Enterprise Security – Instanzbasierte Zugriffskontrolle

Architekturkonzepte und Implementierung mit SAF, JAAS und Spring

Martin Krasser

Der Zugriff auf die Domänenobjekte einer Anwendung wird häufig auf der Ebene von Klasseninstanzen kontrolliert. Zu diesem Thema werden einleitend allgemeine Architekturkonzepte vorgestellt und dann ein Leitfaden zur Implementierung gegeben. Zur Trennung der Sicherheitslogik von der Geschäftslogik in Spring-Anwendungen verwenden wir das Security Annotation Framework (SAF) [1]. Zur Implementierung der Sicherheitslogik erweitern wir den Java Authentication and Authorization Service (JAAS) um Mechanismen zur instanzbasierten Zugriffskontrolle.

Instanzbasierte Zugriffskontrolle regelt den Zugriff auf Instanzen von Domänenobjekten. Instanzbasiert heißt, es wird bei Zugriffsentscheidungen der Zustand der Domänenobjekte berücksichtigt, zusätzlich zu ihren statischen Merkmalen. Das kann eine eindeutig vergebene ID sein aber auch andere sicherheitsrelevante Eigenschaften, wie z.B. die User ID des Besitzers. Instanzbasierte Zugriffskontrolle ist besonders in Systemen von Interesse, in denen Benutzer selbst Zugriffsrechte auf ihre Daten verwalten können. Als Beispiel sei ein Online Fotoalbum erwähnt, bei dem jeder Benutzer für jedes seiner Fotos entscheiden kann, wer darauf Zugriff hat. Die Funktion „Zeige mir die Fotos von Benutzer X“ kann zwar von jedem Benutzer aufgerufen werden, der Aufrufer bekommt aber nur die Fotos von Benutzer X zu sehen, die von Benutzer X auch freigegeben wurden. Die Freigabe kann dabei an einzelne Benutzer erfolgen oder an Benutzergruppen.

Mit den Sicherheitsmechanismen der Java Enterprise Edition kann dieser Anwendungsfall nicht mehr umgesetzt werden. Java EE Security kann zwar gut dazu verwendet werden, das Aufrufen von Funktionen auf bestimmte Rollen einzuschränken, die Zugriffsprüfung auf einzelne Domänenobjekt-Instanzen wird von der Spezifikation aber nicht adressiert [2]. In diesen Fällen muss man auf alternative Lösungen ausweichen. Zum tieferen Verständnis dieser Alternativen besprechen wir zuerst allgemeine Konzepte von Zugriffskontroll-Architekturen und dann Möglichkeiten zur Implementierung. Die Implementierung erfolgt anhand eines Beispiels unter Verwendung des Security Annotation Frameworks (SAF) und Java SE Security Standards [3, 4]. Die hier vorgestellte Lösung wird am Ende des Artikels kurz mit anderen Security Frameworks verglichen.

Im Hauptteil des Artikels beschäftigen uns mit dem SAF. Dieses überprüft den Zugriff auf die Domänenobjekte einer Anwendung an annotierten Stellen im Code und erlaubt somit die Trennung der Sicherheitslogik einer Anwendung von ihrer Geschäftslogik. Sicherheitslogik wird vom SAF über eine Service Provider Schnittstelle in Spring-Anwendungen eingebunden. Durch seinen deklarativen Ansatz zur instanzbasierten Zugriffskontrolle erlaubt das SAF auch die nachträgliche Einbindung von Sicherheitsdiensten, ohne die Geschäftslogik von bestehenden Spring-Anwendungen ändern zu müssen.

Danach beschäftigen wir uns mit der Implementierung eines Sicherheitsdienstes auf Basis von Java SE Security Standards. Wir verwenden dazu den Java Authentication and Authorization Service (JAAS) und erweitern diesen um Mechanismen zur instanzbasierten Zugriffskontrolle. Diese Erweiterungen betreffen dabei nur den Autorisierungsteil von JAAS. Der Authentifizierungsteil ist nicht Thema des Artikels. Durch die Einhaltung von Standards könnten wir die Erweiterungen auch

in anderen Umgebungen wieder verwenden, welche zur Autorisierung auf JAAS bauen, ohne dabei Architekturänderungen vornehmen zu müssen. Anwendungen sprechen diese Erweiterungen dabei immer über die Standard Security APIs des JDK an.

Die JAAS Erweiterungen [5] und die Beispielanwendung [6] sind mittlerweile auch Teil des Open Source SAF Projektes. Dieser Artikel bezieht sich aber mit den Begriffen „Security Annotation Framework“ bzw. „SAF“ alleine auf das SAF Kernmodul [7] des Projekts.

