Evaluierung medizinischer Studiensysteme

Lars Hupel, Michael Kerscher, Marek Kubica

24. September 2011

Inhaltsverzeichnis

1 Studiensysteme **3**
1.1 openCDMS . . . 3
1.1.1 Technologien . . . 3
1.1.2 Authentifizierung . . . 3
1.1.3 Authorisierung . . . 5
1.1.4 Rechteverwaltung . . . 6
1.1.5 Pseudonymisierung/Anonymisierung . . . 6
1.2 openClinica . . . 6
1.2.1 Technologien . . . 7
1.2.2 Authentifizierung . . . 7
1.2.3 Rechteverwaltung . . . 8
1.2.4 weitere Eigenschaften . . . 9
1.3 OBiBa . . . 10
1.3.1 Onyx . . . 10
1.3.2 Opal . . . 11
1.4 Vergleich der Systeme . . . 13

2 Sicherheitsframeworks **14**
2.1 JAAS . . . 14
2.1.1 Authentifizierung . . . 14
2.1.2 Authorisierung . . . 14
2.1.3 Einschätzung . . . 15
2.2 SAML . . . 16
2.2.1 Authentifizierung . . . 16
2.2.2 Authorisierung . . . 18
2.2.3 Einschätzung . . . 18
2.3 Spring Security . . . 19
2.3.1 Mögliche Ebenen der Zugriffsbeschränkung . . . 19
2.3.2 Zugriffsbeschränkung auf Objektebene . . . 20
2.3.3 Beispielszenario . . . 21
2.4 Shiro . . . 22
2.4.1 Authentifizierung . . . 22
2.4.2 Authorisierung . . . 23
2.4.3 Web-Unterstützung . . . 23
2.4.4 Einschätzung . . . 24
2.5 HERAS-AF . . . 24
2.5.1 XACML . . . 24
2.5.2 Implementation . . . 25
2.5.3 Einschätzung . . . 25

1 Studiensysteme

1.1 openCDMS

openCDMS (vormals *PsyGrid*) ist eine Sammlung von Tools, die es ermöglichen klinische Studien anzulegen, zu bearbeiten und zu analysieren. Es wird ebenfalls ein Tool mitgeliefert um neue Datensätze anzulegen und zu bearbeiten.

Im Gegensatz zu den anderen, im folgenden betrachteten Systemen war es leider nicht möglich, *openCDMS* lokal zu installieren, da die Installationsroutine offenbar fehlerhaft ist und ein Versuch das System manuell zu bauen scheiterte an zahlreichen Fehlern beim Kompilationsvorgang. Daher wurden für diese Evaluation nur der Quelltext, die Dokumentation sowie die öffentlich verfügbare Demontrations-Applikation hergezogen.

1.1.1 Technologien

Die Applikation ist in einer 3-Tier Architektur aufgebaut, die Dokumentation bezeichnet sie als Client, Logik und Daten. Dabei wird die Datenhaltung über eine von Hibernate unterstützte Datenbank abgewickelt. Sie beinhaltet unter anderem die Speicherung der Studien, Anonymisierung, Authorisierung. User werden hingegen gegen einen LDAP-Server authentifiziert. Der Logik-Tier, aufbauend auf Tomcat verarbeitet diese Daten und stellt sie dem Client-Tier bereit. Diese Kommunikation verläuft über eine SSL-geschützte HTTP-Verbindung mittels SOAP.

Die Clients, von denen je nach Aufgabe verschiedene benutzt werden sind hingegen als Java-Swing-Applikationen realisiert und können via Java-Web-Start aus dem Browser heraus gestartet werden. Dies hat zwar zur Folge, dass der Benutzer Java installiert haben muss, aber hat bedeutende Sicherheitsvorteile gegenüber Web-Applikationen bringt. So kann man typische Szenarien wie XSS-Lücken sowie CSRF-Angriffe von vorhinein vermeiden.

1.1.2 Authentifizierung

Authentifizierung läuft im gesamten System über die *Security Assertion Markup Language*, einem XML-Dialekt, der auf SOAP aufbaut. Die Prüfung der Berechtigungen (SAML Assertions) erfolgt in der Policy Authority (PA) im Zusammenspiel mit der Attribute Authority (AA) im Logik-Tier serverseitig. Wenn Berechtigungen fehlen wird eine entsprechende SAML-Exception weitergegeben.

Der Login in die Datensammlungs-Applikation läuft ab, indem in `org.psygrid.collection.entry.Launcher.launchLoginDialog` für den Login ein `EntryLoginService` erstellt wird, der in `authenticate()` schließlich auf den Applikations-übergreifenden `org.psygrid.securitymanager.security.SecurityManager` zugreift – dieser ist die zentrale Stelle in

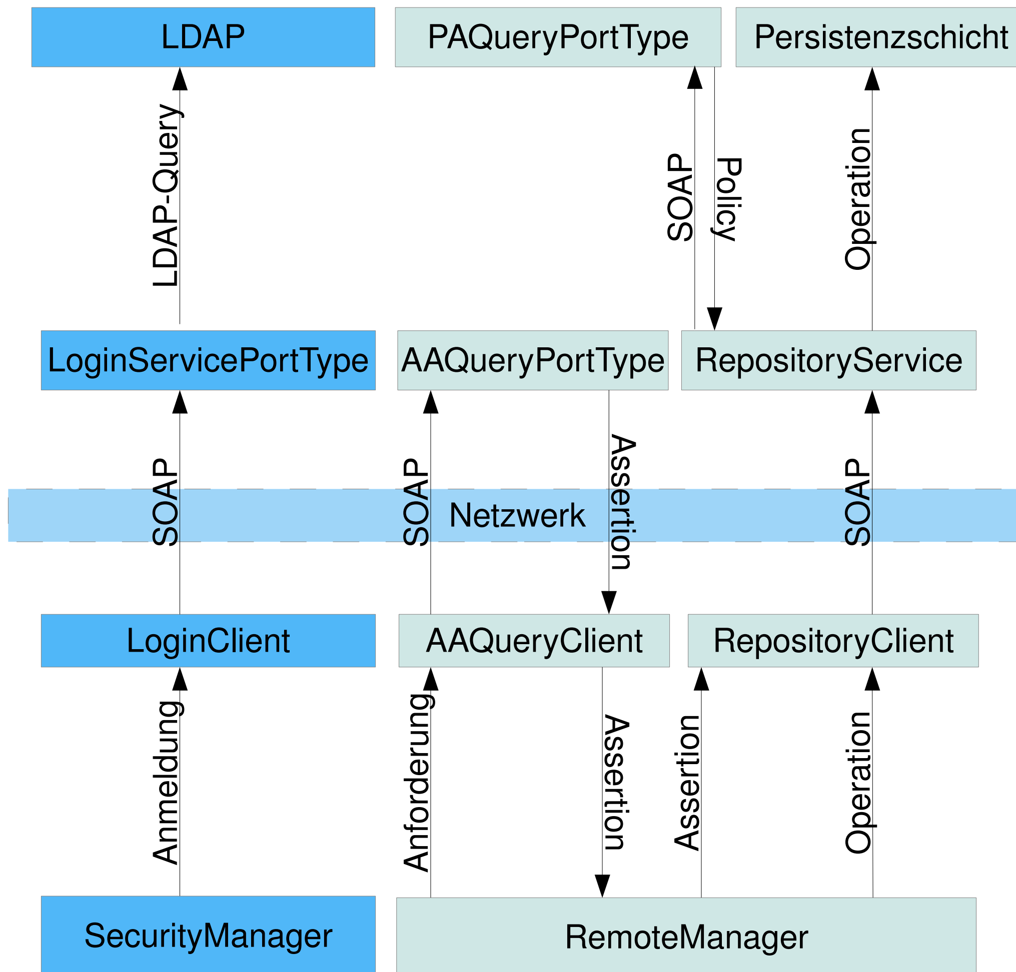

Abbildung 1: Vereinfachte Darstellung der Authentisierung und Authorisierung, Server ist oben, Client unten

openCDMS, in der Authentifikation gehandhabt wird. Im Zuge von `SecurityManager.login()` wird das Passwort in `refreshKey()` verwendet um dort via `doRemoteLogin()` die entsprechenden X.509-Zertifikatsdaten vorzubereiten und die weitere Authentifizierung an den `LoginClient` weiterzuleiten. Dieser ruft dann auf einer SOAP-Gegenstelle namens `LoginServicePortType` die Methode `login()` mit dem Usernamen und Passwort auf, die den User authentifizieren soll. Bei Fehlern werden Exceptions geworfen, so dass der Client-Part feststellen kann dass die Authentifizierung nicht geklappt hat.

