Konfigurierbare Sicherheit für Java Laufzeitumgebungen

Thomas Bühren[2] · Volker Gruhn[1] · Dirk Peters[1]

[1]Universität Leipzig · Lehrstuhl für Angewandte Telematik / e-Business *
{gruhn, peters}@ebus.informatik.uni-leipzig.de

[2]Peperoni GmbH
buehren@peperoni.de

Zusammenfassung

Bei der Entwicklung von Java-Applikationen stehen Entwicklern eine Reihe von Mechanismen zur Verfügung, unsichere Kommunikationskanäle mit Verschlüsselung abzusichern. Diese Techniken sind dann in der Regel tief im Programmcode verwoben. Für Sicherheitsadministratoren ist die Verwaltung dieser Programme dann oft nicht einfach.
Die Ergebnisse dieser Arbeit ermöglichen es, die von Anwendungen genutzten Sicherheitsdienste unabhängig von der Software-Entwicklung erst zum Zeitpunkt der Installation hinzuzufügen und zu konfigurieren. Dadurch wird die Anwendungsentwicklung von Sicherheitsaspekten befreit, was eine Verringerung von Aufwand und möglichen Fehlerquellen verspricht. Sicherheitsmerkmale können so auch nachträglich zu Anwendungen hinzugefügt werden, wobei die Benutzung und die Parametrisierung sämtlicher Sicherheitsdienste für jede Installation individuell festgelegt werden können.
Das Java-Sicherheitsmodell wird so erweitert, dass nicht nur die Zugriffskontrolle, sondern auch weitere Sicherheitsmechanismen in einer Sicherheitspolitik definiert werden können und nicht bei der Entwicklung von Anwendungen vorgegeben werden müssen. Beispielsweise kann für Dateien oder Netzverbindungen neben den Zugriffsberechtigungen auch eine Verschlüsselung oder Integritätsprüfung konfiguriert werden. Dazu wurde eine Bibliothek von Klassen entwickelt, die in Verbindung mit einer für diese Zwecke weiterentwickelten Java Virtual Machine eingesetzt werden kann und die erforderliche Funktionalität bietet. Dabei wurden einerseits existierende Sicherheitsdienste eingebunden, andererseits aber auch neue Funktionalität zu dieser Bibliothek hinzugefügt.

1 Einleitung

Damit in der Java-Laufzeitumgebung eine Anwendung, eine Komponente oder eine Klasse aus einer Klassenbibliothek auf sicherheitsrelevante Methoden oder Systemressourcen wie das Dateisystem oder Netzwerkverbindungen zu anderen Rechnern zugreifen darf, muss diesem Software-Teil zunächst die Erlaubnis dazu gegeben werden. Dies geschieht durch die Definition einer Sicherheitspolitik, die auch jetzt schon je nach Installation angepasst werden kann.

*Der Lehrstuhl für Ang. Telematik / e-Business ist ein Stiftungslehrstuhl der Deutschen Telekom AG.

Die Ressource, für die eine solche Erlaubnis gelten soll, kann genau bestimmt werden. Zum Beispiel kann eine Zugriffserlaubnis für eine Datei, ein Verzeichnis oder für einen bestimmten Netzwerk-Port auf einem bestimmten Rechner definiert werden.

Durch Angabe einer Herkunftsquelle werden die Klassen festgelegt, für die eine Erlaubnis gilt. Eine Quelle kann unter anderem ein lokales oder ein entferntes Verzeichnis sein. Zusätzlich zum Herkunftsort kann gefordert werden, dass die Klassen von einem angegebenen Unterzeichner signiert wurden, um deren Herkunft beweisen zu können; damit wird auch ein Austausch von Klassen gegen namensgleiche Klassen mit fremdem Inhalt verhindert. Mit diesen Mitteln kann definiert werden, welche Klassen mit welcher Herkunft auf welche Ressourcen zugreifen können. Es kann jedoch nicht festgelegt werden, in welcher Weise der Zugriff geschieht.

Sollen vertrauliche Informationen über eine nicht vertrauenswürdige Netzwerkverbindung übertragen werden, so muss der Entwickler der Klassen dafür Sorge tragen, dass entsprechende Maßnahmen wie Verschlüsselung auch eingesetzt werden. Der Anwender, der die Klassen später einsetzt, muss also darauf vertrauen, dass der Entwickler die Sicherheitsaspekte bedacht und korrekt implementiert hat. Dieses Vertrauen ist aus mehreren Gründen problematisch. Zum einen könnte es sich um Klassen oder Komponenten handeln, die ursprünglich nicht unter Berücksichtigung der für den Anwender wichtigen Schutzziele entwickelt wurden. Ebenso kann sich die Einsatzumgebung von Klassen ändern, beispielsweise wenn eine ursprünglich nur im internen Netz ausgeführte Anwendung auch über öffentliche Netze ausgeführt werden soll. Und schließlich können durch die Notwendigkeit der Berücksichtigung von Sicherheitsmerkmalen in jeder einzelnen Klasse potentiell auch jedes mal neue Sicherheitslücken entstehen. Die Belastung der Anwendungsentwickler mit sicherheitsbezogenen Details widerspricht zudem mehreren Prinzipien des Software Engineering [GJM02]:

- Strukturierung.