Architektur

Als Bezugspunkt für die weiteren Abschnitte dient die Referenzarchitektur aus Abb. 1. Sie definiert die Komponenten einer allgemeinen Zugriffskontroll-Architektur, wie sie in vielen Anwendungen zu finden ist. Ziel ist es, Zugriffe von Clients auf die geschützten Ressourcen einer Anwendung zu überwachen und Zugriffsentscheidungen zu treffen. In unserem Beispiel sind das die Domänenobjekte einer Anwendung.

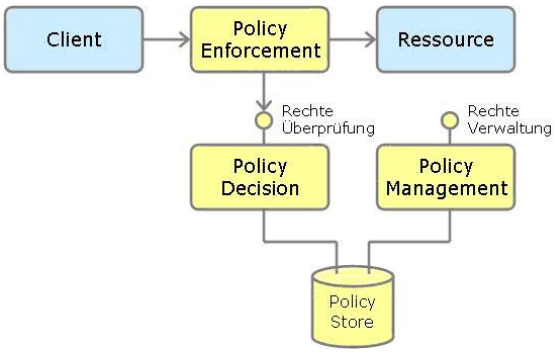

Abb. 1: Zugriffskontroll-Referenzarchitektur

Die Überwachung erfolgt durch eine Policy Enforcement Komponente. Diese stellt sicher, dass beim Zugriff auf eine Ressource die Sicherheitsrichtlinien (Security Policy) der Anwendung eingehalten werden, indem sie eine Berechtigungsprüfung einleitet. Abhängig von der Ressource und der Art des Zugriffs stellt die Policy Enforcement Komponente eine entsprechende Anfrage an eine Überprüfungsschnittstelle, die von einer Policy Decision Komponente bereitgestellt wird. Die Policy Decision Komponente trifft nun eine Entscheidung, ob der Zugriff gewährt werden soll, oder nicht. Dazu überprüft sie, ob für den angemeldeten Benutzer ausreichende Rechte im Policy Store eingetragen sind. Hat der Benutzer das Recht auf die Ressource zuzugreifen, generiert die Policy Decision Komponente eine positive Antwort und die Policy Enforcement Komponente lässt die Anfrage durch. Fällt die Zugriffsentscheidung negativ aus, wird die Anfrage blockiert. Im Policy Store werden die Zugriffsberechtigungen für die Benutzer einer Anwendung gespeichert. Die Verwaltung der Zugriffsberechtigungen geschieht über eine Policy Management Komponente. Die Änderung von Berechtigungen muss zur Laufzeit möglich sein, damit z.B. ein Benutzer einem anderen Benutzer Zugriff auf seine Daten gewähren kann. Die beschriebene Zugriffskontroll-Architektur und die Interaktionsmuster ihrer Komponenten findet man sowohl in verteilten Anwendungen als auch in Anwendungen, deren Komponenten innerhalb einer Laufzeitumgebung agieren. Für diesen Artikel betrachten wir nur letzteren Fall.

Vergleichen wir nun die Referenzarchitektur mit der Lösungsarchitektur unserer Beispielanwendung (Abb. 2). Für die Implementierung der Policy Enforcement Komponente verwenden wir das Security Annotation Framework (SAF). Für die Implementierung der Policy

Decision Komponente verwenden und erweitern wir das JAAS Authorization Framework. Policy Management Funktionalität wird ebenfalls von dieser Komponente zur Verfügung gestellt.

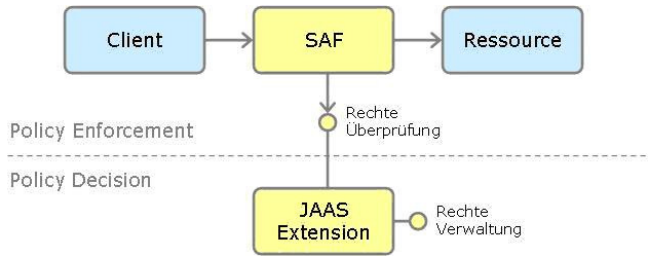

Abb. 2: Zugriffskontroll-Architektur der Beispielanwendung.

Die Policy Decision Komponente ist eine einfache Referenz-Implementierung [5] und soll lediglich als Leitfaden zur Implementierung von Komponenten für den produktiven Einsatz dienen. Man kann an dieser Stelle auch beliebige andere Implementierungen verwenden, die instanzbasierte Zugriffskontrolle unterstützen. Das SAF Kernmodul [7] hingegen ist reif für den produktiven Einsatz in Spring-basierten Enterprise Anwendungen. Mit dem SAF können wir den Zugriff auf die Domänenobjekte einer Spring-Anwendung überwachen und Zugriffsentscheidungen an Policy Decision Komponenten delegieren. Das SAF kann dabei konfiguriert werden, mit welcher Policy Decision Komponente es interagieren soll.