Dabei kann die Authentifizierung nicht nur an einer falschen Benutzername/Passwort-Kombination scheitern, das System sieht ebenfalls vor, dass Accounts gesperrt sein können. Zugriffe mit unbekanntem Usernamen werden ebenfalls Server-seitig geloggt wie auch falsche Passwörter, dem Client wird diese Information natürlich aus Sicherheitsgründen nicht übermittelt. Positiv ist auch, dass im Client versucht wird, das Passwort die meiste Zeit verschlüsselt im Speicher anzulegen, so dass man nicht das Klartext-Passwort im Speicher stehen hat, allerdings ist die Verschlüsselung symmetrisch und kann wieder entschlüsselt werden.

Serverseitig wird der Login dann weiter zum `AuthenticationManager` geleitet, dieser prüft in `checkUID()` ob der User im angeschlossenen LDAP existiert und kann in `login()` den User einloggen – im Erfolgsfall werden Credentials zurückgegeben, die an den Client geschickt werden. Der Client speichert die Credentials in seinem `KeyStore`.

1.1.3 Authorisierung

Bei dem Anlegen neuer Datensätze wird im Collection-Tool nach der Einwilligung des Probanden gefragt. Dabei wird die Datenbank vom Client modifiziert, daher ist es interessant zu sehen wie ein Änderungsvorgang abläuft, um zu sehen ob und wie der Vorgang authorisiert wird. Die Einwilligung wird in `org.psygrid.collection.entry.ui.EditConsentDialog.doConsentChange()` editiert, sowohl der lokal im Client vorgehaltene Datensatz, als auch der Remote-Datensatz, der für die Betrachtung der Authorisierung relevant ist. Dabei wird der eigentliche Editionsvorgang an den sogenannten `RemoteManager` delegiert, der generell für alle Remote-Datenänderungen, nicht nur Einwilligungen, zuständig ist. Er ist das Equivalent vom `SecurityManager` bei der Authentitizierung.

So wird zunächst der editierte Datensatz genommen und via `getSAMLAssertion()` eine SAML-Assertion für das spezielle Projekt angefordert. Diese Anforderung wird an den SecurityManager weitergeleitet, der solche Anfragen über den AttributeAuthorityQuery-Client (`AAQueryClient`) beantwortet. Im `AAQueryClient` werden die Attribute des Users von der Attribute Authority via SOAP abgefragt (`AttributeAuthorityQueryPortType.getAttributesForUserInProject`) welche die Anfrage mit einer passenden SAML-Assertion beantwortet.

Nachdem der Client nun seine passende SAML-Assertion bekommen hat, wird die Daten-Änderungs-Operation zusammen mit der SAML-Assertion an den `RepositoryClient`

(`org.psygrid.data.repository.client.RepositoryClient`) weitergegen, der nun die Operation sowie die SAML-Assertion via SOAP an den `RepositoryService` schickt. Serverseitig wird daraufhin in `checkPermissionsByGroup`/`checkPermissionsByProject` (`org.psygrid.data.utils.service.AbstractService`) geprüft, ob die SAML-Assertion gültig ist und die entsprechende Aktion authorisiert, im Beispiel `ACTION_DR_ADD_CONSENT`. Dazu wird `org.psygrid.security.accesscontrol.AccessEnforcementFunction.authorizeUser()` aufgerufen, welches schließlich über das SOAP-Interface `org.psygrid.security.policyauthority.PolicyAuthorityQueryPortType.makePolicyDecision()` anfragt ob der User einer Rolle zugehört bei der diese Action erlaubt wird. Die exakte Zuordnung von Rollen zu Actions geschieht in der Policy. Nachdem nun bekannt ist, dass dem User erlaubt ist, die Änderung durchzuführen leitet der `RepositoryService` die Operation an die Persistenzschicht weiter, die sie in die Datenbank übernimmt.

1.1.4 Rechteverwaltung

Bei dem Anlegen eines Users erhält man die Möglichkeit die Rollen des neuen Users festzulegen. Die Auswahl enthält die Rollen ChiefInvestigator, ProjectAdministrator, ProjectManager, PrincipalInvestigator, ClinicalResearchManager, ClinicalResearchOfficer, ScientificResearchOfficer, TreatmentAdministrator, RecruitmentManager, DataAnalyst, NamedInvestigator. Diese Rollen sind fest im System verankert und können nicht verändert werden. Zusätzlich existieren die Rollen DELViewer, DELAuthor, DELCurator (zum Zugriff auf die *Data Element Library*), StudyPatcher, DataImporter, QueryData, ViewIdentity (zum Ansehen der Identitäten der Probanden) sowie Pharmacist. Jeder dieser Rollen (`RBACAction`) wird eine Menge von Aktionen (`RBACAction`) zugeordnet, deren Existenz beim Ausführen von Aktionen geprüft wird.

1.1.5 Pseudonymisierung/Anonymisierung

Probanden bekommen vom System eine Identifikationsnummer wie etwa *OLK/001001-217*, wobei der letzte Teil innerhalb einer Studie einfach inkrementiert wird.

1.2 openClinica

openClinica ist ein Tool, welches dazu dient, klinische Tests zu verwalten und durchzuführen. Hierbei ermöglicht das Tool, eine Studie zu definieren und im Anschluss die Messdaten der Studienteilnehmer einzutragen. Es eignet sich dabei auch für größere Studien, die über mehrere Standorte verteilt sind, was sich auch im Berechtigungsmodell widerspiegelt.

Es ist modular aufgebaut und in einen *core* sowie dem *web application* und *web service* Modul unterteilt. Die Installation der Webapplikation ist mit dem Deployment via Tomcat und einer Anpassung der Konfiguration recht schnell erledigt.

1.2.1 Technologien

Um Benutzer zu authentifizieren wird das Springframework verwendet. Im Hintergrund werden die Daten entweder in einer Postgres- oder eine Oracle-Datenbank gespeichert. Die Webapplikation läuft auf der Tomcat-Plattform.

1.2.2 Authentifizierung

Momentan kann sich ein Benutzer nur über ein normales Passwort authentifizieren, jedoch würde Spring auch andere Methoden zulassen, was in openClinica jedoch noch nicht implementiert ist. In der Konfiguration existiert schon ein Attribut *auth*, allerdings mit dem Kommentar, dass momentan außer `password` nichts unterstützt wird – LDAP-Support ist geplant [1].