  Eine Unterteilung in die Aspekte der Sicherheit und der Anwendungslogik ist nicht gegeben. So stellen die Sicherheitsaspekte eine lästige und eventuell vernachlässigte Zusatzaufgabe für die Anwendungsentwickler dar.

- Abstraktion.

  Der Anwendungsentwickler kann sich nicht auf die Anwendungslogik konzentrieren und von den Sicherheitsaspekten abstrahieren.

- Modularität.

  Die Module für Anwendungslogik und Sicherheitsaspekte sind direkt gekoppelt, was eine Wiederverwendung der implementierten Anwendungslogik in verschiedenen Sicherheitsszenarien erschwert.

- Allgemeinheit.

  Die Möglichkeit der Parametrisierung von Software ermöglicht deren Einsatz unter verschiedenen Rahmenbedingungen. Diese Allgemeinheit für verschiedene Sicherheitsszenarien muss jedoch vom Anwendungsentwickler geschaffen werden.

Es gibt zwar verschiedene Lösungsansätze, die allgemein einsetzbare und konfigurierbare Sicherheitsdienste zur Verfügung stellen, aber die Nutzung dieser Dienste bleibt weiterhin

dem jeweiligen Software-Entwickler überlassen. Dadurch kann insbesondere im Hinblick auf komponentenbasierte Software-Entwicklung und den vielfachen Einsatz zugekaufter Komponenten die Gesamtsicherheit eines Software-Systems nicht gewährleistet werden. Darum wird die Reichweite der konfigurierbaren Sicherheitspolitik in dieser Arbeit so erweitert, dass nicht nur eine Kontrolle der Zugriffe auf Systemressourcen, sondern auch die Zuschaltung weiterer Sicherheitsmechanismen damit möglich ist.

2 Techniken zur Anbindung eines Sicherheitsframeworks

Ziel der Entwicklung des Sicherheitsframeworks JSEC (Java Security Enforcement and Configuration) ist die Integration von Sicherheitsmechanismen in bestehende Anwendungen, ohne deren Quelltexte zu modifizieren. Dies könnte mit zunächst geringem Aufwand auch ohne einen Wrapping-Mechanismus erreicht werden, indem die Klassen der Java-Klassenbibliothek so verändert werden, dass zusätzlich zur vorhandenen Funktionalität auch die gewünschten Sicherheitsalgorithmen ausgeführt werden.

Dieses Vorgehen ist jedoch in zweierlei Hinsicht inflexibel. Zum einen werden die einsetzbaren Sicherheitsmechanismen dabei vom Entwickler dieser veränderten Klassenbibliothek festgelegt. Soll später von Dritten eine Modifikation der Algorithmen oder eine Erweiterung um zusätzliche Mechanismen erfolgen, so müssen dafür erneut Änderungen an den zuvor bereits geänderten Quelltexten der Klassenbibliothek vorgenommen werden. Dies führt zu erhöhtem Aufwand im Konfigurationsmanagement, denn vor dem Einsatz eines neuen Release der Java-Laufzeitumgebung müssen zunächst die dazugehörigen Quelltexte der Klassenbibliothek von Sun veröffentlicht werden und anschließend von allen beteiligten Parteien nacheinander erneut um die von ihnen ergänzte Funktionalität erweitert werden.

2.1 Wrapper / Autarke Wrapper

Unser Lösungsansatz ist, die vorhandenen Klassenbibliotheken zu wrappen und dadurch Sicherheitsmechanismen nachträglich einzufügen. Die Grundidee des Wrapping-Ansatzes besteht darin, beliebige Klassen der Klassenbibliothek oder der eingesetzten Komponenten und Anwendungen zur Laufzeit durch andere Klassen, nämlich die Wrapper-Klassen, ersetzen zu können. Dies ermöglicht die Veränderung, Kontrolle oder Protokollierung des Verhaltens eines Software-Systems, ohne dessen Quelltexte, compilierte Klassendateien oder Installation verändern zu müssen.

Ein Wrapper wird vom Rest des Systems statt der durch ihn ersetzten Klasse aufgerufen und muss somit neben dem beabsichtigten Zusatznutzen auch deren ursprüngliche Aufgaben erfüllen. Um die vorhandene Funktionalität nicht innerhalb des Wrappers vollständig neu implementieren zu müssen (autarker Wrapper), bietet sich in den meisten Fällen eine Nutzung der eigentlich ersetzten Klasse durch den Wrapper an. Grundvoraussetzung dafür ist zunächst lediglich, dass die ersetzte Klasse vom Wrapper zur Laufzeit auch angesprochen werden kann. Beziehungen des Wrappers zu dieser Klasse müssen zur Laufzeit unverändert bleiben, da die Wrapper-Klasse sich selbst referenzieren würde, wenn sie auch dann durch den Wrapper ersetzt würde. Konfiguration und Durchsetzung von Sicherheitsspezifikationen mit Hilfe einer erweiterten Java-Laufzeitumgebung Wrapping Bei der konkreten Implementierung eines Wrappers kann die Funktionalität der gewrappten Klasse auf alle in Java auch für herkömmliche Klassen verfügbaren Arten wiederverwendet

