Fraunhofer Institut
Graphische
Datenverarbeitung

Bachelor-Thesis

JAVA-SECURITY IN MEHRBENUTZERSYSTEMEN

von

Renée Bäcker
Matrikelnummer 03BIM02

Provadis School of International Management and Technology AG

Brüningstr. 50

65926 Frankfurt

Betreuer: Dipl.-Inf. Jan Peters

Prüfer:

Prof. Dr. Bodo A. Igler

Prof. Dr. Berthold Franzen

Aufgabenstellung für die Bachelor-Thesis des
Herrn Renée Bäcker
Matrikel-Nr. 03BIM02

Thema: “Java-Security in Mehrbenutzersystemen”

In der Abteilung für Sicherheitstechnologie am Fraunhofer-Institut für Graphische Datenverarbeitung IGD wird an einer Java-basierten Sicherheitsplattform für Softwarekomponenten geforscht, die in ubiquitären Umgebungen Verwendung finden soll. Der Kern der im Rahmen des Projekts SicAri entwickelten und eingesetzten Plattform besteht aus drei Komponenten. Die Kommando-Shell wird vom Plattform-Administrator genutzt, um die Plattform zu starten sowie zur Laufzeit flexibel zu konfigurieren, zu erweitern und zu beobachten. Das sogenannten Environment dient als Dienstumgebung für das lokale Service-Management. Eine Webservice-Erweiterung für dieses Environment ermöglicht die Nutzung lokal gestarteter Dienste in einer verteilten Infrastruktur. Ein Sicherheits-Framework setzt bei der Interaktion von Nutzern, Anwendungen und Diensten Sicherheitspolitiken durch.

Greift ein lokaler Nutzer z.B. auf einen Dienst bzw. eine Systemressource zu, so wird durch das Sicherheits-Framework der Plattform ermittelt, wer der aktuell agierende Nutzer ist, und für diesen Nutzer dann die aktuelle Sicherheitspolitik durchgesetzt. Damit die Durchsetzung von Sicherheitspolitiken auf diese Weise umgesetzt werden kann, muss sich jede agierende Entität auf der Plattform zunächst erfolgreich authentisieren. Die von dieser Entität anschließend ausgeführten Aktionen müssen in einem ihr zugewiesenen Sicherheitskontext ausgeführt werden. Beim Zugriff auf Dienste über das Environment werden dabei zwei verschiedene Paradigmen unterstützt, die bei der Initialisierung der Dienste ausgewählt werden können: Delegation durch impliziten Wechsel des Sicherheitskontexts und Dienstzugriff ohne Wechsel des Sicherheitskontexts. Java bietet bereits einige Designkonzepte, die bei der Implementierung der Sicherheitsplattform Verwendung fanden und in manchen Aspekten erweitert wurden.

Anwendungen, die z.B. auf dem Abstract Window Toolkit (AWT) von Java basieren, untergraben allerdings Javaeigene Sicherheitskonzepte. Desweiteren ist das Bootstrapping der Plattform problematisch, da zu diesem Zeitpunkt das Sicherheits-Framework noch nicht vollständig initialisiert wurde. Durch das oben angesprochene Environment werden lokale Dienste auf der Plattform logisch voneinander getrennt. Diese Trennung beinhaltet allerdings noch nicht die physikalische Trennung bzw. Beschränkung der gemeinsam genutzten physikalischen Ressourcen (Speicher, CPU, etc.). Es gibt also noch einige Aspekte, die bei der Anwendung von Java-Security in einem verteilten Mehrbenutzersystem zu beachten sind, sowie eine Reihe von Stolperfallen und offener Probleme.

Die Bachelor-Arbeit soll folgende Aspekte beinhalten:

- Literaturrecherche und Erstellung von Angriffsszenarien
- Analyse und Bewertung der zur Verfügung stehenden Mechanismen
- Entwurf einer geeigneten Erweiterung der Sicherheitsplattform
- Implementierung und Integration in die Plattform

Ziel der Bachelor-Arbeit ist es, aufbauend auf den in der SicAri-Plattform bereits umgesetzten Sicherheitsmechanismen, die aktuellen Sicherheitsmechanismen von Java in allen drei Ausprägungen des aktuellen Developer Kit (J2ME, J2SE und J2EE) sowie die für die nähere Zukunft geplanten Sicherheitserweiterungen (z.B. Java-Isolates) zu untersuchen. Im Kontext der SicAri-Plattform sollen mögiche Angriffsszenarien sowie offene Sicherheitsprobleme identifiziert, beschrieben und analysiert werden. Anschließend soll die Sicherheitsarchitektur der Plattform geeignet angepasst und getestet werden, um möglichst viele der identifizierten Schwachstellen zu beseitigen.