Policy Enforcement

Um die Besonderheiten der instanzbasierten Zugriffskontrolle zu verstehen, wollen wir die Komponenten aus Abb. 2 genauer betrachten. Wir beginnen mit der Policy Enforcement Komponente und zeigen, wie sie mit dem SAF in Spring-Anwendungen implementiert werden kann. Zunächst ein Überblick über unsere Beispielanwendung (Abb. 3): Es handelt sich um eine Spring-Anwendung zur Verwaltung von Notizbüchern. Benutzer können eigene Notizbücher verwalten (Erzeugen, Suchen und Löschen) und in Notizbüchern Einträge vornehmen (Ändern). Die Notizbuchverwaltung erfolgt über eine `NotebookService` Schnittstelle. Das Erzeugen und Löschen von Einträgen in Notizbüchern geschieht über die Methoden `addEntry()` und `removeEntry()` am `Notebook` Domänenobjekt selbst (Listing 1). Der `Notebo`*`okService`* wird als Bean im Spring Application Context verwaltet.

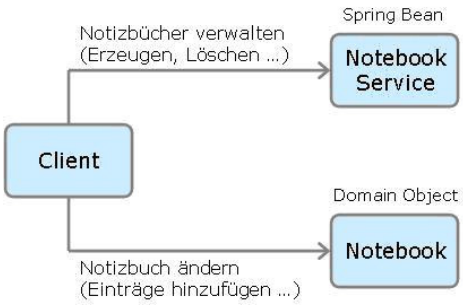

Abb. 3: Überblick über die Beispielanwendung.

LISTING 1

```
public interface NotebookService {

    void createNotebook(@Secure(SecureAction.CREATE) Notebook notebook);
    void deleteNotebook(@Secure(SecureAction.DELETE) Notebook notebook);

    @Filter Notebook findNotebook(String id);
    @Filter List<Notebook> findNotebooksByUserId(String userId);
    …
}

@SecureObject
public class Notebook {

    private String id;
    private User owner;
    private List<Entry> entries;

    @Secure(SecureAction.UPDATE)
    public void addEntry(Entry entry) {
        …
    }
    @Secure(SecureAction.UPDATE)
    public void removeEntry(Entry entry) {
        …
    }
    …
}

public class User {…}
public class Entry {…}
```

Ohne Sicherheitsmechanismen hat jeder Benutzer vollen Zugriff auf die Notizbücher aller anderen Benutzer. Um den Zugriff auf Notizbücher einzuschränken, werden zunächst Berechtigungsprüfungen an bestimmten Stellen im Code erzwungen. Als Beispiel betrachten wir die Methode

```
void deleteNotebook(Notebook notebook)
```

der `NotebookService` Schnittstelle. Diese Methode löscht Notizbücher aus einer Datenbank und sollte von Benutzern der Anwendung nur dann aufgerufen werden dürfen, wenn das als Argument übergebene Notizbuch auch dem aktuell angemeldeten Benutzer gehört. Anders ausgedrückt, ein Benutzer darf seine eigenen Notizbücher löschen, aber nicht die von anderen Benutzern. Hier muss der Inhalt des `Notebook`-Objekts überprüft werden, ob es mit dem angemeldeten Benutzer in Beziehung gebracht werden kann. Es gibt hier also Fälle in denen die Methode von ein und demselben Benutzer mal aufgerufen werden darf und mal nicht, je nach Argument. Man kann daher nicht statisch festlegen, wer diese Methode aufrufen darf, sondern nur, dass eine Überprüfung an dieser Stelle erfolgen soll. Um das zu erreichen, versehen wir den Methodenparameter `notebook` mit einer SAF `@Secure` Annotation. Die Art des Zugriffs wird mit einer Konstante des Enumtyps `SecureAction` festgelegt.

```
void deleteNotebook(@Secure(SecureAction.DELETE)Notebook notebook)
```

In diesem Fall wird für das `notebook` Argument geprüft, ob es vom aktuellen Benutzer gelöscht werden darf. Das SAF delegiert dabei Zugriffsentscheidungen an eine Bean, die das SAF `AcessManager` Interface implementiert (Listing 2). Beim Aufruf der `deleteNotebook()` Methode wird also zuerst die `checkDelete()`-Methode am `AccessManager` aufgerufen, und als Argument die `Notebook` Instanz übergeben, die auch der `delete`*`eNotebook()`* Methode übergeben wurde (Abb. 4). Hat der aktuelle Benutzer die Berechtigung die `Notebook` Instanz zu löschen, erlaubt das SAF den Aufruf der `deleteNotebook()` Methode am `NotebookService`, andernfalls wird eine von der `AccessManager`-Implementierung

geworfene Security Exception an den Client weitergereicht. Die von den `check()`-Methoden des `AccessManager` geworfenen Exceptions müssen vom Typ `java.security.AccessControlException` sein.