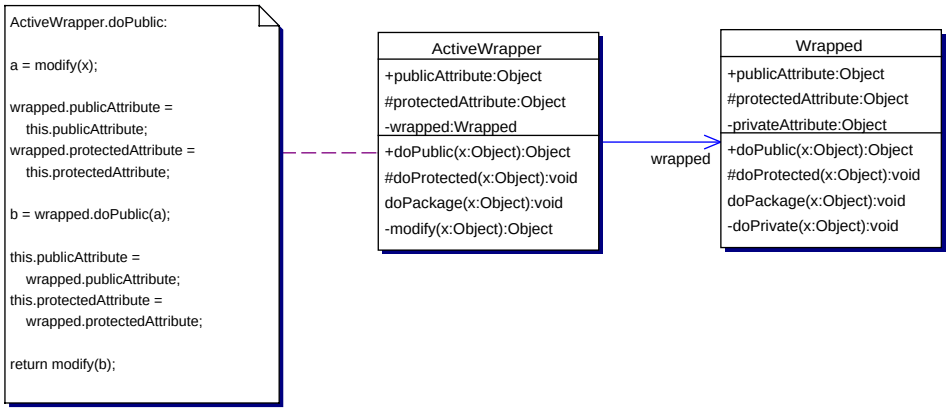

Abbildung 1: Aktiver Wrapper und Aufrufe von Methoden der gewrappten Klasse

werden. Abhängig von der bei der Implementierung eines Wrappers gewählten Referenzierung der ursprünglichen Klasse ergeben sich unterschiedliche Eigenschaften, die im folgenden vorgestellt werden.

2.2 Aktive Wrapper

Eine Nutzung der gewrappten Klasse für ihre ursprüngliche Aufgabe ist beispielsweise durch Instanziierung eines Objekts aus dieser Klasse möglich, wie in Abbildung 1 als Klassendiagramm in der Unified Modeling Language (UML) gezeigt. Jedes Wrapper- Objekt hält damit im Hintergrund ein Originalobjekt bereit, an das es die Methodenaufrufe weiterreichen kann, nachdem oder bevor die für den Zweck des Wrappers nötigen Zusatzschritte durchgeführt werden, beispielsweise eine Verschlüsselung oder Entschlüsselung der Daten in den Aufrufparametern. Da der Wrapper dabei aktiv die übergebenen Daten ändert, wird ein solcher im folgenden als aktiver Wrapper bezeichnet. Dabei ist nicht ausschlaggebend, ob die Anweisungen zur Manipulation der Parameter in der Wrapper-Klasse oder einer zusätzlichen Klasse implementiert sind.

Damit der Wrapper statt der gewrappten Klasse verwendet werden kann, muss er die gesamte von außen zugängliche Schnittstelle mit Methoden und Attributen nachbilden und die Methodenaufrufe auch bei nicht modifizierten Methoden an das Originalobjekt weiterleiten. Ebenso wie für Methodenaufrufe wird auch für die von außen zugänglichen Attribute die Wrapper- Instanz benutzt, so dass diese den aktuellen Zustand enthält. Daher müssen die Werte der Attribute vor dem Aufruf der Originalmethode in das Originalobjekt und danach zurück zum Wrapper kopiert werden.

2.3 Passive Wrapper

Statt die Aufrufparameter einer Methode zu verändern und an ein Objekt der gewrappten Klasse weiterzureichen, kann ein Wrapper die Parameter auch unverändert lassen und an eine spezielle Klasse weiterleiten, die ihrerseits die gesamte, um zusätzliche Maßnahmen erweiterte, Funktionalität der gewrappten Klasse bereitstellt, wie in Abbildung 2 dargestellt.

Auf diese Weise können Änderungen an den von außen zugänglichen Attributen erfasst