Darmstadt, den 15. März 2006

Betreuer:
Dipl.-Inf. Jan Peters

Prüfer:

Prof. Dr. Bodo A. Igler

Prof. Dr. Berthold Franzen

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, 26. August 2006 Renée Bäcker

Danksagung

Dies „Buch“ halt ich nun in der Hand,
allein hätt ichs nicht geschafft, das ist bekannt.
Drum sag ich „Vielen Dank“ an alle Leute,
die mir halfen - von Anfang an bis heute.

Ein großer Wunsch der wurde wahr,
Abschlussarbeit beim Fraunhofer wie wunderbar.
Java und Sicherheit warn das Gebiet,
ich wusste kaum was da geschieht.

Doch Unterstützung war nicht weit,
nach Besuchen in Darmstadt bei Jan
ging’s weiter wie befreit.

Dank Provadis und Frau Rieman gabs das Studium,
nach drei Jahr’n hab ich’s geschafft, es ist nun rum.
Auch Eva will ich herzlich grüßen,
dank ihr steh’ ich beruflich auf eig’nen Füßen.

Meine Eltern, Schwester und auch mein Schwager
gehören - genau wie Philip - in das ganz besonders treue Lager.
Dort trifft man Menschen ganz besonders toller Art,
nahmen alle meine Launen hin - Mensch, das war hart.

Was ganz besondres ist mein Schatz,
hatte den schwierigsten Part von allen.
Meine Launen ertragen und noch mehr...
Katrin - vielen Dank, ich lieb’ Dich sehr!

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

In diesem Kapitel werden die Aufgabenstellung der Bachelor-Thesis dargestellt und die Typographie, die in der gesamten Arbeit verwendet wird, erläutert. Nach den Formalien wird ein Überblick über die weiteren Kapitel gegeben.

1.1 Aufgabenstellung

Sicherheit ist ein stetig wachsender Bereich in der Informationstechnik (IT). Um dem Sicherheitsstreben nachkommen zu können, ist es wichtig, Anwendungen auf Schwachstellen hin zu untersuchen und Programmierern ein Framework anzubieten, das die sichere Programmierung erleichtert.
Der ubiquitäre Internetzugang ist die Zukunft für Anwendungen; so soll es in Zukunft möglich sein, dass der Automechaniker an jedem Ort in der Werkstatt aktuelle Informationen zu einem Fahrzeug bekommt.
SicAri verbindet diese beiden Aspekte. SicAri ist ein Forschungsprojekt des Darmstädter Fraunhofer-Instituts für Graphische Datenverarbeitung IGD mit dem Ziel eine Sicherheitsplattform und deren Werkzeuge für den ubiquitären Internetzugang zu schaffen.
Die Plattform ist in Java programmiert und verwendet einige der von Java bereitgestellten Sicherheitsmechanismen.
Der Fokus dieser Bachelor-Thesis liegt auf drei Aufgaben:

- Literaturrecherche
- Analyse eines Teilbereichs von SicAri
- Implementierung eines neuen Sicherheitsaspekts

In der Literaturrecherche sollen die nötigen Grundlagen in IT-Sicherheit, Java-Sicherheit und in der Bedrohungs- und Angriffsanalyse gelegt werden. Diese Grundlagen werden bei der Analyse des Bootstrapping-Vorgangs von SicAri angewendet.
Ausgehend von der Bedrohungs- und Angriffsanalyse des SicAri-Teilbereichs soll ein neuer Bootstrapping-Vorgang erarbeitet werden, der die – in der Bedrohungsanalyse aufgezeigten – Schwachstellen behebt.