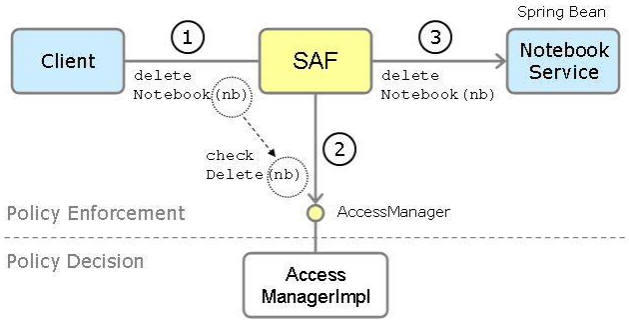

Abb. 4: Das SAF überprüft den Zugriff auf eine Instanz, die als Argument übergeben wurde.

LISTING 2

```
public interface AccessManager {

    void checkCreate(Object obj);
    void checkRead(Object obj);
    void checkUpdate(Object obj);
    void checkDelete(Object obj);
    …
}
```

Das `AccessManager` Interface wird zwar vom SAF definiert, es wird aber keine Implementierung mitgeliefert. Diese muss von einer Policy Decision Komponente zur Verfügung gestellt werden. Das SAF macht an dieser Stelle keine Vorschriften, wie eine Implementierung auszusehen hat bzw. wie eine Zugriffsentscheidung getroffen werden soll. Das bleibt völlig der `AccessManager` Implementierung überlassen. Wir werden später sehen, wie wir JAAS um Mechanismen zur instanzbasierten Zugriffskontrolle erweitern und diese Erweiterungen über die `AccessManager` Schnittstelle in eine Spring-Anwendung integrieren. Man könnte an dieser Stelle aber auch beliebige andere Sicherheitsdienste einbinden.

Listing 2 zeigt nur einen Ausschnitt der Methoden des `AccessManager` Interface. Weitere Methoden ermöglichen auch die direkte Modifikation von Methodenargumenten und Rückgabewerten und somit die Implementierung von beliebigen Zugriffsregeln. Für eine Beschreibung der kompletten Schnittstelle sei auf die SAF Dokumentation verwiesen [7].

Das SAF kann auch verwendet werden, um `@Secure` Annotationen auf Objekte anzuwenden, die nicht vom Spring Application Context verwaltet werden. Das sind typischerweise die Domänenobjekte einer Anwendung, in unserem Fall das `Notebook`. Bei Domänenobjekten wendet man die `@Sec`*`ure`* Annotation meist direkt auf eine Methode an und nicht auf einen Methodenparameter. Das führt dazu, dass das Objekt, auf dem die Methode aufgerufen wird, selbst überprüft wird und nicht irgendein Argument (Abb. 5). Nehmen wir als Beispiel die `addEntry()` Methode des `Notebook` Domänenobjekts. Diese Methode erzeugt einen zusätzlichen Eintrag in der `Notebook` Instanz, auf der sie aufgerufen wird und ändert sie somit. Uns ist dabei egal, wie der Eintrag (d.h. das Argument) aussieht. Uns interessiert nur, ob der aktuelle Benutzer die entsprechende Notebook Instanz ändern darf. Das erreichen wir mittels einer `@Secure(SecureAction.UPDATE)` Annotation auf Methodenebene (Listing 1). In diesem Fall ruft das SAF die `checkUpdate()` Methode auf und übergibt als Argument das Zielobjekt d.h. die `Notebook`-Instanz selbst. Man könnte diesen Ansatz der Überprüfung auch den

„objektorientierten“ Ansatz nennen, während man die Überprüfung auf Parameterebene eher als „prozeduralen“ oder „serviceorientierten“ Ansatz bezeichnen würde.

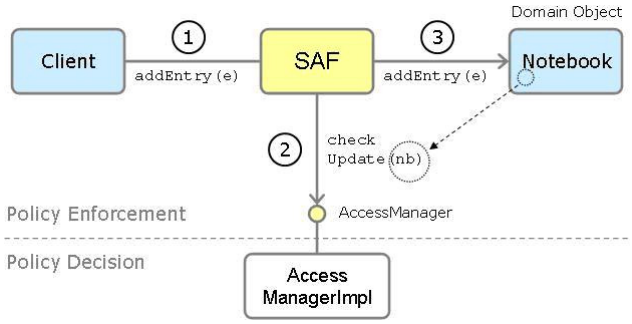

Abb. 5: Das SAF überprüft den Zugriff auf eine Instanz, auf der eine Methode aufgerufen wurde.