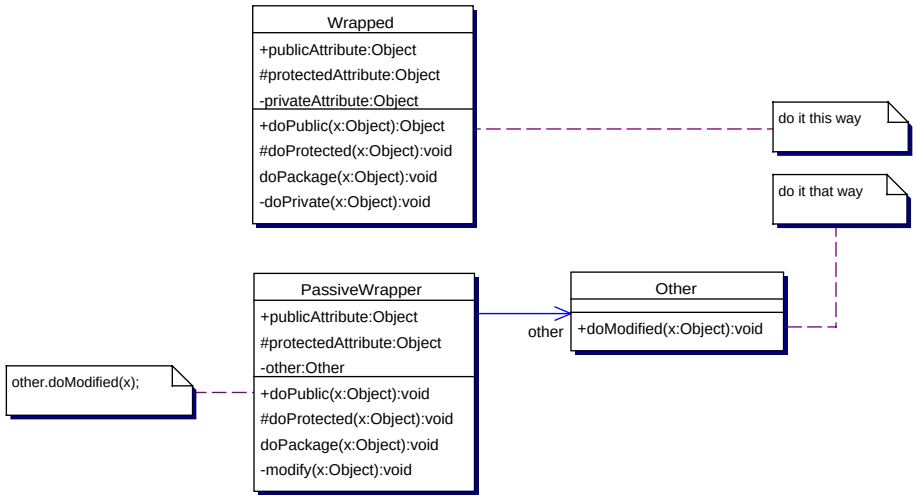

Abbildung 2: Passiver Wrapper und Aufrufe von Methoden einer separaten Klasse

werden, bevor und nachdem die Methode des Originalobjekts ausgeführt wird. Es entstehen allerdings Schwierigkeiten, wenn Attribute während der Ausführung der Methode verändert und noch vor ihrem Verlassen von anderen Objekten benötigt werden. Dies kann dann der Fall sein, wenn von der Methode ein Attribut des Originalobjekts geändert und danach ein Event ausgelöst wird, das von anderen Objekten empfangen wird und zum Lesen des Attributs aus dem Wrapper führt. Diese Fälle sind nur schwer und unter Berücksichtigung von Details der gewrappten Klasse zu berücksichtigen; bei einer sauberen Implementierung der zu wrappenden Klassen – insbesondere aus der Java-Klassenbibliothek – kann allerdings davon ausgegangen werden, dass alle Attribute privat sind und auf sie nur durch Set- und Get-Methoden zugegriffen werden kann, so dass im Wrapper kein duplizierter Zustand verwaltet wird.

Diese passiven Wrapper sind nur dann sinnvoll, wenn eine passende Version der zu ersetzenden Klasse bereits in einer Klassenbibliothek zur Verfügung steht. Ein passiver Wrapper ist bis auf die identische Schnittstelle von der gewrappten Klasse vollständig unabhängig, die zur Laufzeit dann unbenutzt bleibt. Somit entsteht keine Kopie der Werte öffentlicher Attribute, auch wenn diese im Wrapper erneut deklariert werden müssen.

2.4 Schlanke Wrapper

Sowohl aktive als auch passive Wrapper müssen sämtliche in der öffentlichen Schnittstelle der gewrappten Klasse vorhandenen Methoden anbieten und zu einem anderen Objekt umleiten. Um diesen bei der Implementierung und zur Laufzeit auftretenden Mehraufwand zu vermeiden, kann ein Wrapper auch von der gewrappten Klasse erben und muss somit nur die tatsächlich zu verändernden Methoden überschreiben. Durch die in Java implementierte Polymorphie kann ein solcher schlanker Wrapper ebenfalls anstelle der ursprünglichen Klasse eingesetzt werden, ohne die öffentliche Schnittstelle vollständig neu zu implementieren. Die Attribute sind nicht in zwei verschiedenen Instanzen vorhanden und müssen daher auch nicht kopiert werden. In den überschreibenden Methoden werden, falls nötig, wie bei einem aktiven Wrapper die Parameter verändert, in diesem Fall aber an die Superklasse weitergegeben. Diese Situation ist in Abbildung 3 dargestellt.

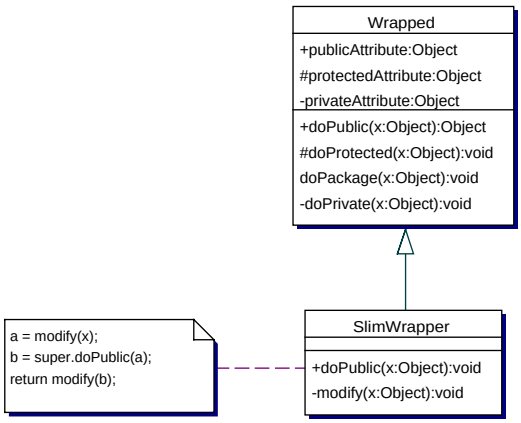

Abbildung 3: Schlanker Wrapper als Subklasse der gewrappten Klasse

2.5 Vergleich der Wrapper-Typen

Schlanke Wrapper führen zu geringem Aufwand und guter Überschaubarkeit bei der Implementierung, denn das veränderte Verhalten der Superklasse kann sehr einfach in der gewohnten objektorientierten Weise durch eine Vererbungsbeziehung und Polymorphie hergestellt werden. Gleichzeitig wird ein besseres Laufzeitverhalten als bei den anderen Typen erreicht; denn nur bei überschriebenen Methoden wird ein Zusatzaufwand verursacht. Alle anderen Methoden der gewrappten Klasse werden weiterhin ohne Umweg über den Wrapper direkt angesprochen. Ein aktiver Wrapper bietet sich nur in Ausnahmefällen an, beispielsweise wenn dieser von einer anderen Klasse als der gewrappten Klasse erben soll; da Mehrfacherbung in Java nicht möglich ist, muss dazu dieser Umweg in Kauf genommen werden. Allerdings ist dann auch die Aktualisierung eventuell vorhandener öffentlicher Attribute zu berücksichtigen, da deren Zustand in zwei Objektinstanzen gespeichert ist. Autarke Wrapper sind nur dann sinnvoll, wenn das Verhalten einer Klasse vollständig verändert werden soll und nicht auf der ursprünglichen Klasse basieren kann. Ist das neue Verhalten bereits in einer anderen Klasse implementiert, kann statt dessen ein passiver Wrapper genutzt werden.