Ablauf der Authentifizierung:

1. Browser greift auf beliebige Seite zu
2. Springframework definiert in `/classes/org/akaza/openclinica/applicationContext-security.xml` *security interceptors* für alle Seiten außer Bilder, Anmeldeseiten, Kontaktseiten etc. Zugriff auf jegliche Seiten außerhalb der definierten Ausnahmen sind nur mit der Spring-Rolle `ROLE_USER` möglich. Diese Rolle erhält man mit der Accounterstellung und wird in der Tabelle `authorities` gespeichert.
   - Anmeldung erfolgt mit Benutzername und Passwort. Die Authentifizierung erfolgt u. A. in `OpenClinicaUsernamePasswordAuthenticationFilter`.
   - Ein Anmeldevorgang, egal ob positiv oder negativ, wird in `audit_user_login` mit Datum und Uhrzeit vermerkt.
3. Im Anschluss wird die Kontrolle an das jeweilige Servlet (definiert in `WEB-INF/web.xml`) übergeben
   - diese sind von `SecureController` abgeleitet welcher ein `mayProceed()` und ein `processRequest()` fordert
   - `SecureController.processRequest()` initialisiert die benötigten Datenstrukturen wie Benutzerdaten, aktive Studie und Rolle und ruft am Ende `mayProceed()` auf
   - `mayProceed()` muss mit einer Methode überschrieben werden, die den jeweiligen User auf entsprechende interne OpenClinica-Rollen überprüft
   - die `processRequest()`-Methode des Servlets wird nur aufgerufen, wenn der Benutzer eine gültige Rolle besitzt, ansonsten wird eine Fehlerseite zurückgegeben

[1] https://wiki.openclinica.com/doku.php?id=developerwiki:security-authentication

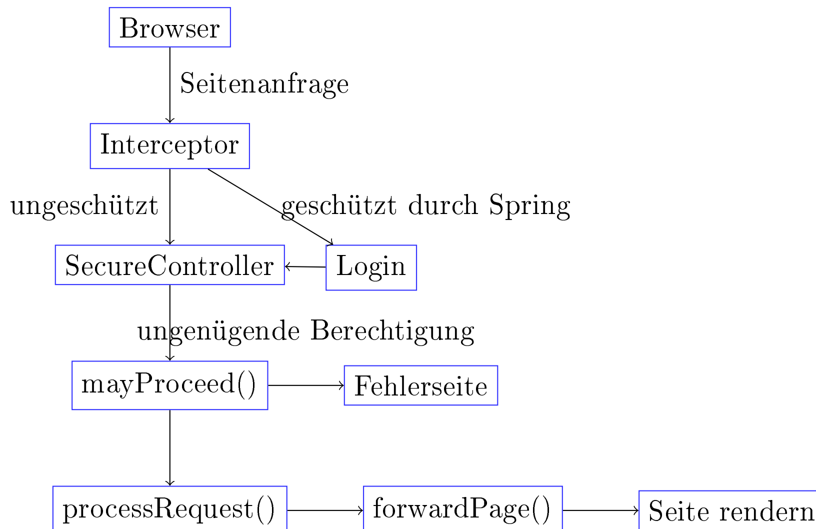

Abbildung 2: Login-Vorgang von openClinica

- in den JSP-Seiten ist konditionaler Code enthalten der je nach aktiver Rolle nur die passenden Menüpunkte anzeigt

Bei einem erneuten Aufruf einer Seite (zum Beispiel durch einen Klick auf einen Link oder ein Formular) wird die Session erneut überprüft bevor der Aufruf wieder an das zuständige Servlet durchgeleitet wird. Wie beim ersten Aufruf, wird auch bei allen folgenden Aufrufen in `mayproceed()` geprüft, ob die Berechtigungen für eine Aktion vorhanden sind.

1.2.3 Rechteverwaltung

openClinica unterstützt rollenbasierte Rechte auf Basis vordefinierter und unveränderlichen Rollen welche im Quellcode verankert sind. Das System bietet die Rollen *Admin, Coordinator, Studydirector, Investigator, Researchassistent, Monitor* an.

Um auf eine Studie auf verschiedene Arten zuzugreifen (submit data, extract data, manage study, monitor) werden die Rollen verwendet. Die Funktionen, welche die Berechtigungen festlegen sind hierbei fest im Quellcode verankert und nicht veränderbar.

Die Rollen können für jeden Benutzer studienabhängig vergeben werden, allerdings kann ein Benutzer nur eine Rolle pro Studie einnehmen. Weiterhin gibt es das Konzept der Trennung von *study*- und *site*-Rollen. Eine site-Rolle dabei im Gegensatz zur study-Rolle nur die Rechte auf Daten für eine site wohingegen eine study-Rolle die jeweiligen Rechte auf Daten von allen zugeordneten sites ausüben kann.

Es existieren zusätzlich zu den *user roles* auch *user types*. Diese Benutzertypen sind orthogonal zu den Rollen und können auch unabhängig von den jeweiligen Rollen vergeben

werden. Es gibt als Typen normale *user, business administrator* und *technical administrator*[2].

Granularität Die Rechteverwaltung ist nicht darauf ausgelegt, die Berechtigungen auf Objektebene zu verwalten. Geprüft wird nur in der Funktion `mayproceed()`, welche anhand der aktiven Rolle (studienbezogen) den Zugriff auf ein Servlet verwehrt oder gewährt. Wenn der Benutzer Zugriff bekommt, kann er dieses Servlet auch vollständig nutzen. Eine Einschränkung der Rechte auf individuelle Objekte im System ist dabei nicht vorgesehen. So wird beispielsweise in `UpdateStudySubjectServlet` nicht geprüft, ob der individuelle Benutzer trotz seiner Rolle in der Studie überhaupt die Berechtigung hat, dieses `StudySubject` überhaupt zu verändern.

Flexibilität Da Berechtigungen fest in `mayproceed()` verankert sind hat dies zur Folge, dass bei einer Änderung einer Rolle alle betroffenen Java-Klassen überprüft werden müssen und gegenenenfalls manuell korrigiert werden. Ein zentraler Ansatz, der festlegt, welche Rollen auf diverse Tätigkeiten zugreifen dürfen ist nicht vorhanden. Da die Kontrolle aller Klassen nur manuell machbar ist, existiert hier eine potentielle Fehlermöglichkeit, falls eine Klasse übersehen wird oder sich Tippfehler einschleichen.

Ein zentraler, eventuell hierarchischer Berechtigungsansatz könnte hier Fehlerpotential einsparen und gleichzeitig ein feingranulareres Berechtigungsmodell ermöglichen.

1.2.4 weitere Eigenschaften

Die endgültigen Seiten die an den Benutzer ausgeliefert werden sind in JSP geschrieben. Die darin angezeigten Daten stammen aus den jeweiligen Servlets die diese an die Render-Engine übergibt. In den JSP-Dateien befindet sich Code, welcher an manchen Stellen auf die hartkodierten Rechte von openClinica verweist und diese nutzt um zu entscheiden, welche Daten angezeigt werden. Dadurch ist die Darstellung relativ stark an die Logik gebunden. Bei einer Veränderung der Rechte müssen nicht nur in den jeweiligen Java-Servlets die Rollen geändert werden sondern auch in allen betroffenen JSPs. Beispielsweise befindet sich in `menu.jsp` folgende Abfrage:

```
1 <c:if test="${userRole.investigator || userRole.
     researchAssistant}">
2 ...
3 </c:if>
```

Durch die hartkodierten Rollen kann das Berechtigungsmodell nur sehr aufwändig verändert werden wenn man verhindern will, dass Probleme oder ungewünschte Ergebnisse auftreten.

[2] https://docs.openclinica.com/3.1/openclinica-user-guide/overview-user-types-and-roles

1.3 OBiBa

Das *OBiBa*-Projekt, welche für den Einsatz in Biobänken[3] gedacht ist, besteht aus mehreren, voneinander größtenteils unabhänigen Softwarepaketen. Relevant für diese Evaluation sind dabei *Onyx* und *Opal*.

1.3.1 Onyx

Onyx ist der Datenerfassungsteil von OBiBa. Der Hauptbestandteil ist dabei eine Webanwendung, über welche vordefinierte *Questionnaires* ausgefüllt werden können. Diese Fragebögen müssen von der einsetzenden Institution programmiert werden

Eine vorkonfigurierte Version von Onyx existiert nicht, da, nach Angaben auf der Webseite, man mit den Entwicklern Kontakt aufnehmen soll, um eine angepasste Version zu erhalten. Eine selbstkompilierte Variante war jedoch nicht lauffähig, da einige Klassen zum Ausführen fehlen, daher basiert diese Betrachtung nur auf Dokumentation und Quelltext. (Das Beispielprojekt war ebenfalls nicht lauffähig.)

Onyx verwendet ein Rechtesystem bestehend aus genau vier Rollen, *system administrator*, *questionnaire editor*, *participant manager* und *data collection operator*. Diese Rollen sind in `opal-core` hartkodiert. Die einzelnen Rechte für diese Rollen können im User Guide[4] nachgelesen werden. Die Authentifizierung läuft manuell über die Datenbankverbindung; hierfür wird – im Gegensatz zu Opal – nicht Shiro verwendet (siehe unten). Auch sind die Rollen nicht veränderbar.

