Resource Access Decision
ein Framework zur Realisierung eines datenbasierten Zugriffsschutzes

Werner Eberling

MATHEMA Software GmbH,
Nägelsbachstraße 25b,
91052 Erlangen, Deutschland
werner.eberling@mathema.de
http://www.mathema.de

Abstract. Die Sicherheitsanforderungen moderner, serverseitiger Applikationen enden nicht auf Klassen- bzw. Methodenebene. Hier wird ein Schutz auf Instanzen- oder Dateneben benötigt. Mit dem Resource Access Decision (RAD) Service hat Object Management Group (OMG) einen Dienst spezifiziert, der als Rahmen für die Realisierung eines datenbasierten Zugriffsschutzes dient. Ausgehend von der OMG-Spezifikation wird der Aufbau eines Frameworks zur Realisierung eines datenbasierten Zugriffsschutzes skizziert. Des Weiteren wird die Notwendigkeit der Anpassung dieses Ansatzes an die jeweiligen konkreten Anforderungen, sowie mögliche Stolpersteine auf dem Weg zur Entwicklung eines sowohl mächtigen (weil typ-, instanz- und/oder datenbasierten) aber gleichzeitig beherrschbaren Autorisierungsmechanismus aufgezeigt.

1 Das Problem

Spätestens seit die 1995 verabschiedete "EU-Direktive zum Schutz des Individuums bei automatisierter Verarbeitung seiner persönlichen Daten und deren Austausch" endgültig ihren Weg in die deutsche Gesetzgebung gefunden hat, stellt sich die Frage nach Möglichkeiten zur Umsetzung der verschärften Datenschutzanforderungen. Die informationstechnische Realisierung soll dabei in der Regel größtmögliche Flexibilität bei möglichst hoher Transparenz für die Zielanwendung gewährleisten.

1.1 Was gibt es schon ?

Nun ist das Thema Sicherheit bei serverseitigen Applikationen keineswegs neu. So bieten z.B. die OMG im Rahmen des CORBA-Standards oder Sun innerhalb der J2EE standardisierte Sicherheitsdienste, mit Möglichkeiten zur Authentifizierung und Autorisierung, an. Beide ermöglichen den Aufbau von security-aware bzw. security-unaware Applikationen, also von Anwendungen, die sich der Existenz des Sicherheitsdienstes bewusst sind und diesen aktiv nutzen (security-aware) bzw. von

Anwendungen, für die der Sicherheitsdienst vollkommen transparent bleibt (security-unaware).

1.2 Warum reicht das nicht aus ?

Durch die Etablierung der Java 2 Enterprise Edition als Standard für die Entwicklung serverseitiger Mehrschicht-Architekturen scheinen damit die Sicherheits-anforderungen, die sich u.a. aus dem Datenschutz ergeben, auf den ersten Blick einfach zu lösen. Schließlich existiert in der J2EE die Möglichkeit der deklarativen Autorisierung. Dieser Ansatz beschränkt sich allerdings auf die Autorisierung auf Typ- oder Methodenebene und geht somit für die einfachsten Datenschutz-anforderungen nicht weit genug, was sich an einem einfachen Beispiel verdeutlichen lässt:

Bei der Modellierung eines Bankkontos ist es offensichtlich, dass beispielsweise die Methode `hebeAb(double betrag)` zu schützen ist. So sollen etwa nur Filialmitarbeiter (für eine Auszahlung am Schalter) und der Kontoinhaber (für Abhebungen über den Geldautomaten) das Recht haben, diese Methode aufzurufen.

Unter der Prämisse dies nicht fest innerhalb der Bankkontoimplementierung zu codieren löst der J2EE-Security-Ansatz nur den ersten Fall. Hierfür wird eine Rolle Filialmitarbeiter definiert, der das entsprechende Recht zugesprochen wird. Alle Bankmitarbeiter die in der Filiale tätig sind können dieser Gruppe zugeordnet werden und dürfen somit von Bankkonten abheben (Hierbei muss man immer noch die Ehrlichkeit der Bankmitarbeiter voraussetzen!).