2.6 Wrapper Beziehungen

In Java kann das Überschreiben einzelner oder aller Methoden einer Klasse durch Einschränkung der Zugriffsrechte verhindert oder auf Klassen desselben Package eingeschränkt werden. Insbesondere ein schlanker Wrapper würde damit auf die von der gewrappten Klasse vorgesehenen Manipulationsmöglichkeiten eingeschränkt. Autarke, aktive und passive Wrapper können dagegen eine Kopie der öffentlichen Klassenschnittstelle zur Verfügung stellen, wobei die aktiven Wrapper aber durch einschränkende Zugriffsrechte am Aufruf von Methoden der ursprünglichen Klasse gehindert werden können, wenn sie nicht darauf zugreifen können. Darum muss ein Wrapper mit der von ihm gewrappten Klasse in eine spezielle Wrapping- Beziehung gebracht werden, die in Java bisher nicht vorgesehen ist. Dabei werden die Zugriffsmöglichkeiten der Wrapper auf die von ihnen gewrappten Klassen erweitert, so dass auch ein Aufrufen oder Überschreiben geschützter Methoden und Klassen möglich ist.

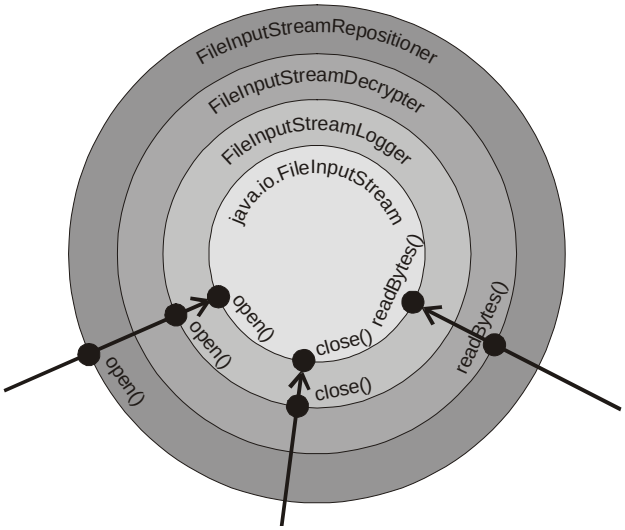

Abbildung 4: Geschachtelte Wrapper und überschriebene Methoden

2.7 Mehrfaches Wrapping

Die Funktionalität einer gewrappten Klasse kann unter verschiedenen Gesichtspunkten verändert werden. Für die Klassen zum Lesen und Schreiben von Dateien bieten sich folgende Manipulationen an:

- Verschlüsselung und Entschlüsselung zur Sicherstellung der Vertraulichkeit von Daten,
- Signierung und spätere Prüfung der Integrität von Dateien,
- Autorisierung zur Verhinderung unberechtigter Zugriffe,
- Logging, um Dateisystemzugriffe abrechnen oder nachvollziehen zu können,
- Umlenken in andere Verzeichnisse, um nur Teile des Dateisystems für Zugriffe freizugeben und der Anwendung dennoch einen vollständigen Zugriff vorzuspiegeln, sowie
- Debug-Hilfen, beispielsweise durch Bildschirmausgaben bei bestimmten Zugriffen.

Bei der Entwicklung eines Wrappers für eine bestimmte Klasse können nicht sämtliche denkbaren Einsatzmöglichkeiten bedacht werden. Zudem wird ein Wrapper durch Kombination Konfiguration und Durchsetzung von Sicherheitsspezifikationen mit Hilfe einer erweiterten Java-Laufzeitumgebung Wrapping mehrerer Zielsetzungen unübersichtlich und die Ausführung unnötig gebremst, wenn nicht alle Möglichkeiten dieses großen Wrappers genutzt werden sollen. Es ist also vorteilhaft, mehrere Wrapper jeweils für eine spezielle Aufgabe zu entwickeln, die aber auch gemeinsam eingesetzt werden können. Dies wird durch eine Schachtelung von Wrappern erreicht, womit ein Wrapper selbst wieder gewrappt werden kann. Durch die Hintereinanderschaltung mehrerer spezieller Wrapper sind so verschiedene Modifikationen an einer Klasse möglich. In Abbildung 4 wird ein Beispiel der Schachtelung mehrerer Wrapper für die Systemklasse java.io.FileInputStream veranschaulicht. Die verschiedenen Wrapper verändern dabei genau die Methoden, für die es bei der jeweiligen Aufgabe nötig ist.