Der Login-Vorgang läuft wie folgt ab:

- über das `LoginPanel` werden Benutzername und Passwort abgefragt und an die `authenticate`-Methode in `OnyxAuthenticatedSession` weitergereicht
- ein eigener `EntityQueryService` fragt in der Datenbank nach dem Benutzernamen und prüft das Passwort
- die zugeordneten Rollen werden automatisch von der Java-Persistence-API in ein Objekt abgelegt, welches in der Wicket-Session gespeichert wird

Zugriffe auf die Administrationsoberfläche werden anschließend von Wicket über Annotationen überprüft, z. B.

```
@AuthorizeAction(action = "RENDER", roles = { "SYSTEM_ADMINISTRATOR" })
```

Darüber hinausgehender Zugriffsschutz, z. B. auf Ebene einzelner Fragebögen, ist nicht ersichtlich.

[3] Datenbank von biologischem Material, z. B. DNA-oder Blutproben

[4] http://wiki.obiba.org/download/attachments/11599994/OnyxUserGuide_1-8.pdf?version=1&modificationDate=1293187542000

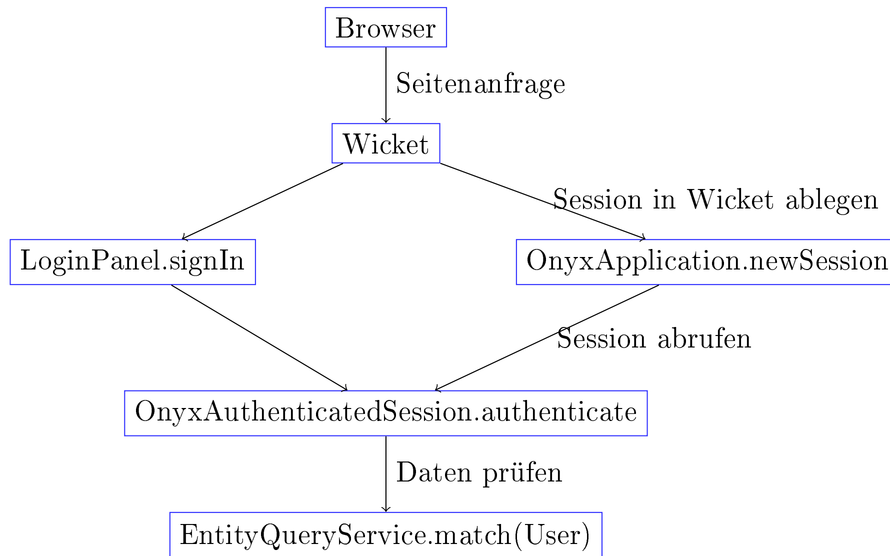

Abbildung 3: Login-Vorgang von Onyx

1.3.2 Opal

Opal ist der Datenbankbestandteil von OBiBa. Das Datenmodell von sieht eine Aufteilung in mehrere Einheiten vor, wobei in Opal mehrere Datenquellen zusammen laufen.

Opal ist eine Anwendung mit zwei Backends: Weboberfläche per Google Web Toolkit und Jetty und Shell-Zugriff per Apache SSH-Server. Zur Authentifikation wird Apache Shiro verwendet. Installation und Konfiguration sind recht einfach durchzuführen; es existiert auch ein Paket für Debian-Systeme.

Wie bereits genannt, besteht das Gesamtsystem aus sogenannten *Units*, wobei Opal selbst eine Unit darstellt. Innerhalb der Opal-Unit werden Personen (*participants*) durch einen *private identifier* identifiziert, welcher nicht öffentlich ist, d. h. nicht vom System aus nach außen gelangt. Um einen Datenaustausch zu gewährleisten, wird ein *shared key* sowie eine Abbildung auf die entsprechenden privaten Schlüssel benötigt. Diese Abbildung wird in einer separaten Datenbank (*identifiers database*) abgelegt, welche sich auch auf einem getrennten Server befinden kann (intern wird dies durch zwei verschiedene JDBC-Datenquellen realisiert). In dieser gesonderten Datenbank liegen außerdem persönliche Informationen des jeweiligen Teilnehmers, während in der *data database* anonyme Daten abgelegt werden. Durch diese Maßnahme soll sichergestellt werden, dass nur tatsächlich benötigte Daten ausgetauscht werden können, wodurch ein Angreifer Zugriff auf beide Datenbanken benötigt.

Der Datenaustausch mit einer spezifischen Unit findet in zwei Stufen statt: Zunächst wird aus dem *keystore* das Schlüsselpaar für diese Unit geladen und die zu importierende Datei im Speicher entschlüsselt. Jeder Datensatz ist nun mit dem gemeinsamen Schlüssel

zwischen Opal und der anderen Unit identifiziert. In der identifiers database wird dieser nun nachgeschlagen und, falls existent, auf den private identifier abgebildet. Falls ein noch nicht bekannter gemeinsamer Schlüssel auftritt, wird eine neuer privater Schlüssel zufällig erzeugt. Im zu importierenden Datensatz vorliegende private Daten werden ebenfalls in der identifiers database abgelegt, der Rest wird in der data database gespeichert[5].

Die Daten werden innerhalb von *Tabellen*, die zu *Datenquellen* zusammengefasst sind, verwaltet. Jede Tabelle definiert eine Reihe an *Variablen*, die neben einem Typ auch weitere Attribute enthalten kann (so z. B. die entsprechende Fragestellung und Antwortmöglichkeiten in einem Fragebogen). Jeder Datensatz in einer solchen Tabelle gehört zu einer *Entität*; üblicherweise eine Person bzw. ein Teilnehmer, der durch den privaten Schlüssel identifiziert wird.

Opal verwendet ein rollenbasiertes Rechtesystem. Sowohl einzelnen Nutzern als auch Gruppen können Leserechte auf gesamte Tabellen einer Datenquelle vergeben werden, aber auch feingranular auf einzelne Variablen. Darüber hinaus werden *views* unterstützt, die ähnlich wie SQL-Views arbeiten und eine bestimmte Sicht auf die Daten erlauben.

Umgesetzt wird die Authorisierung von Shiro, welches auch die Sessionverwaltung umfasst. Der hierfür relevante Code befindet sich in den Teilprojekten `opal-core` und `opal-core-ws`. Opal definiert mehrere *Realms* für die Authentifizierung per HTTP-Header oder per Cookie. Die Benutzeraccounts selbst können per Shiro-Konfiguration einerseits statisch definiert werden, andererseits aber auch an z. B. LDAP angebunden werden, wodurch auch dieser Teil auf einem getrennten Server liegen kann.

[5] http://wiki.obiba.org/display/OPALDOC/General+Concepts#GeneralConcepts-ParticipantIdentifierSeparation

1.4 Vergleich der Systeme

	openCDMS	openClinica
Technologien	OpenSAML, SOAP	Spring, Java Servlets, JSP
Rechteverwaltung	Rollenbasiert, Unterstützung für Projekt-spezifische Berechtigungen. Rollen fest einprogrammiert	Rollenbasiert, Rollen, Typen und Berechtigungen fest einprogrammiert
Authentifizierung	LDAP	Spring sowie intern verwaltete Rollen
Anonymisierung	-	-
Pseudonymisierung	Probanden werden generell über generierte Identifier referenziert	Muss manuell pseudonymisiert werden

	OBiBa/Opal	OBiBa/Onyx
Webadresse	http://www.obiba.org/	
Entwickler	http://www.p3g.org/ (internationale gemeinnützige Organisation), aktiv	
Technologien	Apache Shiro	Hibernate
Rechteverwaltung	rollenbasiert, Zugriffskontrolle feingranular auf Tabellenebene; Rechtevergabe für Tabelle, Variablen und Views	rollenbasiert, fixe Rollen
Authentifizierung	komplett über Shiro	eigener Service, unterstützt durch Wicket
Anonymisierung	Datensätze werden innerhalb einer Unit ausschließlich über anonyme private Schlüssel identifiziert	
Pseudonymisierung	Zuordnung von Datensätzen zu persönlichen Daten über gemeinsamen Schlüssel, Zuordnung kann auf getrenntem Server abgelegt werden	

2 Sicherheitsframeworks

2.1 JAAS

Der Java™ Authentication and Authorization Service (*JAAS*) ist, wie der Name andeutet ein Framework für die Authentisierung von Usern sowie Authorierung von Handlungen. Im Gegensatz zu anderen Frameworks ist seit 1.4 JAAS Teil der Java Standard Edition und daher auf jeder modernen Java-Platform. Für ältere Systeme wie 1.3 war *JAAS* noch optional. So erklärt sich der geringere Funktionsumfang der eingebauten Lösung im Vergleich zu 3rd-Party-Lösungen, die mehr auf die Anforderungen der einzelnen Applikationen eingeben können.