Für den Kontoinhaber reicht eine Rollendefinition allerdings nicht aus, da dies jedem Kontoinhaber das Recht einräumen würde, von allen Konten der Bank Geld abzuheben. Hier ist vielmehr ein instanzbasierter Zugriffsschutz nötig.

1.3 Von Barrieren und Filtern

Unabhängig davon, was geschützt werden soll (Klassen, Methoden, Daten, ...) können grundlegend zwei Arten von Zugriffsschutz unterschieden werden.

Barrieren sind die einfachste Art des Zugriffsschutzes. Versucht der Benutzer, auf einen Dienst zuzugreifen, für den er keine Zugriffsrechte hat, so wird ihm der Zugriff mit einer entsprechenden Fehlermeldung verweigert. Es ist dam Benutzer also bewusst, dass es Bereiche des Systems gibt, auf die er keinen Zugriff hat.

Filter spiegeln dem Benutzer dagegen vor, dass er alle Dienste nutzen kann, indem ihm diejenigen Dienste, auf die er nicht zugreifen darf, gar nicht erst angeboten werden. Sie bilden somit eine elegante Lösung, da sie den Benutzer gar nicht erst dazu verleiten, einen Dienst in Anspruch zu nehmen, den er sowieso nicht benutzen darf.

2 Theoretischer Ansatz

Bereits im Jahre 1998 verfasste die corbamed Taskforce der OMG ein Request for Proposal (RfP corbamed/98-02-01) für einen neuen Dienst zum Schutz von Patientendaten innerhalb medizinischer DV-Systeme. Als Antwort auf diesen RfP gab es eine Reihe von Submissions (u.a. corbamed/99-05-04), in denen die grundlegende Struktur eines Resource Access Decision-Services (RAD) definiert wurde.

2.1 RAD-Spezifikation nach OMG

Obwohl der RAD im ursprünglichen RfP sowohl auf security-aware, als auch für security-unaware Applications abzielte, verlagerte sich der Focus in den Submissions auf den Einsatz im security-aware Bereich.

Die Grundidee des RAD war dabei einen Dienst zu schaffen, den die jeweilige Applikation vor dem Zugriff auf eine schützenswerte Ressource (Klasse, Methode, Instanz, Datum,...) befragen kann, ob dieser Zugriff erlaubt ist. Somit wurde eine zentrale Entscheidungsinstanz geschaffen, die (weitgehend) unabhängig von der jeweiligen Applikation konfiguriert werden kann.
Um eine größtmögliche Unabhängigkeit von dem zugrundeliegenden Authentifizierungsdienstes zu gewährleisten, arbeitet der CORBA-RAD nicht direkt mit dem User-Principal, sondern mit dessen Sicherheits-Attributen.

Der CORBA-RAD besteht aus insgesamt fünf Interfaces, die bei einer Zugriffs-entscheidung wie folgt zusammenarbeiten:

Die Applikation, die den Zugriff auf eine Ressource überprüfen lassen möchte, befragt den RAD-Dienst über das AccessDecision-Interface, wobei sie den Namen der entsprechenden Ressource, die Bezeichnung der darauf auszuführenden Operation und die Sicherheits-Attribute des aktuellen Benutzers übergibt. Je nach Rückgabewert der Methode wird der Zugriff erlaubt oder verwehrt.

```
interface AccessDecision {

   boolean access_allowed (

     in  ResourceName     resource_name,

     in  Operation        operation,

     in  AttributeList    sec_attribute_list

   ) raises (InternalError)

BooleanList multiple_ access_allowed (
```

```
        in  AccessDefinitionList    access_requests,

        in  AttributeList
            sec_attribute_list

      ) raises (InternalError)

    }
```

Zusätzlich bietet die Schnittstelle eine Komfortmethode für die einfache Abfrage mehrerer gleichzeitiger Ressourcenzugriffe.