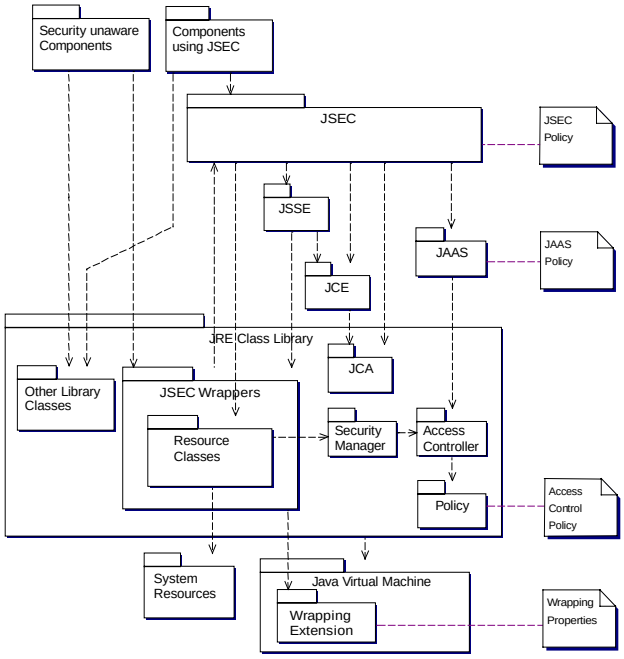

Abbildung 5: Einordnung von JSEC in die Java Sicherheitsarchitektur

Um die einzelnen Wrapper auch unabhängig voneinander einsetzen und entwickeln zu können, darf keine gegenseitige Kenntnis vorausgesetzt werden. Jeder Wrapper muss einzeln so entwickelt werden, dass er zur ursprünglich gewrappten Klasse passt. Zur Laufzeit werden die unabhängig compilierten Wrapper dann je nach konfigurierter Reihenfolge miteinander verwoben. Autarke und passive Wrapper stellen hierbei ein Problem dar, da sie die Aufgaben nicht an die gewrappte Klasse weiterleiten und somit keine Verbindung zum nächstinneren Wrapper hergestellt werden kann. Diese Wrapper können daher nur als erster bzw. innerster Wrapper einer Klasse eingesetzt werden. Das bedeutet auch, dass nur ein einziger der eingesetzten Wrapper einer Klasse ein autarker oder passiver Wrapper sein kann.

3 Konzeption der Änderungen am Java-Laufzeitsystem

Um das vorgestellte Wrapping zu ermöglichen, müssen zwei grundlegende Abläufe des Java- Laufzeitverhaltens verändert werden. Erstens muss statt einer gewrappten Klasse von anderen Klassen ihr Wrapper aufgerufen werden, und zweitens erfordert ein Wrapper Zugriffsrechte auf die gewrappte Klasse, die über die ursprünglich von Java erlaubten Möglichkeiten hinausgehen.

In diesem Kapitel werden wir die Technischen Aspekte der Wrappingmechanismen und deren konzeptionelle Einordnung in die Sicherheitsarchitektur erörtern (siehe Abbildung 5).

4 Konfiguration der Sicherheitsbibliothek

Ein JSEC Policy File enthält Regeln (Rules) zur Erzeugung von Engines. Jede Rule enthält eine Bedingung (Condition), die einen oder mehrere ContextAspects prüft. Wird

```
<policy>
 <rule>
   <condition>
     <access class="java.io.FilePermission"
             target="&lt;&lt;ALL FILES&gt;&gt;" action="read,write"/>
   </condition>
   <implication>
     <apply class="de.xxxxx.jsec.engines.FileStreamCipherEngine">
       <option name="cipher.algorithm">DES</option>
       <option name="keystore.key.alias">mykey</option>
     </apply>
   </implication>
 </rule>
</policy>
```

Abbildung 6: Beispiel einer JSEC Policy-File mit einer rule

die Bedingung vom ExecutionContext erfüllt, so werden die in der Implication angegebenen Engines für den angegebenen Zugriffstyp erzeugt.

JSEC Policy Files sind XML basiert. Grundelement der JSEC Policy Files in XML ist das Tag `<rule>`, das stets eine `<condition>` und eine `<implication>` enthält. Ein Policy File kann eine oder mehrere Rules enthalten. In Abbildung 6 ist ein Beispiel für ein XML Policy File mit nur einer Rule dargestellt:

- Die Condition enthält gemäß der DTD genau einen Operator, in diesem Fall das Tag `<access>`, das die Klasse einer Permission sowie deren Target-Namen und Actions als Parameter enthält, hier also eine `java.io.FilePermission "<<ALL FILES>>", "read,write"`. Das `<access>`-Tag prüft, ob der ausgeführte Zugriff von der angegebenen Permission-Klasse abgedeckt wird; trifft die Condition zu, so wird die Implication ausgewertet.

- Eine Implication kann gemäß DTD einen oder mehrere Activators enthalten; im Beispiel ist lediglich ein `<apply>`-Tag angegeben, das eine `FileStreamCipherEngine` anwendet, also eine Engine zum Ver- und Entschlüsseln von Dateiströmen. In den `<option>`- Tags wird im Beispiel der DES-Algorithmus unter Benutzung des Schlüssels `mykey` aus dem Keystore vorgegeben.

Es gibt zusätzlich Operatoren, die eine Definition logischer Ausdrücke ermöglichen, um eine Condition aus mehreren Bedingungen kombinieren zu können; dies sind `<and>`, `<or>`, `<not>` sowie `<true>` und `<false>`. Gemäß Ihrer Funktionalität enthalten Sie keinen, genau einen oder mehrere andere Operatoren.

Durch weitere Techniken der Policy-Kombinationen wird so eine flexible Konfigurierbarkeit ermöglicht.

5 Anwendung an einem Beispiel der Versicherungsbranche

Als Validierungsbeispiel dient eine Softwarearchitektur die auf der Versicherungsanwendungsarchitektur (VAA) beruht. Dabei wird das folgende Szenario angenommen: Eine imaginäre Versicherungsgruppe nutzt eine VAA-Anwendung innerhalb ihrer zentralen Niederlassung zur Durchführung aller Geschäftsprozesse. Das Fehlen von Sicherheitsmechanismen wird dabei toleriert, da die Anwendung bisher lediglich auf Rechnern innerhalb des internen Netzes ausgeführt wird.

