HOCHSCHULE MÜNCHEN

Fakultät 07 für Informatik und Mathematik

Aktuelle Technologien zur Entwicklung verteilter Java-Anwendungen

Dozent: Theis Michael

Studienarbeit

Sicherheit in unternehmenskritischen Applikationen

vorgelegt von:

Student:	Manuel Rudi Herdt
Studiengang:	Wirtschaftsinformatik (Bachelor)
Fachsemester:	IBB7W
Matrikelnummer:	05116016
Geburtsdatum:	01.12.1986
E-Mail:	manuel.herdt@yahoo.de

München, den 30.05.2014

Inhalt

Verzeichnis der Abbildungen

1 Einleitung

Diese Studienarbeit befasst sich mit Thema „Sicherheit in unternehmenskritischen Applikationen“. Im Rahmen der Studienarbeit werden als erstes die Grundlagen der Sicherheit im Sinne von Java EE behandelt. Anschließend werden verschiedene Maßnahmen der Java EE-Spezifikation zur Verbesserung des Schutzes vor unzulässigen Zugriffen an Beispielen erläutert. Des weiteren wird gezeigt wie weiter eigene Sicherheitsmechanismen in eine Java EE Umgebung integriert werden können. In der Einleitung werden zunächst einige Grundüberlegung aufgeführt, die für die Sicherheit einer Applikation von Bedeutung sind.

Was verstehen wir unter Security/Sicherheit? Im Sinne von Applikationen eines Unternehmens wäre dies die Sicherstellung der Verfügbarkeit des Services sowie die Kontrolle über den Zugriff von Benutzern oder Services auf sensitive Ressourcen.

Wieso ist die Sicherheit von Applikationen so wichtig?

Was verspricht man sich von den Sicherheitsmaßnahmen?

- Einhalten von Sicherheitsstandards
- Halten von Wettbewerbsvorteilen/-fähigkeit
- Investitionssicherheit
- Risikomanagement
- Vertrauen des Kunden

Ein wichtiger Aspekt, der bei dem Thema Security verinnerlicht werden muss, ist das unkoordinierte Einzelmaßnahmen zur Sicherung des Systems nie den benötigten Schutz bieten können, da bereits eine Lücke genügen kann um unerlaubte Zugriffe zu ermöglichen und dadurch großer Schaden entstehen kann. Daher ist es unabdingbar ein ausgewogenes und an das Unternehmen angepasste Sicherheitskonzept parat zu haben. Um ein sinnvolles Sicherheitskonzept zu realisieren sollte man sich zuerst der eigenen Rahmenbedingungen klar werden. Ein absolut vollständiger Schutz kann nie Gewährleistet werden und würde den Kostenrahmen eines jedem Projekt sprengen aber um eine gewisse Basis bei einem Sicherheitskonzept kommt man heute nicht mehr herum.

Um die Waage von Sicherheit und Kosten im Gleichgewicht zu halten ist eine Analyse des Projekts notwendig. Eine der ersten Fragen die man sich stellen sollte, ist:

Welche Arten von Gefahren gibt es?

Analyse der möglichen Gefahren:

- Natürliche Ereignisse
- Technisches Versagen
- Menschliches Versagen
- Kriminelle Handlungen

Was ist an meiner Applikation besonders schützenswert?

Analyse des Projekts:

- Welche Bereiche sind besonders gefährdet?
- Welche Bereiche sind Unternehmens-kritisch?
- Welche Bereiche/Daten müssen aus gesetzlicher Verpflichtung besonders gehandhabt werden? (z.B. Datenschutzgesetz)

Risiko-/Gefahrenanalyse:

- Wie relevant sind die Gefahren?
- Wie hoch steht die Wahrscheinlichkeit für deren Eintreten?
- Welche Bereiche sind von welchen Gefahren besonders betroffen?
- Welcher Schaden bzw. welche Folgen könnten beim Eintreten der Gefahrensituation entstehen?

Mit Hilfe dieser Informationen ist es möglich das benötigte Schutzniveau festzulegen und bestehende Schwachstellen in der Applikation zu erkennen. Aus der Analyse sollten nun konkrete Ziele definiert werden. Der Vergleich dieser Ziele mit dem IST-Zustand der Applikation sollte zeigen, ob Handlungsbedarf besteht. Nun können bislang unerreichte Ziele mit gezielte Schutz- bzw. Gegenmaßnahmen angegangen werden. Im Laufe des Applikationslebenszyklus sollten die oben aufgeführten Prozesse und Analysen im Rahmen des Qualitätsmanagement regelmäßig durchlaufen und dokumentiert werden. Denn nur so ist ein langfristiger Schutz der Applikation durch ständig neue Bedrohungen möglich.

2 Hauptteil

2.1 Grundlagen der Sicherheit im Sinne von Java EE

2.1.1 Sicherheitsarchitektur von Java EE

In der Java Enterprise Edition wird die Sicherheit der Applikationsschicht über die jeweiligen Container verwaltet. Die Authentisierung sowie die Autorisierung können über deklarative Sicherheitsmaßnahmen wie Annotationen und/oder über Deployment Deskriptors definiert werden. Dies hat zur Folge, dass die Sicherheit exakt an die Applikation angepasst werden kann. Dadurch sind die Sicherheitsmaßnahmen effektiver und einfacher zu administrieren. Die Java EE Plattform bietet auch eine Vielzahl von Authentifikationsmethoden wie z.B. HTTP Basic Authentication, Form-Based Authentication oder auch Mutual Authentication bei der die Authentifizierung beidseitig erfolgt. Neben den Sicherheitsmechanismen der Container gibt es auch noch diverse Möglichkeiten die Application Server wie JBoss oder GlassFish zu konfigurieren. In der Transportschicht sorgen HTTPS bzw. Secure Sockets Layer (SSL) für sichere Übertragung von Daten zwischen dem Client und dem Web Server. Es werden digitale Zertifikate benötigt um SSL zu nutzen.