Das AccessDecisionObject des RAD-Dienstes bereitet die übergebenen Sicherheits-Attribute über den DynamicAttributeService für die eigentliche Zugriffsentscheidung auf. Dieser Dienst ist für eventuelle User-, Rollen- oder Rechte-Mappings zuständig.

```
   interface DynamiAttributeService{

      AttributeList get_dynamic_attributes(

               in  AttributeList    attribute_list,

               in  ResourceName     resource_name,

               in  Operation        operation

      ) raises (ComponentError)

    }
```

Nach der Aufbereitung der Sicherheitsattribute ermittelt das AccessDecisionObjekt über den PolicyEvaluatorLocator die für die jeweilige Ressource zuständigen Entscheidungsinstanzen (PolicyEvaluatoren) sowie das passende Objekt zur Abstimmung innerhalb der einzelnen Entscheidungsinstanzen (DecisionCombinator).

```
   interface PolicyEvaluatorLocator{

       PolicyDecisionEvaluators
  get_policy_decision_evaluators(

                 in  ResourceName   resource_name

       ) raises (ComponentError)

    }
```

PolicyEvaluatoren bilden die eigentliche Entscheidungsinstanz. Sie entscheiden auf Basis von hinterlegten Policies ob der Zugriff auf eine Ressource erlaubt ist. Die Konkrete Ausprägung der PolicyEvaluatoren und der Policies ist dabei abhängig von der jeweiligen RAD-Implementierung.

Für jede Ressource muss mindestens ein zuständiger PolicyEvaluator existieren. Allerdings kann ein PolicyEvaluator für eine ganze Gruppe von Ressourcen bzw. mehrere PolicyEvaluatoren für eine Ressource zuständig sein.

```
interface PolicyEvaluatorLocator{

    DecisionResult evaluate(

              in  ResourceName    resource_name,

              in  Operation operation,

              in  AttributeList   attribute_list
    )

 }
```

Für den Fall, dass mehrere PolicyEvaluatoren in den Entscheidungsprozess involviert sind, erfolgt die „Abstimmung“ des Ergebnisses über einen DecisionCombinator.

```
interface DecisionCombinator{

   boolean combine_decisions(

             in  ResourceName      resource_name,

             in  Operation   operation,

             in  AttributeList     attribute_list,

             in  PolicyEvaluatorList policy_eval_list

   ) raises (ComponentError)

 }
```

Neben den hier vorgestellten operativen Schnittstellen existiert auch eine Reihe administrativer Schnittstellen zur Konfiguration des CORBA-RAD. Diese sollen hier aber nicht näher betrachtet werden.

2.2 Resource Naming

Die Grundlage für die Zugriffskontrolle über einen CORBA-RAD ist die eindeutige Identifikation von Ressourcen. Hierfür ist das sogenannte ResourceNaming zuständig. Die konkrete Realisierung eines solchen „Ressourcen-Namensraums“ ist dabei abhängig von der jeweiligen RAD-Implementierung. Dabei ist allerdings zu beachten, dass eine Ressourcen-Gruppierung (z.B. über „Wildcards“) möglich ist. Darüber hinaus gibt die Granularität der Ressourcen-Namen die feinste Granularität der Rechtevergabe vor, da der Ressourcen-Name die kleinste Einheit für die Rechtevergabe ist.

Die OMG schlägt für ein systemübergreifendes ResourceNaming einen hierarchischen Aufbau in Form von Name/Wert-Paaren vor, wobei jede Gruppe von ResourceNames durch eine NamingAuthority über Systemgrenzen hinweg eindeutig ist.

3 Java Implementierung

Aufbauend auf der OMG-Spezifikation des CORBA-RAD wurde im Rahmen eines Projektes im Bankenumfeld eine „handhabbargemachte“ Java Implementierung realisiert.

3.1 Grundlegender Aufbau

Die Implementierung kommt mit Standard-Java-Mechanismen aus, benötigt also keine J2EE-Umgebung. Einzige Technologievoraussetzungen sind das Java-Permission-Konzept und der Java Authentication and Authorization Service (JAAS) (beide Teile des j2sdk (Java 2 Software Development Kits) ab Version 1.4).

Der Ressourcenschutz erfolgt über Positivlisten (AccessControlLists) sowie über Bedingungen. Er realisiert sowohl Barrieren als auch Filter und bietet ein Rollenkonzept für eine vereinfachte Administration.