Um auch die Vertriebspartner in die elektronisch durchgeführten Geschäftsprozesse integrieren zu können, sollen nun deren Büro- und Außendienstrechner mit der Client-Software des Workflow-Systems ausgestattet werden und auf die Server in der Zentrale zugreifen. Die Anbindung soll dabei aus Kostengründen über das Internet erfolgen, womit eine Sicherung der Verbindungen notwendig wird.

Da die Anwendung in Java implementiert ist, kann sie in den sehr unterschiedlichen Umgebungen bei den verschiedenen Vertriebspartnern eingesetzt werden. Diese Inhomogenität macht jedoch eine Nutzung von Betriebssystemfunktionalität für Sicherheitsmechanismen wie IPSEC unmöglich, da die Vielzahl von Plattformen eine zentrale Sicherheitsadministration erschwert und einige der eingesetzten Betriebssysteme keine Sicherheitsdienste anbieten. Zusätzlich lässt sich IPSEC nicht gut genug in die Applikation integrieren, so dass Sicherheitskerne zu weit gefasst werden.

Daher sollen die Sicherheitsmechanismen in die Java-Anwendung integriert werden, so dass Daten bereits vor dem Verlassen der Virtual Machine verschlüsselt werden und bei Netzwerkverbindungen innerhalb des Programms eine Authentifizierung der Gegenstelle ausgeführt werden kann.

Eine Veränderung der bestehenden Anwendung ist jedoch problematisch, da Komponenten eingesetzt werden, die von externen Zulieferern entwickelt wurden und zukünftig in neuen Versionen weiterentwickelt werden, was eine dauerhafte Pflege eigener Varianten mit Sicherheitsfunktionalität erschwert. Zudem müssten die Entwickler im Bereich Sicherheit ausgebildet werden und alle Anwendungsteile auf eventuelle Sicherheitslücken untersucht werden. Externe Sicherheitsspezialisten müssten andererseits zunächst die Implementierung des Systems kennenlernen. Daher soll die Bibliothek JSEC in Verbindung mit der Java Virtual Machine mit Wrapping- Erweiterung eingesetzt werden, um die bestehende Anwendung mit Sicherheitsmechanismen auszustatten. Diese separate Schicht der Anwendung kann leichter auf Sicherheitslücken untersucht werden, da beispielsweise kryptographische Schlüssel nicht von anderen Anwendungsteilen benötigt werden. Die Sicherheitspolitik kann von einem Sicherheitsadministrator ohne weitreichende Kenntnis der Anwendung definiert werden.

Durch die JSEC-Bibliothek lässt sich eine Sicherung der Kommunikationsbeziehung mit oben genannten Anforderungen erreichen. Die JSEC-Policy zur Konfiguration der gewünschten Sicherheitsmechanismen besteht aus drei Teilen: Neben separaten XML-Dateien für Server und Clients werden gemeinsame Definitionen in einer weiteren Datei vorgenommen. Beim Programmstart wird java.lang.SecurityManager aktiviert und die Sun-Provider für JCA, JCE und JAAS geladen. Zur Überwachung des Systems wer-

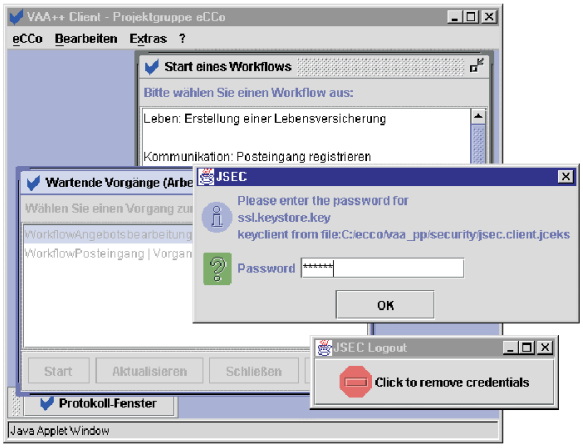

Abbildung 7: Zusätzliche JSEC-Dialoge in der Anwendung

den Netzwerk- und Dateizugriffe protokolliert, wobei die Programm- und die JSEC-Konfigurationsdateien dabei ausgenommen sind. Auf den Clients wird für alle Socket-Verbindungen SSL vorgeschrieben, wobei der Algorithmus TLS und sofortiger Handshake beim Aufbau der Verbindung definiert sind.