2.1.2 Eigenschaften von Application Security

Java EE Applikationen bestehen aus offenen und geschützten Ressourcen. Die Application Security stellt sicher das nur berechtigte Benutzer auf geschützte Ressourcen zugreifen können.
Sie sollte folgenden Eigenschaften aufweisen:

- Authentication: Feststellen der Identität von Server und Client für die Autorisierung
- Authorization: Hat der Benutzer die Berechtigung um die Daten oder Operationen auszuführen
- Data Integrity: Sind die Daten von einer dritten Partei verändert worden
- Confidentiality: Sensitive Daten sollten nur von Berechtigten Benutzern eingesehen werden können
- Non-repudiation: Handlungen von Benutzern können nachvollzogen werden

- Quality of Service: Verbesserung der Dienstgüte über verschiedene Technologien
- Auditing: Möglichkeiten die Effektivität der Sicherheitsmaßnahmen zu evaluieren

2.1.3 Eigenschaften von Sicherheitsmechanismen

Es gibt eine Vielzahl von Möglichkeiten Anwendungen zu schützen. Detaillierter werden Hauptteil der Arbeit eingehen aber alle Sicherheitsmechanismen sollten möglichst folgende Funktionalitäten aufweisen:[1]

- Authentifikation um unautorisierten Zugriff auf Funktionen der Applikation und/oder sensitive Daten zu unterbinden
- Um Nutzer für ihre Operationen auf dem System falls notwendig zur Verantwortung zu ziehen
- Schutz des Systems vor Ausfällen und anderen Vorfällen die die Qualität des Services beeinträchtigen könnten

Im Optimalfall sollten die Maßnahmen, falls ordentlich programmiert, ebenfalls

- einfacher zu administrieren sein
- transparenter für den Anwender
- sowie interoperabel über Applikations- und Unternehmensgrenzen hinaus

2.1.4 Java SE Security Mechanismen[2]

Java SE bietet eine breite Palette an Möglichkeiten Applikationen zu härten.
Hier einige Beispiele:

- Java Authentication and Authorization Service (JAAS) ist eine Zusammenstellung von APIs die Dienste zur Authentifizierung und Autorisierung bereitzustellen. JAAS stellt Pluggable Authentication Modules (PAM) dafür bereit.
- Java Generic Security Services (Java GSS-API) ist eine auf Token basierende API und sorgt für sichere Kommunikation zwischen Applikationen.
- Java Cryptography Extension (JCE) stellt ein Framework und Implementierungen für Verschlüsslung, Schlüssel Generierung/Agreement, und Message Authentication Code (MAC) Algorithmen zur Verfügung.

[1] http://docs.oracle.com/javaee/7/tutorial/doc/security-intro001.htm
[2] http://docs.oracle.com/javaee/7/tutorial/doc/security-intro002.htm#BNBWY

- Java Secure Sockets Extension (JSSE) stellt ein Framework sowie eine Implementierung für eine Java Version von Secure Sockets Layer (SSL) und Transport Layer Security (TLS) zur Verfügung.
- Simple Authentication and Security Layer (SASL) ist ein Internet Standard (RFC 2222) der ein festes Protokoll zur Authentifikation beinhaltet. SASL definiert wie die Authentifizierungsdaten ausgetauscht werden sollen und legt dabei aber nicht fest wie diese Daten auszusehen haben

Java SE bietet noch weiter Werkzeuge zur Verwaltung von Schlüsseln, Zertifikaten oder Policy Files.

2.1.5 Realms, Users, Groups and Roles

In einer Web Applikation ist es notwendig gewisse Ressourcen vor unberechtigten zugriffen zu schützen. Um dies zu realisieren benötigen wir Realms, Useres, Groups und Roles.

Realm: Ein Realm ist eine für den Application bzw. Web Server definierte Security Policy Domain und beinhaltet eine Ansammlung von Benutzern die Gruppen zugeteilt sein können. Der Application Server GlassFish beinhaltet bereits vorkonfiguriert mit den Realms file, admin-realm und certificate.

User: Ein User ist eine Person oder eine Identität einer Applikation. User können Rollen zugewiesen und in Groups eingeteilt werden. Die Authentifikation der User kann über Benutzername/Passwort oder Zertifikat erfolgen.

Group: Eine Group ist eine Liste von authentifizierten Usern die Gemeinsamkeiten aufweisen und aufgrund dieser gleich behandelt werden.

Role: Eine Rolle entspricht einer Zugriffsberechtigung auf gewisse Ressourcen einer Applikation.

2.1.6 JBoss 6.1.0 Final

Die Deklarierung von Rollen und Benutzern erfolgt beim JBoss Application Server über die Properties Files die unter „jboss-6.1.0.Final\server\default\conf\props“ zu finden sind.

Beispiel:

```
  <application-policy name="web-console">
    <authentication>
      <login-module
       code="org.jboss.security.auth.spi.UsersRolesLoginModule"
       flag="required">
           <module-option name="usersProperties">web-console-
                    users.properties</module-option>
         <module-option name="rolesProperties">web-console-
roles.properties</module-option>
      </login-module>
    </authentication>
  </application-policy>
```

Beispiel für das Login Modul für einen WebService:

```
<application-policy name="JBossWS">
    <authentication>
      <login-module
       code="org.jboss.security.auth.spi.UsersRolesLoginModule"
       flag="required">
            <module-option name="usersProperties">props/jbossws-
              users.properties</module-option>
            <module-option name="rolesProperties">props/jbossws-
              roles.properties</module-option>
            <module-option name="unauthenticatedIdentity">anonymous
             </module-option>
      </login-module>
    </authentication>
</application-policy>
```

2.1.7 Sichere Verbindung via SSL[3]

„Sicher oder nicht Sicher“! Eine sichere Verbindung ist essenziell in unternehmenskritischen Applikationen, aber selbst als sicher erachtete Verfahren und Techniken können Schwachstellen haben, die SSL Sicherheitslücke Heartbleed ist sicherlich ein Beispiel dafür, das Sicherheit oftmals mehr gefühlt als tatsächlich vorhanden ist.