Der grundlegende Ablauf der Entscheidungsfindung entspricht der Vorgehensweise des CORBA-RAD. Während dieser die Authentifizierung des Benutzers außen vor lässt, ist sie im Rahmen dieser Implementierung natürlich mit zu berücksichtigen. Hierzu wird der standardisierte JAAS Mechanismus eingesetzt, der den Java-RAD sowohl im Standard-Java, als auch im J2EE Umfeld einsetzbar macht.

Auf Applikationsseite ist hier vor allem der User-Principal des gerade angemeldeten Benutzers von Interesse. Dieser wird, zusammen mit dem Ressource-Namen und einer Liste der Aufrufparameter an das AccessDecisionObjekt übergeben.

```
public interface AccessDecision {

   public final static short
                   ACCESS_DECISION_ALLOWED = 2;

   public final static short
                   ACCESS_DECISION_NOT_ALLOWED = 4;

   public final static short
                   ACCESS_DECISION_UNKNOWN = 8;

   public short accessAllowed(
                           Principal caller,
                           String resourceId,
                           Hashtable params)
                  throws RechteverwaltungException;

   public short[] multipleAccessAllowed(
                           Principal caller,
                           String[] resourceIds,
                           Hashtable[] params)
                  throws RechteverwaltungException;

   public String[] getAvailableResources(
                           Principal caller,
                           String resourceId,
                           Hashtable params)
                  throws RechteverwaltungException;

   public Object[] filterObjects(
                           Principal caller,
                           String resourceId,
                           Object[] obj2filter)
                   throws RechteverwaltungException;

 }
```

Zusätzlich zu den vom CORBA-RAD bekannten Methoden wurde die Schnittstelle um die Methoden `getAvailableResources()`, zum erfragen eine Liste zugreifbarer Ressourcen, und `filterObjects()`, zum ausfiltern verbotener Objekte, erweitert. Über diese beiden Methoden wird ein erweitertes Filtering möglich.
Die Ermittlung der Credentials des übergebenen Principals ist Aufgabe des DynamicAttributeServices. Dabei werden zusätzlich zu den Rechten, die direkt dem Principal zugeordnet sind, auch die ihm zugeordneten Rollen berücksichtigt. Nachdem die Credentials aufgebaut wurden, erfolgt der weitere Entscheidungsprozess auf Basis dieses Credential-Objektes.

```
public interface DynamicAttributeService {

   public Credentials getCredentials(
                            Principal caller);

}
```

Der PolicyEvaluatorLocator liefert direkt einen DecisionCombinator zurück, der entweder die Entscheidung über einen Ressourcen-Zugriff fällen kann, oder eine Liste von Ressource-Namen zurückgibt, auf die zugegriffen werden darf.

```
public interface PolicyEvaluatorLocator {

   public DecisionCombinator getDecisionCombinator(
                            String resourceId);

}

public interface DecisionCombinator {

   public void addPolicyEvaluator(
                            PolicyEvaluator pe);

   public short combineDecisions(
                            Credentials creds,
                            String resourceId,
                            Hashtable params);

   public String[] getAvailableResources(
                            Credentials creds,
                            String resourceId,
                            Hashtable params);

}
```

Auch die Schnittstelle der PolicyEvaluatoren ist um eine Methode zum Auflisten der zugreifbaren Ressourcen erweitert worden.

```
public interface PolicyEvaluator {

   public final static short ACCESS_ALLOWED = 0;
```

```
    public final static short ACCESS_NOT_ALLOWED = 1;

    public final static short ACCESS_UNKNOWN = 2;

    public short checkPolicy(
                            Credentials creds,
                            String resourceId,
                            Hashtable params);

    public String[] getAvailableResources(
                            Credentials creds,
                            String resourceId,
                            Hashtable params);

 }
```

Zusätzlich zum klassischen AccessControlList-Ansatz, der über einen zentralen PolicyEvaluator unter zu Hilfenahme des Java-Permission-Konzeptes realisiert wird (mehr dazu weiter unten) ermöglicht der Java-RAD die Vergabe bedingter Rechte. Hierzu wurde das Konzept der PolicyEvaluatoren erweitert und die Schnittstelle BedingungPrüfer eingeführt.

```
public interface BedingungPruefer {

    public short pruefeBedingung(
                            String beds,
                            Properties bedParams,
                            Credentials creds,
                            String resourceId,
                            Hashtable callParams);

    public String[] ermittleRessourcen(
                            String beds,
                            Properties bedParams,
                            Credentials creds,
                            String resourceId,
                            Hashtable callParams);

}
```

Einzelne Bedingungen werden in Form spezieller Methoden innerhalb des Prüfers implementiert, die – basierend auf Namenskonventionen – per Reflection gerufen werden. Zusätzlich können Bedingungen parametrisiert werden.