Weiterhin unterstützt das SAF auch die Überprüfung und Filterung der Rückgabewerte eines Methodenaufrufs. Das geschieht mit der SAF `@Filter` Annotation. Bei der `@Filter` Annotation wird implizit ein Lesezugriff angenommen. Wird eine `Collection` oder ein Array zurückgeliefert werden deren Elemente überprüft, ansonsten wird das zurück gelieferte Objekt selbst überprüft. Im Falle einer `Collection` kann noch angegeben werden, ob Elemente, auf die keine Leseberechtigungen existieren, aus der Collection gelöscht werden sollen, oder ob die lesbaren Elemente in eine neue Collection kopiert werden sollen. Die Klasse einer evtl. neu zu erzeugenden `Collection` kann ebenfalls mit angegeben werden. Die Filterung von großen `Collection`s kann aus Performance-Sicht problematisch sein. Es gibt hier Möglichkeiten zur Optimierung, auf die aber aus Platzgründen nicht genauer eingegangen werden kann.

Damit das SAF die Annotationen prozessieren und Berechtigungsprüfungen durchführen kann, wird es im Spring Application Context durch das Element `<sec:annotation-driven>` aktiviert (Listing 3). Die Schemadefinition für den `sec` Namespace wird vom SAF mitgeliefert und kann in Spring 2.x Umgebungen verwendet werden. Die `AccessManager` Bean, an die das SAF delegiert, wird über das `access-manager` Attribut des `<sec:annotation-driven>` Elements gesetzt. Das SAF erlaubt uns also mit minimalem Konfigurationsaufwand, Zugriffe auf die Domänenobjekt-Instanzen einer Anwendung zu prüfen. Leser, die sich mit deklarativem Transaktionsmanagement und `@Transactional` Annotationen in Spring 2.x Anwendungen beschäftigt haben, erkennen hier sicher Parallelen.

LISTING 3

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:sec="http://safr.sourceforge.net/schema/core"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://safr.sourceforge.net/schema/core
http://safr.sourceforge.net/schema/core/spring-safr-core-1.0.xsd">