[3] http://docs.oracle.com/javaee/7/tutorial/doc/security-intro006.htm#BNBXW

Secure Sockets Layer (SSL) ist eine Sicherheitstechnologie der Transportschicht und ermöglicht Kommunikation zwischen Web Server und Client über eine sichere Verbindung. Die Daten werden verschlüsselt bevor sie versendet und entschlüsselt bevor sie verarbeitet werden.
SSL beinhaltet folgende Sicherheitsaspekte:

- Authentication
- Confidentiality
- Integrity

2.2 Beispiel: Applikation zur Projektverwaltung

Als Rahmenbedingungen für folgenden Beispielen dient eine Applikation zur Projektverwaltung. Die grafische Benutzeroberfläche wird über den Browser genutzt. Die Web-Komponente wurde mit JSF realisiert.
Es sind folgende Rollen in dieser Hierarchie definiert:

1. **Admin:** Ein Benutzer der Rolle „Admin“ hat Zugriff auf alle Bereich, Funktionen und Informationen. (read and write)
2. **Geschäftsführer:** Der Geschäftsführer hat Zugriff auf die Bereiche:

- Firmen intern (read and write)
- Projekt: über alle Projekte (read)
- Projektmanagement: über alle Projekte (read)
- Mitarbeiterinformation: (read and write)

3. **Projektleiter:** Ein Benutzer mit der Rolle des Projektleiters hat Zugriff auf die Bereiche:

- Firmen intern (read and write)
- Projekt: über alle Projekte in denen er Projektleiter oder Mitarbeiter ist (read and write)
- Projektmanagement: über alle Projekte in denen er Projektleiter ist (read and write)

4. **Mitarbeiter:** Ein Mitarbeiter hat Zugriff auf die Bereiche:

- Firmen intern (read and write)
- Projekt: über alle Projekte in denen er Mitarbeiter ist (read and write)

Bereiche der Applikation:

Login: Dieser Bereich ist öffentlich. Keine Rolle ist notwendig.

Firmen intern: Enthält Funktionen die für jeden Mitarbeiter relevant sind (Stechuhr, Firmen News, ...)

Projekt: Informationen zum Projekt sowie der Zugriff auf die Ressourcen des Projekts.

Projektmanagement: Bietet Funktionen zum Controlling (Reports), Planung und Ressourcenverwaltung

Mitarbeiterinformation: Sämtliche Daten über Mitarbeiter (editierbar) sowie Funktionen für Mitarbeiter spezifische Reports (Krankheitstage, Projektmitgliedschaften, Arbeitsstunden, Gehalt)

Projektteams bestehen aus einem Projektleiter und einer beliebigen Anzahl von Mitarbeitern.

Alle Daten zu den Projekten und Mitarbeitern werden in einer relationalen Datenbank gespeichert.

2.3 Java EE Security Mechanismen

In Java EE wird die Sicherheit von den jeweiligen Containern verwaltet. Die Implementierung der Sicherheitseinstellungen erfolgt über deklarative und programmatisch Techniken. Die Java EE Security Mechanismen können einfach und sicher über Annotationen und Deployment Dispriptors konfiguriert werden. In den folgenden Kapiteln werden wir die Java EE Security Mechanismen etwas genauer beleuchten.

2.3.1 Securing Enterprise Beans[4]

Java Enterprise Beans laufen in einen EJB-Container welcher im Application-Sever (z.B. JBoss oder GlassFish) gehostet werden. Im Allgemeinen dienen sie dazu die Geschäftslogik des Unternehmens abzubilden sowie deren Daten. Es gibt hauptsächlich drei Typen von Beans, die Entity Beans, die Session Beans und die Message Driven Beans. Entity Beans werden genutzt um persitente Daten abzubilden. Session Beans

[4] http://docs.oracle.com/javaee/7/tutorial/doc/security-javaee002.htm#BNBYL

können stateful oder stateless sein und beinhalten Daten eines Benutzers für die jeweilige Session solange die Session aktiv ist. Message Driven Beans bieten die Möglichkeit der asynchronen Kommunikation. Die Möglichkeiten Java Enterprise Beans zu sichern kann man in zwei Kategorien einteilen.

2.3.1.1 Declarative Security

Eine dieser Kategorien ist die deklarative Sicherheit (declarative security) bei der die Sicherheitseinstellungen via Deployment Descriptors oder Annotationen erfolgen. Mit Annotationen können die Zugriffsberechtigungen auf Klassen oder Methoden auf bestimmte Rollen festlegt werden. Voraussetzung für die Nutzung der Annotationen zur Autorisierung ist eine Authentifikation der Benutzer. Der angemeldet Benutzer kann nun alle Klassen und Methoden mit der Annotation `@DeclareRoles` bzw. `@RolesAllowed` aufrufen, die für seine Rolle zugelassen sind. Eine weiter Konfiguration ist über den Deployment Discriptor möglich, der mit dem Namen *application.xml* im Unterverzeichnis *META-INF zu finden ist.*
Zu den wichtigsten Annotationen:
`@DeclareRoles` dient der Auflistung der im Bean verwendeten Rollen. Die Rollen werden im Deployment Discriptor auf die bestehenden Rollen gemappt. Die deklarierten Parameter können innerhalb der gesamten Klasse mit der Methode isCallerInRole(String roleName) überprüft werden.