2.1.1 Authentifizierung

Bei der Authentifizierung wird eine Instanz eines `LoginContext` gebildet, welcher optional einen Callback-Handler enthält. In diesem Callback kann der Programmierer angeben, wie er auf die entsprechenden Anfragen des `LoginModule` reagieren will. Anfragen sind zum Beispiel `NameCallback` oder `PasswordCallback`, miit dem das `LoginModule` nach dem Namen resp. Passwort des Users fragen kann. Dadurch wird die Authentisierung von der Präsentation entkoppelt. Die eigentliche Authentifikation findet daraufhin in den `LoginModule`n statt. So wäre ein `LoginModule` für LDAP denkbar, ebenso Anmeldung via SQL oder einfachen Dateien. Java liegen LoginModule für JNDI, Keystores, Kerberos 5, Login an Windows NT-Systemen sowie Login an Unix-Systemen bei. Weitere Module können ebenfalls vom Programmierer geschrieben werden.

Die Konfiguration der `LoginModule` findet in einer speziellen, externen Konfigurationsdatei statt, in der man festlegen kann welche `LoginModule` notwendig sind, welche optional sowie ihnen etwaige debug-Parameter übergeben. Diese Datei wird der JVM entweder beim Start als Parameter übergeben oder in der `java.security`-Datei der JVM eingetragen.

Nach der Erstellung eines solchen `LoginContext` kann nun die Methode `login()` aufgerufen werden, die im Fehlerfall eine `LoginException` wirft.

2.1.2 Authorisierung

JAAS nutzt für die Authentifizierung das Konzept von *Subjects* und *Principals.* Nach dem Login kann man den `LoginContext` nach dem eingeloggten *Subject* befragen. Mit diesen Daten ist es möglich Code als dieses eingeloggte *Subject* auszuführen, indem man für die auszuführende Handlung eine Klasse von `java.security.PrivilegedAction` ableitet. Die API ist ähnlich zu der Threading-API, so existiert eine `run`-Methode die ausgeführt wird wenn man die Ausführung der Handlung mit `Subject.doAsPrivileged(subjectInstanz,`

`actionInstanz, null)` einleitet. Zudem können dem *Subject* beim Login sogenannte *Principals* zugeordnet werden, welche Rollen entsprechen.

In der `run()`-Methode kann man nun beliebigen Code schreiben, wobei während der Ausführung in einer Policy-Datei geprüft wird, ob der Code ausgeführt werden darf. Die Policy-Datei wird, ähnlich wie die LoginModule-Konfiguration ebenfalls der JVM übergeben. In ihr können Berechtigungen feingranular vergeben werden, je nachdem aus welcher Codebase (JAR) der auszuführende Code stammt, welcher User (*Subject*) oder Rolle (*Principal*) den Code ausführen will, welche Funktionen aufgerufen werden dürfen. Allerdings müssen dazu die aufzurufenden Funktionen entsprechende Permissions bereitstellen und prüfen. Bei den Klassen des JVMs ist dies der Fall, bei eigenen Klassen muss man selbst Permissions definieren.

2.1.3 Einschätzung

Obwohl *JAAS* Teil der Java-Umgebung ist, ist es für die Verwendung in einem Clinical-Trial-System nicht optimal:

- Die Konfigurationsformate sind ad-hoc und nicht sonderlich intuitiv. Gerade bei längeren Policies wird das schreiben von feingranularen Regeln sehr mühsam, zudem auch noch separat Policies in Java implementiert werden müssen.
- Es ist keine Drop-In-Lösung, erfordert eine Reihe von Konfigurationsdaten und Anpassung von JVM-Parametern. Wenn zwei Libraries JAAS gleichzeitig verwenden sollen kommt es zu Problemen welche Konfigurationsdateien genutzt werden sollen.
- Es ist stark auf Desktop-Applikationen ausgelegt, so bietet es zwar flexible Regeln zur Kombination von verschiedenen Codebases, dieses ist aber bei Web-Applikationen völlig unnötig
- Die LoginModule sind nicht annähernd so umfassend wie bei anderen Frameworks, es würde wohl notwendig sein eigene LoginModule für JAAS zu schreiben
- Es bietet keinerlei Schutzmechanismen vor üblichen Web-Problemen wie Cross-Site-Request-Forgery, Cross-Site-Scripting oder Injections, weil sich JAAS auf einer wesentlich tieferen Abstraktionsschicht befindet. Es wäre nötig, das alles selbst zu Implementieren oder auf ein anderes Framework zurückzugreifen, womit dann der Sinn JAAS überhaupt einzusetzen fraglich ist
- Die Dokumentation ist eher minimalistisch.

2.2 SAML

SAML, die *Security Assertion Markup Language* ist, im Unterschied zu den anderen Security-Frameworks keine Implementation von Sicherheitsfeatures sondern ein industriestandard von OASIS (Organization for the Advancement of Structured Information Standards). Als solcher konkurriert SAML mit anderen Protokollen wie etwa OpenID.

In der Welt von *SAML* existieren User, sogenannte *principals*, Authentisierungs-Provider (*identity provider*) sowie *service provider*, die die Dienste bereitstellen auf die die *principals* zugreifen wollen. Der typische Ablauf von SAML ist es, dass der *principal* eine Aktion anfordert, diese Anfrage vom *identity provider* beglaubigt wird und als sogenannte *Assertion* an den *service provider* weiterleitet, der je nach seinen Richtlinien entscheidet ob der authentisierte Benutzer authorisiert ist, diese Aktion durchzuführen und im Erfolgsfall diese Aktion durchführt.

SAML besteht aus einer Reihe von komponenten die zusammenspielen. Der Datenaustausch erfolgt über einen speziellen XML-Dialekt, SAML. Die Integrität der Assertions wird durch XML Signatures sichergestellt, so dass die Assertions nicht vom *principal* manipuliert werden können. Zusätzlich können vertauliche Teile der Assertions verschlüsselt werden, um keine geheimen Informationen preiszugeben. Die Kommunikation erfolgt vornehmlich über SOAP auf HTTP, obwohl SAML über Bindings auch weitere Kommunikationswege unterstützt.

2.2.1 Authentifizierung

Bei der Authentifizierung steht es dem *identity provider* frei, wie er den User authentifiziert. Er kann dabei beliebige Verfahren nutzen, SAML spricht sich nicht darüber aus, wie die Authentifizierung stattzufinden hat. Bei erfolgreicher Authentifizierung stellt der *identity provider* für den User eine *Assertion* aus.

