - Passwörter verschlüsseln (reversibel)

Hilfsklasse: PasswordDigest

```
// calculate SHA digest
String passwordDigest;
try {
  passwordDigest = PasswordDigest.calculateDigest(password);
}
catch (NoSuchAlgorithmException e) {
  // error
}
```

secret → e5e9fa1ba31ecd1ae84f75caaa474f3a663f05f4

6. Fehlermeldungen vermeiden

- Exceptions falls möglich abfangen
- Eigene Fehler-Seite anzeigen

$\rightarrow$ Informationen über Anwendungslogik verbergen

web.xml:

```
...
  <error-page>
    <exception-type>java.lang.Throwable</exception-type>
    <location>/error.jsp</location>
  </error-page>
...
```

7. Code-Review

- Durchsicht des Codes durch anderen Entwickler
- Übersehene Probleme können entdeckt werden

Keine technische, sondern *organisatorische* Massnahme.

Schlussfolgerungen

- Zentral: Überprüfung der Eingabewerte
- Kein Verlass auf nur eine Massnahme $\rightarrow$ Verkettung
- Die meisten Massnahmen können auf beliebigen Systemen angewendet werden (ASP.NET, PHP, CGI, C, C++, ...).

- Signifikante Unterschiede zwischen Datenbanken und -Treibern verschiedener Produkte

- Demo-Applikation dient als Illustration und zu Ausbildungszwecken

Details: schriftliche Arbeit