3.2 Resource Naming mit RADPermissions

Zum Aufbau eines Ressourcen-Namensraumes wird das Java-Permission-Konzept verwendet. Hierdurch wird ein hierarchischer Namensraum, mit der Möglichkeit zur Gruppierung von Ressourcen, aufgebaut. Zusätzlich wird dadurch die Rechteverwaltung vereinfacht, da eine Rechteprüfung durch ein einfaches `implies()` realisiert werden kann. Auch die Rechtespeicherung kann durch diesen Ansatz optimiert werden, da nur noch das oberste Recht innerhalb der implies-Kette gespeichert werden muss (=> Transitivität).

```
public class RADPermission extends
                  java.security.BasicPermission{

   public RADPermission(String name) {

       super(name.equals("")?"*":name+".*");

   }

   public boolean impliesBedingt(RADPermission perm){

     [...]

   }

   public String getRechteBezeichnung(){

     [...]

   }

}
```

Die Rechtevergabe erfolgt über ein XML-Konfigurationsfile. Hier können sowohl statische, als auch bedingte Rechte vergeben werden.

```
...

 <LIST name="RECHT" size="2">

  <RECHT RESOURCE_ID="Bankkonto.einzahlen" />

  <RECHT RESOURCE_ID="Bankkonto.abbuchen"/>

 </LIST>

 <LIST name="BEDINGTESRECHT" size="4">
```

```
<BEDINGTESRECHT RESOURCE_ID="Bankkonto.abheben">

 <LIST name="BEDINGUNG" size="1">

  <BEDINGUNG
       VALUE="KontoPruefer;isKontoInhaber" />

 </LIST>

</BEDINGTESRECHT>

<BEDINGTESRECHT RESOURCE_ID="Bankkonto.abheben">

 <LIST name="BEDINGUNG" size="1">

  <BEDINGUNG
       VALUE="KontoPruefer;
        isKontoPruefer:isMinusKleinerAls;grenze=2500"
  />

 </LIST>

</BEDINGTESRECHT>

...
```

4 Zusammenfassung

Die Object Management Group (OMG) hat mit dem Resource Access Decision (RAD) Service einen Dienst spezifiziert, der als Grundlage für die Realisierung eines datenbasierten Zugriffsschutzes dient. Dieser Dienst ermöglicht die klare Trennung von Applikationslogik und Datenschutzlogik bei gleichzeitiger Gewährleistung eines Maximums an Anpassungsfähigkeit. Wie viele Spezifikationen der OMG leidet auch der (wenig bekannte) RAD-Service unter der grundlegenden Komplexität des OMG-Ansatzes, der versucht, alle Eventualitäten abzudecken, weshalb es auch bis heute keine ernstzunehmende kommerzielle Implementierung dieses Dienstes gibt.

Allerdings lässt sich die Komplexität durch die Fokussierung auf die konkreten Projektanforderungen deutlich reduzieren, so das es sich sehr wohl lohnt, aufbauend auf dieser Spezifikation einen projektspezifischen, "abgespeckten" RAD-Dienst zu entwickeln, der sowohl für security-aware, als auch für security-unaware Applications einsetzbar ist. Ein solcher Dienst ermöglicht neben der klassischen Ja/Nein-Entscheidung bzgl. des Zugriffs auf eine schützenswerte Ressource (Barriere) auch Möglichkeiten zum ausblenden nicht-zugreifbarer Daten (Filterung).

Der kritische Punkt bei einem solchen System ist aber die Administration. Diese kann sich leicht ins Uferlose ausweiten, wenn bei der Rechtevergabe nicht frühzeitig auf eine gewisse Struktur mit Ressourcen-Gruppen, Bedingungen und Rollendefinitionen geachtet wird.