Diese *Assertion* kann verschiedene Arten von *Statements* enthalten:

1. Authorisierungs-Statements: Wer hat wen wie wann authorisiert?
2. Attributs-Statements: Schlüssel-Wert-Paare die die Nutzdaten transportieren, etwa was der *principal* für eine Aktion ausführen will
3. Authorisierungs-Entscheidungs-Statements: Zusatzbedingungen für die Authorisierung, etwa wie lange die Assertion gültig sein soll. SAML spezifiziert diese nicht im Detail, für komplexere Szenarios wird auf *XACML* verwiesen

Eine beispielhafte *Assertion* ist in Listing 4 dargestellt. Diese teilt dem *service provider* mit, von wem authentisiert wurde (example.com), wer authentisiert wurde (alice@example.com), für wen diese authentifikations gilt (example2.com) sowie die Mitteilung über eine Telefonnummer (+1-888-555-1212).

```
1  <Assertion ID="_a75adf55-01d7-40cc-929f-dbd8372ebdfc"
2     IssueInstant="2003-04-17T00:46:02Z" Version="2.0"
3     xmlns="urn:oasis:names:tc:SAML:2.0:assertion">
4     <Issuer>
5        example.com
6     </Issuer>
7     <Subject>
8        <NameID
9          Format=
10         "urn:oasis:names:tc:SAML:1.1:nameid-format:
              emailAddress">
11         alice@example.com
12       </NameID>
13       <SubjectConfirmation
14         Method="urn:oasis:names:tc:SAML:2.0:cm:sender-vouches
              "/>
15    </Subject>
16    <Conditions NotBefore="2003-04-17T00:46:02Z"
17                NotOnOrAfter="2003-04-17T00:51:02Z">
18       <AudienceRestriction>
19          <Audience>
20             example2.com
21          </Audience>
22       </AudienceRestriction>
23    </Conditions>
24    <AttributeStatement>
25       <saml:Attribute
26    xmlns:x500=
27      "urn:oasis:names:tc:SAML:2.0:profiles:attribute:X500"
28    NameFormat=
29      "urn:oasis:names:tc:SAML:2.0:attrname-format:uri"
30    Name="urn:oid:2.5.4.20"
31    FriendlyName="telephoneNumber">
32          <saml:AttributeValue xsi:type="xs:string">
33                +1-888-555-1212
34          </saml:AttributeValue>
35       </saml:Attribute>
36    </AttributeStatement>
37 </Assertion>
```

Abbildung 4: Unsignierte *SAML-Assertion*

2.2.2 Authorisierung

Der *service provider* empfängt die *Assertion* und entscheidet nach seiner Policy ob er diese Aktion zulässt oder ablehnt. Zusätzlich hat der *service provider* ebenfalls die Möglichkeit Daten vom *identity provider* über drei Arten von *Queries* zu empfangen:

1. Authentication query
2. Attribute query
3. Authorization decision query

Diese entsprechen den Feldern aus der *Assertion* und werden üblicherweise über SOAP ausgetauscht.

Weitere Vorgaben wie die Authorisierung abläuft macht SAML nicht. Die Attribute sind ebenfalls applikationsspezifisch und sollen auf der applikationsebene gehandhabt werden.

2.2.3 Einschätzung

SAML ist ein interessanter Ansatz für das Delegieren von Authentisierung sowie Authorisierung. Der Einsatz von Signierung sowie Verschlüsselung ermöglicht sichere Übertragung, allerdings die Tatsache dass das Attributsformat keine spezifizierte Semantik hat, macht es schwierig sinnvoll interoperable *identity provider* zu haben, die ensprechende *attribute queries* an die *service provider* weitergeben.

Allerdings ist es bei SAML natürlich noch notwendig sich für eine Implementation zu entscheiden. Eine mögliche Wahl aus dem Open-Source-Umfeld ist etwa OpenSAML, jedoch existieren auch eine Reihe von SAML-Implementationen von IBM, Sun, VeriSign, RSA etc. Zudem bietet SAML eigentlich nur ein Protokoll zum übermitteln von Authentisierungs-Nachrichten, die tatsächliche Authorisierung muss auf Applikationsebene gemacht werden, selbiges gilt für die Authorisierungs-Entscheidungen, diese müssen auch von der Applikation vorgenommen werden. Daher wäre, wenn man die Authentifizierung und Authorisierung nicht selbst implementieren will ein Applikationsframework wie Shiro oder Spring Security notwendig.

Weiterhin hat SAML in letzten Jahren zunehmend Konkurrenz von OpenID[6] für die Authentisierung sowie OAuth[7] im Web-Umfeld bekommen. *SAML* ist ein wesentlich breiteres Protokoll, dass mit dem Web Browser SSO Profil auch Single-Sign-On unterstützt, dadurch aber auch umfangreicher im Vergleich zu OpenID/OAuth. Zudem letztere immer breitere Unterstützung finden, u.a. von Twitter, Yahoo, AOL, Google und VeriSign muss man sich fragen, ob man für einen auf das Web begrenzten Service nicht lieber auf die auf REST besierende Konkurrenz setzt. Spring Security, als Applikationsframework

[6] https://openid.net/
[7] http://oauth.net/

bietet Integration mit OAuth an, daher ist dies ein schwerwiegendes Argument wenn man sowohl einfache und integrierte Authentisierung will als auch Interoperabilität mit *service providern*.

2.3 Spring Security

Spring Security kann benutzt werden um Benutzer zu authentifizieren und für unterschiedliche Aktionen authorisieren. Dabei bietet das Framework viele Möglichkeiten wie Authentifizierung anhand normaler Benutzernamen und Passwörter, LDAP, OpenID, etc. an. Auch für Webapplikationen bietet es eine Fülle von Funktionalität wie Session Management, Remember Me Funktionalität etc.

2.3.1 Mögliche Ebenen der Zugriffsbeschränkung

Der Zugriff auf Seiten, Methoden oder Ressourcen kann dabei auf mehreren Ebenen geregelt werden. Die Spring Referenz [8] beschreibt eine ausdrucksbasierte Zugriffskontrolle (Expression-Based Access Control) welche einen sehr feingranularen Zugriffsfilter bietet. Dabei kann man zum einen schon auf der Ebene von URLs erste Rechteprüfungen anstellen und den Zugriff auf diese Ressourcen komplett verwehren.

```
1 <http use-expressions="true">
2   <intercept-url pattern="/admin*"
3     access="hasRole('admin') and hasIpAddress
        ('192.168.1.0/24')"/>
4     ...
5 </http>
```

Eine weitere Möglichkeit um Berechtigungen auf Methodenebene zu prüfen gibt es mit `@Pre` und `@Post` Annotationen. Dazu können mit Ausdrücken Regeln erstellt werden die erfüllt werden müssen um die Methode aufzurufen.

```
1 @PreAuthorize("hasRole('ROLE_USER')")
2 public void create(Contact contact);
```

Es ist weiterhin sogar möglich, nicht nur ganze Methodenaufrufe zu filtern sondern anhand der übergebenen Objekte zu entscheiden, ob der aktuelle Benutzer die Rechte besitzt um dieses Objekt mit dieser Methode zu verarbeiten. Dabei kann nicht nur auf die Parameter zugegriffen werden sondern auch noch auf andere eingebaute Ausdrücke wie `authentication` welches Zugriff auf das Authentifizierungsobjekt des aktuellen Sicherheitskontexts bietet. Um innerhalb solcher Ausdrücke auf Parameter zugreifen zu können, muss der Code mit Debug-Informationen kompiliert werden.

[8] http://static.springsource.org/spring-security/site/docs/3.1.x/reference/el-access.html

```
1 // Zugriff nur erlauben, wenn der angemeldete Benutzer admin-
     Rechte fuer das Contact-Objekt hat
2 @PreAuthorize("hasPermission(#contact, 'admin')")
3 public void deletePermission(Contact contact, Sid recipient,
     Permission permission);
4
5 // nur Zugriff auf Contact-Object des angemeldeten Benutzers
     erlauben
6 @PreAuthorize("#contact.name == authentication.name")
7 public void doSomething(Contact contact);
```

Mit `PostFilter` Ausdrücken können zurückgegebene Collections anhand der angegebenen Regeln gefiltert werden. Dabei sollte allerdings vermieden werden viele Elemente aus einer langen Liste herauszufiltern da dies viel Zeit in Anspruch nehmen kann. Stattdessen sollte versucht werden bereits innerhalb der Methode nur erlaubte Werte in die Collection einzutragen.

2.3.2 Zugriffsbeschränkung auf Objektebene

Man kann den Zugriff auf Objekte durch Überprüfen der jeweiligen Rolle oder auch über das Vorhandensein der notwendigen Berechtigungen (permissions) limitieren. Diese Permissions können pro Benutzer und Objekt vergeben werden und reichen von Standard `BasePermissions` über selbsterstellte Permissions die von `AbstractPermission` abgeleitet sind [9].