Beispiel:

```
...
import javax.faces.application.FacesMessage;
...
@ManagedBean(name = "projektManagementBean")
@DeclareRoles
({"Admin","Projektleiter","Geschaeftsfuehrer"})
public class ProjektManagementBean implements
ProjektManagement {
...
```

Für die Klasse ProjektManagementBean wurden die Rollen Admin, Geschäftsführer und Projektleiter deklariert.

Die Annotation `@RolesAllowed` kann auf eine gesamte Klasse oder auf eine und/oder eine beliebige Anzahl an Methoden angewandt werden und legt fest welche Rollen die Klasse bzw. die Methode nutzen dürfen. Auf Klassenebene gibt die Annotation `@RolesAllowed` alle Methoden der Klasse für die angegeben Rollen frei. Zu beachten ist, dass falls sie auf Methodenebene eingesetzt wird, die Klassenebene überschreibt.

Beispiel:

```
import javax.faces.application.FacesMessage;
...
@ManagedBean(name = "projektManagementBean")
@DeclareRoles
({"Admin","Projektleiter","Geschaeftsfuehrer"})
public class ProjektManagementBean implements
ProjektManagement {
...
@RolesAllowed
({"Admin","Projektleiter","Geschaeftsfuehrer"})
public String getControllingReport () {
...
}
...
@RolesAllowed ({"Admin", "Projektleiter"})
public void addProjectMember (Employee employeeToAdd) {
...
}
```

Nur die Rollen Admin, Geschäftsführer und Projektleiter haben Zugriffsberechtigung auf die Methode `getControllingReport()` der Klasse ProjektManagementBean. Allerdings darf in unsere fiktiven Welt der Geschäftsführer keinen Einfluss auf die Teammitglieder der Projekte haben.

Eine weiter Security Annotationen wären `@PermitAll`, welche den Zugriff für alle definierten Rollen erteilt. Sie kann also nicht für einen öffentlichen Bereich verwendet werden, da immer noch eine Authentifikation notwendig ist.

Die Annotation `@DenyAll` verbietet jeglichen Zugriff auf die Klasse bzw. Methode. Diese ist nur sehr selten sinnvoll und zwar wenn durch Vererbung Methoden implementiert werden müssen, die aus Sicherheitsgründen nie aufgerufen werden sollen.
`@PermitAll, @DenyAll` und `@RolesAllowed` können nie gleichzeitig an einer Klasse oder Methode eingesetzt werden.

2.3.1.2 Programmatic Security

Die zweite Kategorie der Autorisierung ist die programmatische Sicherheit (programmatic security) . Sie kommt zum Einsatz, wenn die Implementierung der Geschäftslogik nach den Rollen unterschieden wird. In unserem Beispiel ist dies bei den Report- und Controlling-Funktionen denkbar, da für den Projektleiter andere Kennzahlen wichtiger sind als für den Geschäftsführer. Hier können wir mit der bereits genannten Methode isCallerInRole eine Unterscheidung je nach Rolle vornehmen.
Beispiel:

```
@Stateless
@ManagedBean(name = "projektManagementBean")
@DeclareRoles
({"Admin","Projektleiter","Geschaeftsfuehrer"})
public class ProjektManagementBean implements
ProjektManagement {
...
@Resource
SessionContext context;
@RolesAllowed
({"Admin","Projektleiter","Geschaeftsfuehrer"})
public String getControllingReport () {
     // Erstellen des Reports
     ...
     if(context.isCallerInRole(„Geschaeftsfuehrer“)) {
          //  sensible Daten hinzufügen
          ...
     }
...
}
```

Eine andere Methode die javax.ejb.EJBContext zur Verfügung stellt um die Rolle des gegenwärtigen Nutzers zu prüfen wäre getCallerPrincipal().

2.3.2 Securing Web Applications[5]

Eine der stärken der Java EE Platform ist, dass die Applikation durch Web-Komponenten wie JSF (Java Sever Faces) oder Java Servlets eine einfache Möglichkeit bietet sie auf einem Webserver zu hosten.
Wie auch im Kapitel zuvor können die Sicherheitseinstellungen von Web Applikationen durch Annotationen und Deployment Descriptors erfolgen. Des weiteren bieten sich weiter Optionen zur Konfiguration wenn die Applikation deployed worden ist. Hierbei gilt das die spezifische Einstellungen im Deployment Discriptor eine höherer Gewichtung haben und eventuell die durch Annotationen deklarierten Werte überschreibt.
Die Implementierung der Security für Web Applikationen kann deklarative, programmatisch oder auch Nachrichten-basiert erfolgen.

2.3.2.1 Declarative Security[6]

Wie können wir unsere Http Resourcen schützen? Security Constraints bieten die Möglichkeit Zugriffe auf Http Ressourcen einzuschränken. Falls die Applikation Serlets verwendet kann man Security Constraints durch die Annotationen `@HttpConstraint`, `@HttpMethodConstraint` und `@ServletSecurity` deklarieren.
Werden keine Servlets verwendet, ist es immer noch möglich über den Deployment Descriptor (die web.xml) Security Constraints zu definieren.
`<security-constraint>` können folgende Elemente: beinhalten:

- `<web-resource-collection>` beinhaltet mindestens ein URL Pattern (`<url-pattern>`) auf welches der Zugriff beschränkt werden soll. Optional können Http-Methoden via `<http-method>` bzw. `<http-method-omission>` für das URL pattern restriktiert werden.

[5] http://docs.oracle.com/javaee/7/tutorial/doc/security-webtier002.htm#GKBAA
[6] http://docs.oracle.com/javaee/7/tutorial/doc/security-webtier002.htm#GJJCD

- `<auth-constraint>` legt fest ob eine Authentifikation notwendig ist um auf die vom URL Pattern beschriebenen Seiten zukommen und welche Rollen zugriffsberechtigung haben.
- `<user-data-constraint>` legt fest wie die Datentransfer zwischen Client und Server geschützt ist.