    <sec:annotation-driven access-manager="accessManager"/>

    <bean id="accessManager" class="...AccessManagerImpl" />
    <bean id="notebookService" class="...NotebookServiceImpl" />
    …
</beans>
```

Abb. 6 illustriert den dahinter liegenden Mechanismus. Beim Laden des Spring Application Context verarbeitet das SAF das `<sec:annotation-driven>` Element und erzeugt für Spring Beans mit SAF Annotationen automatisch AOP Proxies. Diese erzwingen über einen `MethodInterceptor` vor und nach Methodenaufrufen entsprechende Berechtigungsprüfungen. Der `MethodInterceptor` ruft dabei die unterschiedlichen `check()`-Methoden am `AccessManager` auf. Wirft eine `check()`-Methode eine `AccessControlException`, wird sie im Falle einer `@Secure` Annotation an den Client weitergereicht. Im Falle einer `@Filter` Annotation wird sie vom Interceptor abgefangen und das Objekt aus dem Resultat entfernt.

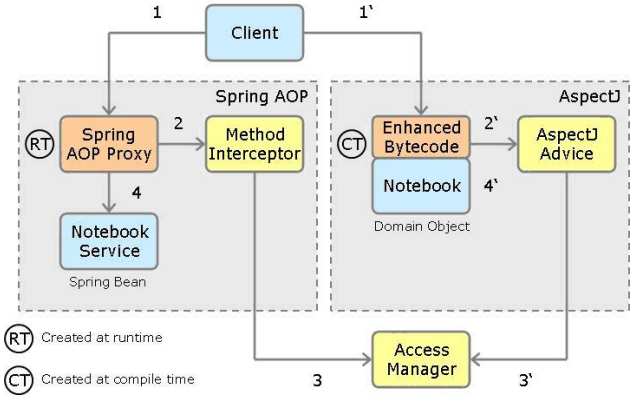

Abb. 6: SAF verwendet Spring AOP und AspectJ.

Für Spring Beans erfolgt das Erzeugen von AOP Proxies zur Laufzeit des Systems. Bei Domänenobjekten wird das anders gemacht. Hier können keine Spring AOP Mechanismen [8] verwendet werden, da die Domänenobjekte dem Spring Application Context nicht bekannt sind. Das SAF verwendet in diesem Fall AspectJ [9]. Zur Kompilierungszeit wird der Bytecode von annotierten Domänenobjekten so modifiziert, dass beim Aufruf einer Methode ein AspectJ Advice getriggert wird, welcher am `AccessManager` eine entsprechende `check()`-Methode aufruft. Um sicherzustellen, dass nur Domänenobjekte und keine Spring Beans vom AspectJ Compiler „enhanced“ werden, müssen Domänenobjekte zusätzlich mit einer `@SecureObject` Annotation auf Klassenebene annotiert werden. Ansonsten ist die Semantik von `@Secure` und `@Filter` Annotationen auf Spring Beans und Domänenobjekten gleich. Spätere Versionen des SAF werden auch „load-time weaving“ unterstützen. Dabei erfolgt die Modifikation des Bytecodes zum Zeitpunkt des Ladens einer Domänenobjekt Klasse.

Ohne Verwendung des SAF müsste man die Konfiguration der Interceptoren bzw. Advices aus Abb. 6 selbst übernehmen, was bei großen Anwendungen schnell unübersichtlich und fehleranfällig wird. Die Konfiguration enthielte dann auch (Teile von) Klassen- und Methodennamen, welche beim Refactoring zusätzlich angepasst werden müssten. Das kann leicht übersehen werden und zu Sicherheitslücken in der Anwendung führen. Durch die Verwendung von SAF Annotationen werden umbenannte Klassen und Methoden zur Übersetzungs- bzw. Laufzeit automatisch neu erkannt und geschützt.

Policy Decision

Bisher haben wir nur Stellen im Quellode annotiert, an denen Zugriffe auf Domänenobjekt-Instanzen geprüft werden sollen. Zugriffsentscheidungen wurden noch nicht getroffen. Diese werden vom SAF über den `AccessManager` an eine Policy Decision Komponente delegiert. Diese Komponente entscheidet letztendlich ob der Zugriff auf eine Domänenobjekt Instanz erlaubt ist, oder

nicht. Zum Treffen von Zugriffsentscheidungen wollen wir auf die Zugriffskontrollmechanismen der Java SE Plattform und JAAS zurückgreifen und diese entsprechend unserer Bedürfnisse erweitern. Diese Erweiterung kann aus Platzgründen hier nur sehr oberflächlich beschrieben werden. Für Implementierungsdetails sei auf die Beispielanwendung verwiesen [6, 5].

Abb. 7 gibt einen groben Überblick über den Aufbau der Policy Decision Komponente unserer Beispielanwendung. Die Klassen der Java Security Architektur, die wir verwenden und erweitern wollen, liegen im `java.security` Paket. Das SAF `AccessManager` Interface ist Teil des `net.sourceforge.safr.core.provider` Pakets.

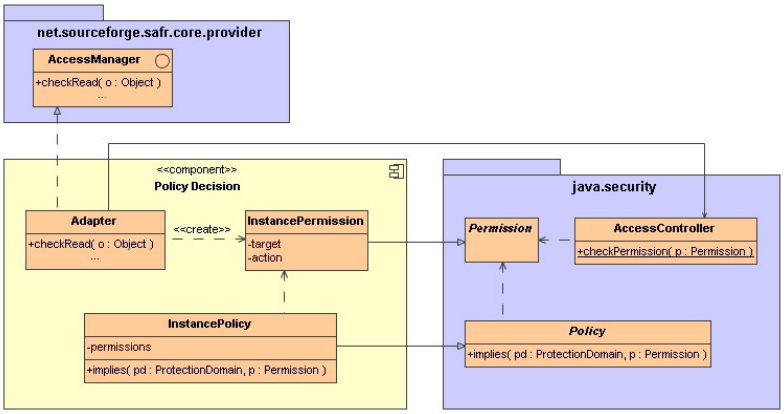