Verwaltet werden solche Berechtigungen durch ACEs (AccessControlEntry) in ACLs (AccessControlList) die es pro Objekt nur einmal gibt. Implementiert wird dieses Konzept mittels vier Tabellen in der Datenbank. Dieses Schema muss vor Benutzung in der Datenbank angelegt werden damit es von Spring verwendet werden kann.

ACL_SID SID steht für *security identity*. Diese Tabelle enthält in einer Spalte die `Principals` (meist Benutzer) und `GrantedAuthoritys` und in einer zweiten Spalte ein Flag das kennzeichnet welche der beiden Arten die Zeile darstellt.

ACL_CLASS enthält alle Domänenobjekt-Klassennamen zusammen mit einer ID. Für jede Klasse für die ACL-Berechtigungen vergeben werden existiert eine Zeile.

ACL_OBJECT_IDENTITY speichert Informationen über die eindeutigen Domänenobjekte. Die Tabelle enthält als Spalten eine ID, einen Fremdschlüssel auf die `ACL _CLASS`-Tabelle, einen eindeutigen Bezeichner welche das Objekt kennzeichnet, einen Fremdschlüssel auf `ACL_SID` um den Besitzer der Objektinstanz zu speichern und ein Flag, ob ACL-Einträge von übergeordneten ACLs Berechtigungen erben können.

[9] http://static.springsource.org/spring-security/site/docs/3.0.x/reference/domain-acls.html

ACL_ENTRY speichert die individuellen Berechtigungen für jeden Rechte*empfänger* (SID). Dazu gibt es als Spalten einen Fremdschlüssel auf `ACL_OBJECT_IDENTITY` und auf `ACL_SID`, ein Flag welches kennzeichnet ob ein Audit erstellt werden soll und eine Spalte mit einem 32 Bit Integer in dem die Berechtigungen als Bits kodiert sind.

Standardmäßig stehen die die ersten vier Bits für Lesen (Bit 0), Schreiben (Bit 1), Erstellen (Bit 2), Löschen (Bit 3) und Administrieren (Bit 4). Die restlichen Bits können über eine eigene `Permission`-Instanz erweitert werden falls dies notwendig sein sollte.

Der folgende Beispielcode zeigt auf, wie ein Objekt, welchem wir die ID 42 (es kann auch der Hash oder irgendetwas eindeutiges verwendet werden) geben so gesichert wird, dass ein Principal mit dem Namen Samantha das Objekt administrieren darf.

```
1  // Prepare the information we'd like in our access control
   entry (ACE)
2  ObjectIdentity oi = new ObjectIdentityImpl(Foo.class, new
   Long(42));
3  Sid sid = new PrincipalSid("Samantha");
4  Permission p = BasePermission.ADMINISTRATION;
5
6  // Create or update the relevant ACL
7  MutableAcl acl = null;
8  try {
9    acl = (MutableAcl) aclService.readAclById(oi);
10 } catch (NotFoundException nfe) {
11   acl = aclService.createAcl(oi);
12 }
13
14 // Now grant some permissions via an access control entry (
   ACE)
15 acl.insertAce(acl.getEntries().length, p, sid, true);
16 aclService.updateAcl(acl);
```

2.3.3 Beispielszenario

Um SpringSecurity zum Absichern eines klinischen Studienverwaltungssystems zu verwenden, wird hier ein Beispielszenario entworfen und das Berechtigungsmodell mit SpringSecurity abgebildet.

Ein klinisches Studienverwaltungssystem soll mehrere Studien verwalten können. Diese Studien können jeweils auf mehrere Standorte (Sites) aufgeteilt sein. Das System verwaltet Patientendaten welche vor unerlaubtem Zugriff oder Veränderung geschützt werden

müssen. Dazu werden dem Klinikpersonal Berechtigungen für die einzelnen Daten zugewiesen.

Es soll ein Berechtigungssystem mit Spring realisiert werden womit folgende Eigenschaften erfüllt werden:

Patienten können nur vom Ersteller verändert werden Damit nur der Ersteller schreibend auf seine Patienten zugreifen kann, werden ACLs genutzt und der jeweilige eintragende Benutzer mit Administratorrechten für diesen Patienten ausgestattet.

globale Rechte Für das Personal das Zugriff auf alle Patientendaten benötigt soll es möglich sein, alle Daten einer Studie von allen Standorten auszulesen. Dazu kann die Vererbung von Rechten benutzt werden. Es werden ACEs für das jeweilige Studienobjekt eingetragen und dieses Studienobjekt als übergeordnetes Objekt (parent) der jeweiligen Standorte (studienspezifisch) genutzt. Die Patienten wiederrum können die ACLs von den Standort-ACLs vererbt bekommen

Standortbegrenzte Leserechte Für Monitoringzwecke soll es möglich sein, dass Rollen existieren die lesend auf alle Patienten des jeweiligen Standortes zugreifen können. Um diese Rechte auf die Standorte einzuschränken, können dem Benutzer die Berechtigungen auf das jeweilige Standortobjekt (studienspezifisch) erteilt werden. Durch die Vererbung der ACLs können diese Rechte auch für die Patienten genutzt werden.

individuelle Rechte Über ACEs die vom Objekteigentümer vergeben werden können ist es möglich anderen Benutzern Berechtigungen zu erteilen.

2.4 Shiro

Shiro ist ein Apache-Projekt und ein Framework, welches Methoden zu Authentifizierung, Authorisierung, Kryptographie und Session Management anbietet. Es arbeitet unabhängig von der Art der Anwendung, erlaubt also sowohl Desktop- als auch Webanwendungen. Für letzteres gibt es weitergehende Unterstützung, z. B. durch automatisches Setzen von Cookies.

2.4.1 Authentifizierung

Mittelpunkt der Sicherheitsarchitektur ist ein *security manager*, der zu Beginn ein *token*, welches die Anmeldedaten des Nutzer enthält, entgegennimmt. Dieses wird dann an einen *authenticator* weitergeleitet, in der Regel der `ModularRealmAuthenticator`. Dieser wiederum ist in der Lage, eine Authentifizierung gegen einen oder mehrere *realms* durchzuführen, derer da wären:

- JDBC (Benutzername und Passwort sind in der Datenbank abgelegt)

- LDAP
- ActiveDirectory
- Textdatei
- applikationsspezifisch (selbst implementiert)

Falls mehrere Realms spezifiziert sind, kann über eine eigene `AuthenticationStrategy` festgelegt werden, ob z. B. eine einzelne positive Antwort ausreichend ist, oder ob sämtliche Realms die Anmeldedaten akzeptieren müssen.

Am Ende der Authentifizierung steht ein Objekt der Klasse `AuthenticationInfo`, welches Zugriff auf sämtliche *principals* (ein Objekt, welches einen Benutzer repräsentiert) bietet, die von den Realms ermittelt worden sind.

2.4.2 Authorisierung

Shiro ordnet jedem Benutzer eine Menge an *Permissions* zu. Diese können beliebig definiert werden; eine Möglichkeit dafür sind *WildcardPermissions*. Diese bieten eine hierarchische Form von Berechtigungen an, z. B. `participant:edit,view:*` (repräsentiert die Berechtigung, sämtliche Teilnehmer betrachten und bearbeiten zu können). Berechtigungen können in einer Konfigurationsdatei vordefiniert , aber auch zur Laufzeit vergeben werden; außerdem sind beliebig viele Ebenen in der Hierarchie möglich. Die Berechtigungen werden den Nutzern über Rollen zugeordnet.

Überprüft werden können diese Berechtigungen einfach via einer der überladenen Varianten von `isPermitted`. Alternativ ist noch eine automatische Überprüfung via Annotationen möglich, was aber die Einbindung eines AOP-Frameworks nötig macht.

Intern wird die Prüfung wieder über die Realms realisiert, wobei der standardmäßig verwendete `ModularRealmAuthorizer` die Berechtigung bejaht, wenn mindestens ein Realm dies tut.