Beispiel:

```
<!-- SECURITY CONSTRAINT  -->
<security-constraint>
    <display-name>Projektmanagement</display-name>
    <web-resource-collection>
       <web-resource-name>projektmanagement</web-resource-name>
       <url-pattern>/projekt/management/*</url-pattern>
       <http-method-omission>GET</http-method-omission>
       <http-method-omission>POST</http-method-omission>
    </web-resource-collection>
    <auth-constraint>
        <role-name>admin</role-name>
        <role-name>projektleiter</role-name>
        <role-name>geschaeftsfuehrer</role-name>
    </auth-constraint>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>
```

In diesem Beispiel haben wir dafür gesorgt, das in unserem Bereich Projektmanagement geschützt ist. Mit `<http-method-omission>` werden ausschließlich die Http-Methoden GET und POST erlaubt und dank `<auth-constraint>` müssen die Anwender eine der drei angegebenen Rollen (Admin, Projektleiter, Geschäftsführer) haben um Zugriff zu bekommen.[7]

2.3.2.2 Authentifizierungsmechanismen[8]

Die in den Security Constraints definierten Rahmenbedingungen werden durch ein Authentifizierungsverfahren realisiert. Der Benutzer muss zunächst Identifiziert werden um überprüfen zu können, ob er Zugriffsberechtigungen auf die von ihm angeforderten Ressourcen hat.

[7] http://docs.oracle.com/javaee/7/tutorial/doc/security-advanced004.htm#BABGEJJJ
[8] http://docs.oracle.com/javaee/7/tutorial/doc/security-webtier002.htm#GKBAA

Die Grundvoraussetzungen für eine Authentifikation sind zum einen Datenbank mit Einträgen zu den Benutzern, ihren Passwörtern und Rollen. Des weiteren müssen der Web-/Applikationsserver dementsprechend konfiguriert sein.
Java EE unterstützt fünf Authentifizierungsmechanismen:

- Basic Authentication
- Form-Based Authentication
- Digest Authentication
- Client Authentication
- Mutual Authentication

Gleich vorab muss erwähnt werden, dass die Basic Authentication sowie Form-Based Authentication nicht mehr heutigen Sicherheitsanforderungen entsprechen. Die Übertragung des Benutzernamen/Passworts ist kaum bis gar nicht verschlüsselt. Abfangen ist in beiden Fällen möglich, da der Zielserver authentifiziert wird. Somit ist es nicht empfehlenswert Basic/Form-Based Authentication ohne Mechanismen wie SSL, Internet Protocol Security oder VPN, die deren Mängel wieder ausgleichen, in einer unternehmenskritischen Applikation einzusetzen.

2.3.2.2.1 HTTP Basic Authentication[9]

HTTP Basic Authentication ist Standardeinstellung der meisten Applikation Server. In der Basic Methode werden Benutzername sowie Passwort des Benutzers mit den Einträgen der Datenbank verglichen um festzustellen ob er die Zugriffsberechtigung auf den Realm hat.
Der Ablauf einer Basic Authentication sieht wie folgt aus:

1. Der Client sendet ein Anfrage auf eine geschützte Ressource
2. Vom Web Server wird eine Dialogbox zurückgesendet, die nach der Eingabe von Benutzername und Passwort verlangt
3. Der Client sendet den Benutzernamen sowie das dazugehörige Passwort an den Web Server
4. Ist die Authentifizierung erfolgreich sendet der Web Server die angeforderten Ressourcen an den Client

[9] http://docs.oracle.com/javaee/7/tutorial/doc/security-webtier002.htm#GKBAA

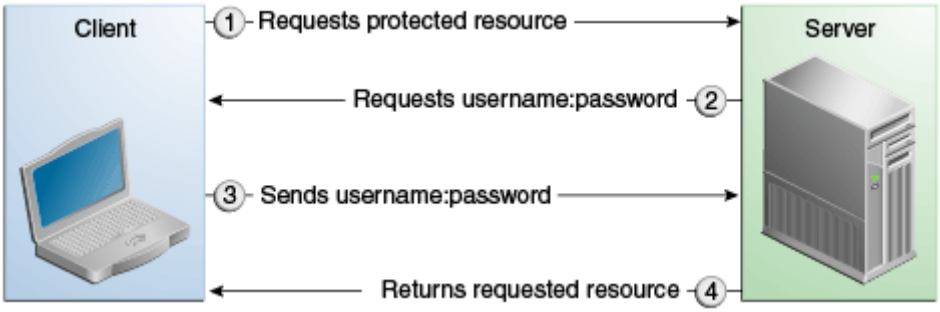

Abb. 1: HTTP Basic Authentication

Quelle: Java EE 7 Tutorial 48.2.2.1 HTTP Basic Authentication

```
<login-config>
    <auth-method>BASIC</auth-method>
</login-config>
```

2.3.2.2.2 Form-Based Authentication[10]

Bei der Form-Based Authentication bestimmt der Entwickler das Aussehen des Login Moduls und der Error Page. Wie bereits erwähnt ist bei dieser Authentifizierungsmethode die Übertragung der Login Daten im Klartext.