Abb. 7: Aufbau der Policy Decision Komponente

Um Zugriffe auf die Ressourcen einer Anwendung zu beschreiben, definiert die Java Security Architektur die abstrakte Klasse `Permission`. Da wir den Zugriff auf Domänenobjekt-Instanzen repräsentieren wollen, leiten wir davon unsere eigene `InstancePermission` Klasse ab. Das `target` Attribut dieser Klasse ist eine Art Referenz und beschreibt auf welche Domänenobjekt-Instanzen zugegriffen werden soll. Das `action` Attribut beschreibt die Art des Zugriffs (z.B. Lesen). Um eine Zugriffsentscheidung zu treffen erzeugen wir ein `InstancePermission` Objekt und rufen damit am `AccessController` die `checkPermission()` Methode auf. Dieser Aufruf wird von der Adapterklasse aus Abb. 7 getätigt. Diese bindet somit den `AccessController` an das SAF oder, anders ausgedrückt, das SAF delegiert Zugriffsentscheidungen über den Adapter an den `AccessController`. Die Logik für die Zugriffsentscheidung ist in einer Klasse gekapselt, die von der abstrakten Klasse `Policy` des JDK erbt. Die abgeleitete Klasse wird auch Policy Provider genannt. Die Standard-Implementierung, die mit dem JDK mitgeliefert wird, können wir für unsere Zwecke nicht verwenden, da wir spezielle Logik bei der Verarbeitung von `InstancePermissions` brauchen und auch Berechtigungen zur Laufzeit ändern wollen. Wir stellen daher eine eigene Implementierung (`InstancePolicy`) zur Verfügung. Diese trifft auf Basis von `InstancePermission` Objekten Zugriffsentscheidungen.

Abb. 8 zeigt den Ablauf einer Zugriffsentscheidung. Das SAF ruft z.B. die `checkRead()` Methode am Adapter auf und übergibt die zu prüfende `Notebook` Instanz als Argument. Der Adapter erzeugt ein `InstancePermission` Objekt, das einen Lesezugriff auf die `Notebook` Instanz beschreibt, und übergibt es dem `AccessController` beim Aufruf der `checkPermission()` Methode. Dann ruft der `AccessController`, stark vereinfacht gesagt, die `implies()` Methode der `InstancePolicy` auf und übergibt als Argument die vom Adapter erzeugte `InstancePermission` und ein `ProtectionDomain` Objekt. Das `InstancePermission` Argument liefert die Information, auf welche Domänenobjekt-Instanz

zugegriffen werden soll; das `ProtectionDomain` Argument liefert u.a. Auskunft darüber welcher Benutzer darauf zugreifen möchte. Die `InstancePolicy` prüft nun, ob für den aktuellen Benutzer ausreichende Rechte im Policy Store vorhanden sind (hier nicht gezeigt). Falls ja, liefert die `implies()` Methode `true` zurück, ansonsten `false`. Der Rückgabewert `false` veranlasst den `AccessController` eine `AccessControlException` zu werfen, welche über den Adapter an das SAF und weitergereicht wird.

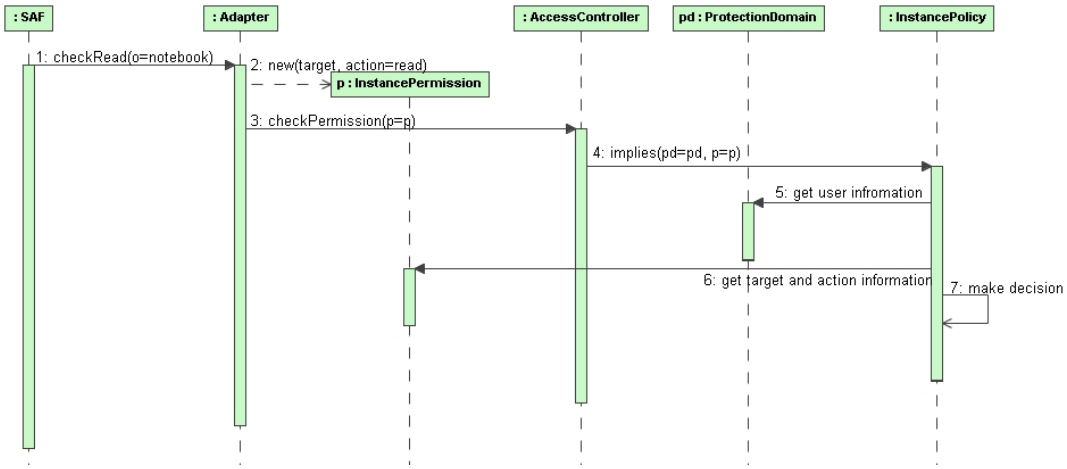

Abb. 8: Treffen einer Zugriffsentscheidung.

Die `InstancePermission` Klasse wurde so entworfen, dass sie den Zugriff auf beliebige Domänenobjekte repräsentieren kann. Zusammen mit der `InstancePolicy` kann sie also als generischer Policy Provider auch in anderen Anwendungen wieder verwendet werden. Für den Testbetrieb ist dieser Policy Provider ausreichend, für den Produktivbetrieb müsste er noch um leistungsfähige Persistenz- und Caching-Mechanismen erweitert werden. So verwendet z.B. die LifeSensor Gesundheitsakte der InterComponentWare AG einen entsprechenden Hochleistungs-Provider, um den Zugriff auf die Gesundheitsdaten von Kunden zu kontrollieren [10]. Die Einbindung erfolgt, wie oben beschrieben, durch das SAF.

Vergleich