2.4.3 Web-Unterstützung

Per Filter lässt sich Shiro in die `web.xml` einbinden. In der Konfiguration können dann URL-Muster und zugehörige Zugriffsrechte angegeben werden, So kann z. B. folgendes festgelegt werden:

```
1 /participant/*/view = authc, ssl, roles[dataEntry]
```

Dies legt fest, dass, um den angegebenen Pfad aufrufen zu können, der Benutzer authorisiert (eingeloggt) sein muss, die Seite per SSL aufruft und die Rolle „dataEntry“ innehat. Es ist ferner möglich, eine URL zu konfigurieren, unter der ein Login möglich ist, so dass nicht eingeloggte Benutzer weitergeleitet werden können. Ebenso wird ein Redirect auf SSL automatisch erledigt.

2.4.4 Einschätzung

Shiro eignet sich aus folgenden Gründen für den Einsatz in einem Clinica-Trial-System:

- Es erlaubt sehr feingranulare Zugriffsberechtigungen, so dass es problemlos möglich ist, Sites und Nutzer voneinander zu unterscheiden. So könnte mit Berechtigungen wie `clinic-a:study-a:participant:view:*` festgelegt werden, dass ein Nutzer in der Studie A in der Klinik A alle Teilnehmerdaten betrachten darf.
- Shiro bietet viele Mechanismen standardmäßig und ohne Konfigurationsaufwand an, z. B. Sessionmanagement. Dadurch verringert sich der Implementierungsaufwand deutlich.
- Der Einsatz für Webanwendungen ist vorgesehen, was sich u. a. durch die direkte Integrierbarkeit in die `web.xm` äußert. Grobe Zugriffsbeschränkungen können dadurch mit wenig Aufwand direkt realisiert werden, was das Risiko von unberechtigten Zugriffen minimiert.
- Es wird eine große Zahl an *authenticators* mitgeliefert. Sollten diese nicht ausreichend sein, ist es einfach, einen selbst zu implementieren.

Ebenso wie bei JAAS führt Shiro keine weitergehenden Schutzmaßnahmen gegen übliche Sicherheitslücken durch.

2.5 HERAS-AF

Die Abkürzung *HERAS-AF* steht für *Holistic Enterprise-Ready Application Security – Architecture Framework* und stellt die Referenzimplementation von *XACML* dar (*Extensible Access Control Markup Language*). Derzeit besteht es nur aus einem Kern, welcher XACML 2.0 implementiert (d. h. sogenannte *access requests* auswerten kann). Weitergehende Komponenten wie z. B. das Management von *policies* stehen noch nicht zur Verfügung.

2.5.1 XACML

XACML ist ein XML-Dialekt, mit welchem sowohl Zugriffsrechte als auch deren Interpretation festgelegt werden.

Das typische Anwendungsszenario läuft dabei so ab, dass ein Benutzer eine Anfrage an eine durch einen *PEP* (*Policy Enforcement Point*) geschützte Ressource stellt. Der PEP sendet dann eine XACML-Anfrage mit den Daten des Zugriffs (Benutzer, Ressource, Aktion, etc.) an einen *PDP* (*Policy Decision Point*). Dieser entscheidet dann anhand von *policies*, ob der Zugriff gewährt werden kann und sendet eine entsprechende Antwort an den *PEP*.

Dieser Entscheidung zu Grunde liegen *PolicySets*, welche aus *Policies* bestehen, die einzelne *Rules* enthalten. Jede einzelne Regel trifft eine konkrete Entscheidung, die dann von verschiedenen *Combining Algorithms* zu einer Gesamtentscheidung zusammengefasst werden (z. B. *Deny Overrides*, der bei einer einzigen Teilverweigerung den Zugriff verweigert).

Die Domänen, auf welchen Regeln entscheiden, sind sogenannte *Targets* und *Attributes*. Targets bestehen aus Bedingungen für Benutzer, Ressource und Aktion, welche festlegen, ob eine Regel überhaupt auf einen Anfrage anwendbar ist. Attributes sind einfache Schlüssel-Wert-Paare, angereichert mit Zeit- und Ausstellungsinformationen, anhand derer Regeln entschieden werden. Eine Anfrage von einem PEP an den PDP enthält solche Attribute, z. B. über die konkrete Ressource.

Ein Beispiel für das Zusammenspiel zwischen Anfrage, Policy und Antwort findet sich unter `http://sunxacml.sourceforge.net/guide.html#xacml-example`.

2.5.2 Implementation

Die Implementation kann nicht beurteilt werden, da keine sinnvolle Dokumentation zur Verfügung steht.

Es existieren auch andere XACML-Implementationen:

- PicketBox XACML (von JBoss[10]), auch weitestgehend undokumentiert
- Sun XACML (`http://sunxacml.sourceforge.net/`): Diese Implementation scheint aktiv entwickelt zu werden. Leider steht kein Referenzhandbuch zur Verfügung. Außerdem liegt nur eine Datei-basierte Implementation eines *PolicyFinders* vor. Eigene Erweiterungen sind möglich (und auch mit Beispielen dokumentiert), jedoch würde der Aufwand den Nutzen aufgrund der hohen Komplexität von XACML deutlich übersteigen.

2.5.3 Einschätzung

Der XACML-Standard hat bisher noch keine breite Unterstützung erfahren. Dies könnte zum einen daran liegen, dass er eine hohe Komplexität hat und daher nicht nur für Bibliotheksentwickler, sondern auch -anwender schwierig umzusetzen ist. Es ist allerdings möglich, Berechtigungen sehr feingranular umzusetzen, unter anderem auch unter Berücksich der aktuellen Systemzeit. Ob eine derartige Flexibilität jedoch nötig ist, bzw. nicht besser ad hoc unter Umgehung von umfangreicher Konfiguration implementiert werden kann, ist jedoch fraglich.

| domänenspezifische Lösungen

[10] `http://community.jboss.org/wiki/PicketBoxXACMLJBossXACML`

	openCDMS
Produktinformationen	
Kontakt/Webadresse	http://opencdms.org
Entwickler	The University of Manchester
Installationsbasis	keine Angaben
Anschaffung	Freie Software (LGPL3)
Open-Source	Ja
Weiterentwicklung	
- aktiv	Ja
- kostenpflichtig	Nein
- eigene Erweiterungen möglich	kaum
Application Architecture	
Verwendetes Security-Framework	SAML
Softwarearchitektur (Komponenten, -struktur)	Webstart Fat-Clients, WebService-Server
Module	-
Importschnittstellen	
Exportschnittstellen	
Audittrail/Logging Nein	
Berechtigungsmodell	RBAC
Rollen	Vorgegeben in Policy
Multi-User	Ja
Multimandantenfähigkeit	Nein
Internationalisierung	Nein
Data Architecture	
Datenmodell (Domänenmodell)	Rollen, User, Studien, Subjects, Oranizationen, Policies
Schema	Daten relational, User LDAP
Datenlebenszyklus	keiner
Verwaltung von Berechtigungen	GUI + Policy
Speicherung von Berechtigungen	Datenbank
Anbindung an Verzeichnis-/Sicherheitsdienste	LDAP
Technological Architecture	
Programmiersprachen/Frameworks	Java 5, Hibernate, Tomcat
Lines of Code	365.312
Kommunikationsmiddleware	Axis2
Netzwerk (TCP-Ports, Firewall, VPN)	SOAP über HTTPS
Client (fat/thin)	Fat-Client
Clientanforderungen	
- Softwareanforderungen	JRE 5.0
- Updatemöglichkeiten	N/A download aktuellster Version per Web-Start
- Hardwareanforderungen	-
Serveranforderungen	

- Softwareanforderungen	JRE 5.0 + beliebige, von Hibernate unterstützte DB
- Hardwareanforderungen	-
nichtfunktionale Eigenschaften	
Performance	Clients benötigen lange zum Starten, ansonsten OK
Usability	Unintuitiv
Deployment	schwer bis nahezu unmöglich
Benutzerdokumentation	mager
Entwicklerdokumentation	mager