Der Ablauf einer Form-Based Authentication sieht wie folgt aus:

1. Der Client sendet einen Request nach einer geschützten Ressource
2. Ist der Client noch nicht authentifiziert wird er auf die Login Seite geleitet. Ist er authentifiziert und autorisiert erhält er Zugriff auf die von ihm angeforderte Ressource
3. Der Client füllt das Login Formular aus und sendet es zum Web Server
4. Der Server versucht den Benutzer zu authentifizieren
    - ist die Authentifizierung erfolgreich, kann überprüft werden ob die Rolle des Benutzers für den Zugriff auf die angeforderte Ressource autorisiert ist. Im Falle einer Autorisierung leitet der Server den Client auf die angeforderte Ressource
    - Falls die Authentifizierung oder Autorisierung fehl schlägt wird der Client auf eine Fehler Seite umgeleitet

[10] http://docs.oracle.com/javaee/7/tutorial/doc/security-webtier002.htm#GKBAA

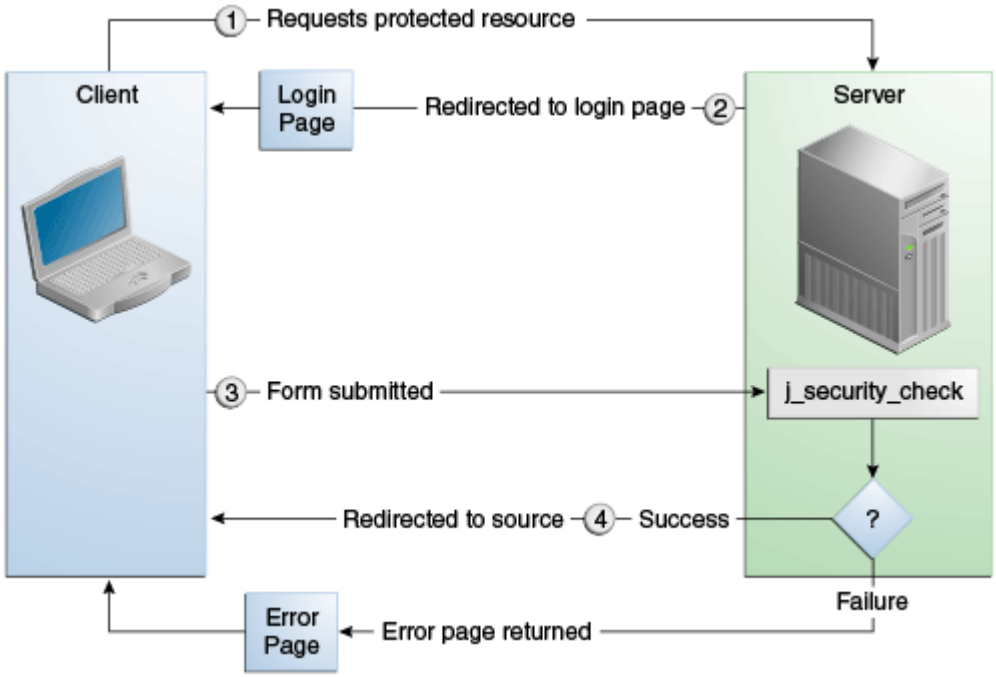

Abb. 2: Form-Based Authentication

Quelle: Java EE 7 Tutorial 48.2.2.2 HTTP Basic Authentication

Dieses Beispiel zeigt wie Form-Based Authentication über den Deployment Descriptor deklariert wird:

```
<login-config>
    <auth-method>FORM</auth-method>
    <realm-name>file</realm-name>
    <form-login-config>
        <form-login-page>/login.xhtml</form-login-page>
        <form-error-page>/error.xhtml</form-error-page>
    </form-login-config>
</login-config>
```

Wenn in der Applikation Form-Based Authetication verwendet werden soll, ist es ratsam Wget Cookies oder SSL Session Information zu verwenden. Auf der HTML Seite des Login Moduls sollte die Action immer auf j_security_check gesetzt werden.

Siehe Beispiel:

```
<form method="POST" action="j_security_check">
<input type="text" name="j_username">
<input type="password" name="j_password">
</form>
```

2.3.2.3 Digest Authentication[11]

Die Digest Athentication benötigt wie auch HTTP Basic und From-Based Authentication einen Benutzernamen sowie ein Passwort. Der entscheidende Unterschied liegt darin, dass bei dieser Authentifikation das Passwort nicht über das Netzwerk geschickt wird sondern stattdessen der Server dem Client eine Zeichenfolge. Beim Client wird ein Hashcode aus dem Benutzername, dem Passwort, der vom Server erhaltenen Zeichenfolge sowie dem Request. Der Hashcode wird vom Client zum Server geschickt , welcher anhand der Prüfsummen die Authentifizierung vornimmt. Der Nachteil ist das der Container Klartextinformationen zur Authentifikation beinhaltet.

2.3.2.3.1 Client Authentication[12]

Bei der Client Authentication nutzt der Web Server das Public Key Zertifikat des Clients zur Authentifizierung. Diese Methode zur Authentisierung ist sicherer als HTTP Basic und From-Based Authentication. Client Authentication nutzt HTTP über SSL (HTTPS). Die SSL bietet die nötige Verschlüsselung, die Server Authentisierung sowie Message Integrity (fortlaufende Nummerierung der Datenpakete).
Um Client Authentication nutzen zu können muss im Deployment Discriptor unter <auth-method> CLIENT-CERT stehen:

```
<login-config>
    <auth-method>CLIENT-CERT</auth-method>
</login-config>
```

2.3.2.3.2 Mutual Authentication[13]

Bei der Mutual Authentication werden authentifizieren sich der Client und der Server gegenseitig. Dies geschieht entweder Certificate-Based (Server und Client identifizieren sich mit Zertifikaten) oder Username/Password-Based (Server identifiziert sich durch ein Zertifikat und der Client nutzt Benutzername/Passwort) .