Dieser Abschnitt liefert einen kurzen Vergleich der hier vorgestellten Lösung mit dem Acegi Security Framework [11] und den JBoss Seam Security Funktionalitäten [12]. Acegi Security ist ein mächtiges Authentifizierungs- und Autorisierungs-Framework und ist in vielen Spring-Anwendungen produktiv im Einsatz. Im Vergleich zu unserer Lösung basiert Acegi's Policy Decision Funktionalität aber nicht auf Java Security Standards sondern auf einer proprietären Lösung. Eigene `AccessDecisionManager` prüfen hier den Zugriff auf die Domänenobjekte einer Anwendung unter Verwendung von Access Control Lists (ACLs). Eine ACL definiert pro Domänenobjekt-Instanz, wer darauf wie zugreifen darf. Beim Policy Enforcement greifen sowohl Acegi als auch SAF auf Spring AOP und AspectJ zurück. Jedoch ist bei Acegi mehr Konfigurationsaufwand notwendig. Annotationen auf Domänenobjekten werden ebenfalls (noch) nicht unterstützt. Hier ist zusätzliche Arbeit durch den Entwickler notwendig. Wer die einfachen Konfigurations- und die vielseitigen Annotierungsmöglichkeiten des SAF zusammen mit der Policy Decision Funktionalität von Acegi nutzen möchte, könnte dies über einen Adapter machen, der das SAF `AccessManager` Interface implementiert und Anfragen an das Acegi ACL Subsystem bzw. an einen `AccessDecisionManager` weiterleitet.

JBoss Seam geht bei der Policy Decision Funktionalität einen anderen Weg und legt Zugriffsregeln mit Hilfe von JBoss Rules [13], der Rule Engine von JBoss, fest. Der Zugriff auf die Domänenobjekte einer Anwendung kann somit auf Basis von beliebig komplexen Regeln entschieden werden. Das Policy Enforcement erfolgt bei Seam über spezielle `@Restrict` Annotationen. Soll der Zugriff auf eine Domänenobjekt-Instanz überprüft werden, so muss im Wert der Annotation diese Instanz explizit über einen Variablennamen referenziert sein. Auch hier wäre eine Kombination aus SAF und JBoss Rules denkbar.

Zusammenfassung

Wir haben die Architekturgrundlagen instanzbasierter Zugriffskontrolle und Möglichkeiten zur Implementierung kennen gelernt. Dabei wurde eine klare Trennung zwischen Policy Enforcement und Policy Decision gemacht. Beim Policy Enforcement haben wir die Stellen im Quellode, an denen wir den Zugriff auf Instanzen kontrollieren wollen, mit SAF Annotationen versehen. Das führt während der Laufzeit zu entsprechenden Berechtigungsprüfungen. Das Treffen von Zugriffsentscheidungen delegiert das SAF aber an eine Policy Decision Komponente, die über eine Service Provider Schnittstelle angesprochen wird. Die Policy Decision Komponente unserer Beispielanwendung wurde auf Basis von Java SE Security Standards entwickelt. Es wurden auch Möglichkeiten zur Integration von Policy Decision Funktionalität anderer Security Frameworks besprochen.

Martin Krasser arbeitet als Software Architekt bei der InterComponentWare AG. Seine Schwerpunkte sind Sicherheit in Enterprise Anwendungen sowie die Entwicklung von Anwendungs- und Integrationsplattformen im eHealth Umfeld.

Links & Literatur

[1] Security Annotation Framework http://safr.sourceforge.net/ und http://sourceforge.net/projects/safr
[2] Java EE 5 Specification (Abschnitt EE.3.7.2)
http://jcp.org/aboutJava/communityprocess/final/jsr244
[3] Java Security Architecture
http://java.sun.com/javase/6/docs/technotes/guides/security/spec/security-spec.doc.html
[4] JAAS Reference Guide
http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html
[5] SAF JAAS Modul http://safr.sourceforge.net/safr-jaas/
[6] SAF Notebook Beispiel http://safr.sourceforge.net/safr-sample-notebook
[7] SAF Core Modul http://safr.sourceforge.net/safr-core/
[8] Spring AOP APIs http://static.springframework.org/spring/docs/2.0.x/reference/aop-api.html
[9] AspectJ http://www.eclipse.org/aspectj/
[10] LifeSensor Gesundheitsakte http://www.lifesensor.de
[11] Acegi Security System for Spring http://acegisecurity.org/
[12] JBoss Seam http://www.jboss.com/products/seam
[13] JBoss Rules http://www.jboss.com/products/rules