Auch auf dem Client werden ServerSockets verwendet, weshalb auch dafür ein Konfigurationseintrag in der Client-Policy enthalten ist. Der in beiden Fällen benutzte private Schlüssel mit dem Namen keyclient wird im Keystore jsec.client.jceks erwartet. Der Truststore jsec.trust.jceks ist ebenfalls ein Keystore und enthält das Zertifikat der Versicherung als Zertifizierungsstelle; eine Verbindung wird nur zu Servern aufgebaut, deren Schlüssel mit dem zu diesem Zertifikat gehörenden privaten Schlüssel signiert wurden. Die zum Öffnen der Keystores und Lesen des privaten Schlüssels notwendigen Kennwörter sind nicht in der Konfigurationsdatei enthalten, sondern werden mit Swing-Dialogen vom Benutzer erfragt (siehe Abbildung 7)

Die Credentials, also Keystores, Schlüssel und Zertifikate, werden von einem GlobalCredentialManager im Arbeitsspeicher zwischengespeichert und entweder per Klick auf einen Swing-Logout-Button oder im Beispiel nach 4 Minuten wieder gelöscht. Auf Server-Seite wird ebenfalls SSL für alle Sockets vorgeschrieben, wobei eine Client- Authentifizierung gefordert wird, so dass nur Clients akzeptiert werden, die ebenfalls SSL einsetzen und das im Truststore des Servers vorhandene Zertifikat für ihren privaten Schlüssel vorweisen können.

6 Vergleich mit existierenden Techniken

Die Trennung von Sicherheitscode und Programmcode in Applikationen ist enorm wichtig, um auch größere Anwendungen sicherheitstechnisch wartbar zu machen. Ein Grundpfeiler in Java-Anwendungen ist hier der Java Authentication und Authorisation Service (JAAS) [LGK+99]. Mit Hilfe von JAAS lassen sich Authentifizierungen und Autorisierungen in Applikationen einarbeiten, ohne sich auf ein konkretes Nutzermanagement oder eine Autentifizierungstechnik in der Applikation festlegen zu müssen. Nutzer werden durch die Klasse `Subject` dargestellt, und Rechte durch eine Menge von Credentials vergeben. Im Gegensatz zu JSEC müssen JAAS-Verankerungen im Code untergebracht werden. Die

JAAS-Techniken bilden jedoch eine Grundlage der JSEC-Umsetzung.

Vertraulichkeit läßt sich nicht alleine durch Authentifizierungstechniken gewährleisten, Verschlüsselung und Signaturen sind dabei unverzichtbar. Die gängigen Verschlüsselungs- und Signaturtechniken werden mit der Java Cryptography Architecture (JCA), welche in Standardlaufzeitungebungen mitgeliefert wird, sowie dem zusätzlichen Paket Java Cryptography Extension (JCE) angeboten [Knu98, MDOY98]. Das Problem bei dem Einsatz dieser Techniken ist, dass Applikationsentwickler oft Gefahr laufen, sicherheitsrelevanten Code zu eng mit Applikationscode zu verweben. Der Einsatz von JSEC kann hier die nötige Trennung bringen (siehe Abbildung 5).

Die Idee, Vertrauensbereiche zu konfigurieren kommt dem JSEC-Ansatz schon näher. In Java ist es möglich, über Security Policies Vertrauensbereiche zu definieren und Informationen an den Grenzen dieser Vertrauensbereiche zu blockieren [Gon99]. Hierbei handelt es sich um eine Erweiterung des Sandbox-Modells, welches ursprünglich für Java-Applets aus unsicherer Quelle entwickelt wurde. Der Schutz dieser Vertrauensbereiche zielt daher auch eher auf vertrauensunwürdigen Code, als auf vertrauliche Daten ab.

In einem Ansatz der TU Dresden wurde eine Plattform zur Bereitstellung multiliteraler Sicherheit in verteilten Anwendungen geschaffen [PSW+98]. Hier lassen sich Sicherheitspolitiken nachträglich zuschalten und der Grad der Sicherheit einstellen. In den Betrachteten Szenarien wird besonderer Wert auf die Unabhängigkeit von Kommunikationspartnern gelegt. Anders als bei JSEC ist es jedoch nötig, eine spezielle Klassenbibliothek zu verwenden.

In [LB98] wird vorgeschlagen, zur nachträglichen Modifikation bestehender Klassen deren Bytecode von einem speziellen Class Loader bearbeiten zu lassen. Damit wäre ein Eingriff in die Virtuelle Maschine nicht nötig. Die Nachteile liegen jedoch in der komplexen Behandlung des Bytecodes und in der Tatsache, dass die Java-Klassenbibliothek nicht vom Class Loader geladen wird.

Literatur

[GJM02] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering.* Pearson Intgernational, September 2002.

[Gon99] L. Gong. *Inside Java 2 Platform Security.* Addison Wesley, June 1999.

[Knu98] J. B. Knudsen. *Java Cryptography.* O'Reilly, Mai 1998.

[LB98] Sheng Liang and Gilad Bracha. Dynamic Class Loading in the Java Virtual Machine. Oktober 1998.

[LGK+99] C. Lai, L. Gong, L. Koved, A. Nadalin, and R. Schemers. User Authentication and Authorization in the Java Platform. Dezember 1999.

[MDOY98] R. Macgregor, D. Dudrbin, J. Owlett, and Andrew Yeomans. *JAVA Network Security.* Prentice Hall, Januar 1998.

[PSW+98] A. Pfitzmann, A. Schill, A. Westfeld, G. Wicke, G Wolf, and J. Zöllner. A Java-Based Distributed Platform for Multilateral Security. *IFIP Working Conference on Trends in Distributed Systems / Electronic Commerce*, Juni 1998.