Der Ablauf bei Certificate-Based Mutual Authentication sieht wie folgt aus:

1. Der Client sendet einen Request nach einer geschützten Ressource
2. Der Web Server sendet sein Zertifikat an den Client

[11] http://docs.oracle.com/javaee/7/tutorial/doc/security-webtier002.htm#GKBAA
[12] http://docs.oracle.com/javaee/7/tutorial/doc/security-advanced002.htm#GLIEN
[13] http://docs.oracle.com/javaee/7/tutorial/doc/security-advanced002.htm#GLIEN

3. Der Client überprüft das Zertifikat des Web Servers
4. Ist die Authentifikation des Web Servers erfolgreich, sendet der Client sein eigenes Zertifikat an den Web Server
5. Nun überprüft der Server das Zertifikat des Client
6. Ist die Authentifikation des Client erfolgreich, erlaubt der Server dem Client den Zugriff auf die Ressourcen

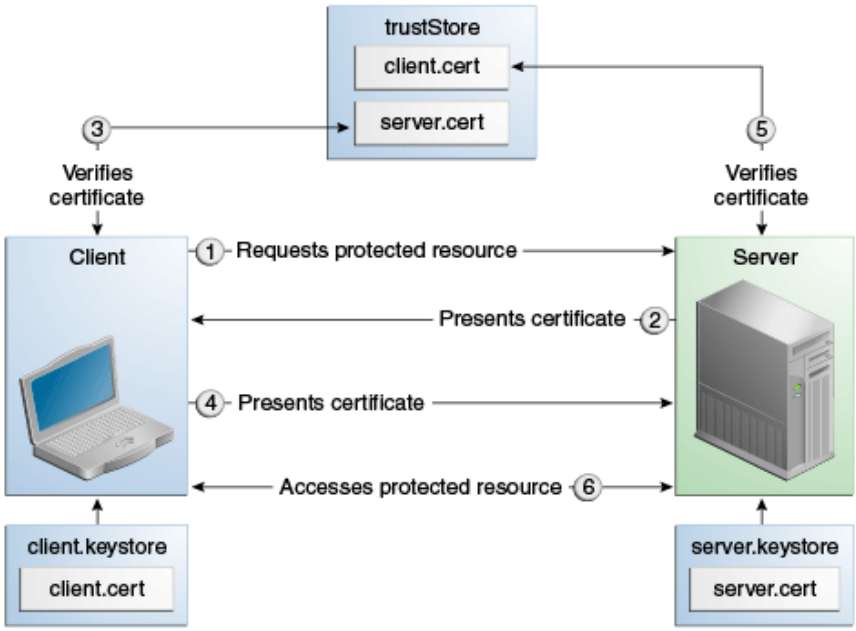

Abb. 3: Certificate-Based Mutual Authentication

Quelle: Java EE 7 Tutorial 50.2.2 Mutual Authentication

Der Ablauf bei Username/Password-Based Mutual Authentication sieht wie folgt aus:

1. Der Client sendet einen Request nach einer geschützten Ressource
2. Der Web Server sendet sein Zertifikat an den Client
3. Der Client überprüft das Zertifikat des Web Servers
4. Ist die Authentifikation des Web Servers erfolgreich, sendet der Client den Benutzernamen sowie das Passwort an den Web Server
5. Nun überprüft der Server das die Zugangsdaten des Client
6. Ist die Authentifikation des Client erfolgreich, erlaubt der Server dem Client den Zugriff auf die Ressourcen

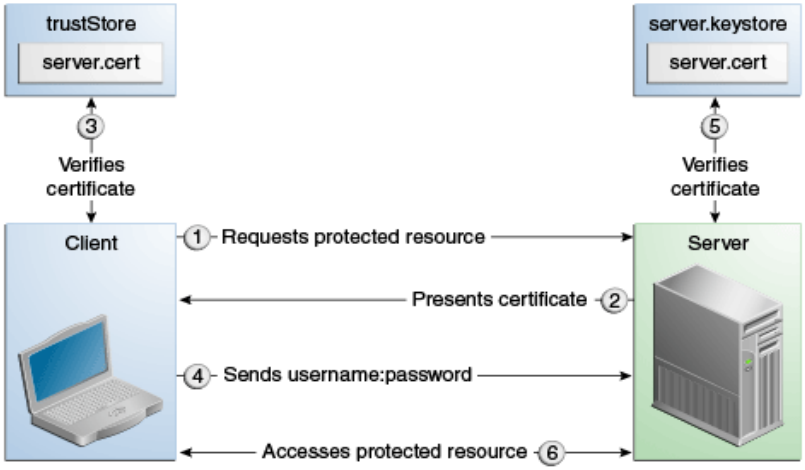

Abb. 4: Username/Password-Based Mutual Authentication

Quelle: Java EE 7 Tutorial 50.2.2 Mutual Authentication

Es gibt mindestens zwei Möglichkeiten Mutual Authentication über SSL zu ermöglichen. Die von vielen bevorzugte Methode erfolgt über den Deployment Discriptor (hier die web.xml). Die Einstellung auf CLIENT-CERT sorgt dafür das beidseitige Authentifizierung notwendig ist bevor Zugang auf geschützte Ressourcen möglich ist.
Eine weniger genutzte Methode ist die <clientAuth> Property im certifiate Realm auf true zu setzen. Es sollte beachtet werden das nur eine der Methoden verwendet wird sonst wird die Client Authentifikation doppelt ausgeführt.

2.3.2.4 Programmatic Security[14]

Programmatische Sicherheit wird angewandt wenn deklarative Sicherheitsmaßnahmen nicht ausreichen um das Sicherheitsmodell der Applikation abzubilden.
Die folgenden Methoden des Interfaces HttpServletRequest ermöglichen die Authentifizierung von Benutzern für Web Applikationen:

- authenticate: Sendet eine Authentisierungsanfrage an den Nutzer. Eine Login Dialog Box wird angezeigt um die Eingabe des Benutzername und Passwort zu ermöglichen
- login: Erlaubt der Applikation Benutzername und Passwort einzusammeln. Diese Methode stellt eine Alternative zu der Deklarierung von Form-Based Authentication im Deployment Discriptor dar
- logout: Reset der bereits bestimmten Identität des Benutzers

[14] 48.3.1 Authenticating Users Programmatically
http://docs.oracle.com/javaee/7/tutorial/doc/security-webtier003.htm#GJIIE

Hier ein Beispiel zur Methode authenticate:

```
package com.example.test;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class TestServlet extends HttpServlet {

    protected void processRequest(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            request.authenticate(response);
            out.println("Authenticate Successful");
        } finally {
            out.close();
        }
    }
```

Quelle: Java EE Tutorial 48.3.1 Authenticating Users Programmatically

Servlet 3.1 stellt Methoden zur Verfügung um zugriff auf die Zugangsinformationen des Callers zu bekommen. Diese sollten allerdings nur in spezifischen Situationen verwendet werden, da es eigentlich die Aufgabe des Containers wäre die Zugriffsberechtigungen zu verwalten.
Diese Methoden sind:

- getRemoteUser: Liefert den Caller oder null
- isUserInRole: Überprüft die Rolle des Users
- getUserPrincipal: Liefert den „principal name“ und ein java.security.Principal Objekt oder null falls der User nicht authentifiziert ist

2.4 Eigene Maßnahmen gegen Bedrohungen

Wie bereits in der Einleitung erwähnt ist es notwendig, die existierenden Bedrohungen für unser Webanwendung zu kennen um effektive Gegenmaßnahmen zu entwickeln. OWASP (Open Web Application Security Project) ist ein Community Projekt das sich zum Ziel gesetzt hat, Unternehmen und Organisationen zu unterstützen, sichere Anwendung zu entwickeln. Es werden jährlich die „Top 10 Risiken für die Anwendungssicherheit“ veröffentlicht.

Die Top 10 Liste[15] aus dem Jahr 2013 sah folgendermaßen aus:

1. Injection
2. Fehler in der Authentisierung und Session-Management
3. Cross-Site Scripting (XSS)
4. Unsichere direkte Objektreferenzen
5. Sicherheitsrelevante Fehlkonfiguration
6. Verlust der Vertraulichkeit sensibler Daten
7. Fehlerhafte Autorisierung auf Anwendungsebene
8. Cross-Site Request Forgery (CSRF)
9. Benutzen von Komponenten mit bekannten Schwachstellen
10. Ungeprüfte Um- und Weiterleitungen

So betrachten wir mal den Platz 1 der Liste. Das wichtigste um Injections zu verhindern ist die Kontrolle der Benutzereingaben. Eine von OWASP Lösungsvorschlägen gegen SQL Injection sind Prepared Statements. Eine andere aber aufwendige Möglichkeit wären sorgfältige Validierung aller Eingaben. Aber man kann auch mit einfachen grundsätzlichen Überlegungen die Applikation härten. Ist das Eingabefeld unbedingt notwendig? Spinner, Radio-Buttons und diverse andere Elemente können gar nicht von Injections betroffen sein, daher wäre es die Überlegung wert eventuell auf etwas Usability zu Gunsten der Sicherheit zu verzichten.
Es gibt noch unzählige weiter Möglichkeiten Web-Applikationen zu härten.

[15] https://www.owasp.org/index.php/Germany/Projekte/Top_10_fuer_Entwickler-2013/Inhaltsverzeichnis

3 Fazit / Ausblick

Obwohl die Sicherheit von unternehmenskritischen Applikationen bereits ein ein essenzielles Thema ist, wird es in Zukunft von noch weit größerer Bedeutung sein. Die Wirtschaft ist heute so stark abhängig von dem Zugang zu Informationen und deren Auswertung wie noch nie zuvor und dieser Trend wird sich auch weiter fortsetzen.
Absolute Sicherheit gibt es nicht! Selbst Technologien die im Allgemeinen als zuverlässig und sicher gelten können Sicherheitslücken aufweisen. Der berühmte Fall des Heartbleed-Bugs in der OpenSSL-Implementierung der Heartbeat-Erweiterung zeigt dies nur zu gut. Niemand kann genau beziffern welcher finanzielle Schaden allein durch diesen Fehler aufgetreten ist. Der Schutz durch Standard Lösungen ist umfassend und wird ständig aktualisiert, aber man muss bedenken durch den hohen Verbreitungsgrad der Gegenseite ein großes und lohnendes Ziel bietet. Individuelle Sicherheitsmaßnahmen sind schwerer zu knacken, da die Informationen über das Sicherheitskonzept schwerer zugänglich sind und somit weniger herkömmliche Ansatzmöglichkeiten aufweisen. Allerdings müssen diese von Experten entwickelt und gewartet werden, da ein unvollständiger Schutz fatale Auswirkungen haben kann. Das bringt ein großes Problem der ungleichen finanziellen Möglichkeiten mit sich. Kleine und mittelständische Unternehmen können unmöglich den benötigten Sicherheitsetat aufbringen um ihre Innovationen und Informationen vor Industriespionage durch fremde Regierungen oder kriminelle Strukturen zu schützen. Ich denke, dass in Zukunft die Sicherheit der eigenen Applikationen und Daten immer größeren Anteil an dem langfristigen Erfolg des Unternehmens haben wird.

4 Literaturverzeichnis

Primärquellen

JavaEE 7 Tutorial URL: http://docs.oracle.com/javaee/7/tutorial/doc/partsecurity.htm#GI-JRP [April 2014]

OracleJava EE Management and Security Technologies Homepage von ORACLE URL: http://www.oracle.com/technetwork/java/javaee/tech/management-139662.html [September 2013]

OWASP (Open Web Application Security Project) URL: https://www.owasp.org/index.php/Germany/Projekte/Top_10_fuer_Entwickler-2013/Inhaltsverzeichnis

JBoss Enterprise Application Plattform 6.1 Security Guide URL: https://access.redhat.-com/site/documentation/en-US/JBoss_Enterprise_Application_Platform/6.1/pdf/Security_Guide/JBoss_Enterprise_Application_Platform-6.1-Security_Guide-en-US.pdf