1.2 Typographie und Terminologie

Klassennamen und Datentypen sowie Quelltextauszüge sind in `Schreibmaschinenschrift` gesetzt. Um die für einen Absatz relevanten Schlüsselwörter oder Satzteile bzw. Begriffsdefinitionen hervorzuheben, wird ein serifenloser Schriftsatz verwendet. Abkürzungen werden bei ihrem erstmaligem Auftreten explizit erklärt und dann anschließend ohne besondere Hervorhebung verwendet.
Gerade in der Informatik werden viele englische Fachwörter verwendet. Sofern sich diese akzeptabel übersetzen lassen, wird das in dieser Arbeit auch getan. Beschreibt ein englischer Begriff den geschilderten Sachverhalt allerdings besser bzw. hat er direkten Bezug auf Komponenten oder Vorgänge des beschriebenen oder zu entwickelnden Modells, so wird er in seiner ursprünglichen Form belassen. In diesem Fall wird er bei jedem Auftreten *kursiv* gesetzt und für die Konjugation oder Nutzung im Plural nach den Regeln der englischen Rechtschreibung und Grammatik behandelt. Das heißt, dass diese Begriffe klein geschrieben werden, auch wenn sie als Nomen in Gebrauch sind, und im Besonderen keine Genitiv-Form existiert. Eine Ausnahme bilden die im Kontext der Informatik feststehenden, englischen Ausdrücke, die in ihrer ursprünglichen Form teilweise auch groß geschrieben werden. Sie werden behandelt, wie deutsche neu eingeführte Begriffe (siehe oben). Bei der Kombination zwei verschiedensprachiger Wörter, wird kein Bindestrich benutzt, d.h. die einzelnen Wörter werden getrennt nach den genannten Konventionen dargestellt.
In den Abbildungen und in den Spezifikationen im Anhang werden ohne spezielle Hervorhebungen durchgängig die englischsprachigen Begriffe verwendet. Der Quellcode des zu implementierenden Prototyps wird streng nach den $SicAri^{TM}$ *Code Conventions* [8] entwickelt und englisch dokumentiert.

1.3 Überblick über die Kapitel

Diese Bachelor-Thesis ist in acht Hauptkapitel gegliedert, die sich in drei große Teile gliedern.
Die Kapitel 2 bis 4 bilden eine Einleitung in die Grundlagen und beruhen auf der Literaturrecherche. Dabei dient Kapitel 2 als Überblick über die Gesichtspunkte der IT-Sicherheit im Allgemeinen. Hier werden die Ziele der IT-Sicherheit genannt und den Zielen entsprechende Sicherheitsmechanismen vorgestellt. Java bietet viele Mechanismen, diese Ziele der IT-Sicherheit zu erreichen. Wie diese Mechanismen funktionieren und wie sie sich entwickelt haben, wird in Kapitel 3 vorgestellt. Die Fokussierung auf Java wird dadurch erklärt, dass die in diesem Projekt eingesetzte Programmiersprache Java ist. Um die Sicherheit einer Anwendung beurteilen zu können, wird eine Bedrohungs- und Angriffsanalyse für diese Anwendung durchgeführt. Wie eine solche Analyse erstellt wird und mit welchen Angriffen zu rechnen ist, wird in Kapitel 4 beschrieben.
Als Vorbereitung zur Implementierung dienen die Kapitel 5 und 6. In Kapitel 5 wird die Sicherheitsplattform SicAri einführend dargestellt. An einem Bereich von SicAri - dem *Bootstrapping* - wird die Bedrohungs- und Angriffsanalyse durchgeführt. Dieser Bereich ist dann auch der Bereich von SicAri, für den der neue Sicherheitsmechanismus implementiert wird. Die Ergebnisse der Analyse werden in Kapitel 6 vorgestellt.
Kapitel 7 beschreibt den alten *Bootstrapping*-Vorgang und wie es nach der Implementierung aussieht. In diesem Kapitel werden die durchgeführten Veränderungen erläutert.
Das abschließende Kapitel der Bachelor-Thesis fasst die Arbeit zusammen und gibt einen Ausblick auf mögliche zukünftige Veränderungen.

Kapitel 2

Grundlagen der IT-Sicherheit

In diesem Kapitel werden die Grundlagen zum Thema „IT-Sicherheit“ gelegt. Diese Grundlagen sind unabhängig von Systemen oder Programmiersprachen. In Abschnitt 2.1 geht es um die allgemeinen Schutzziele der Informationstechnik (IT) und in Abschnitt 2.2 um Schutzmechanismen.

IT-Systeme werden oft das Ziel von Angriffen. Dabei geht es nicht um rein physische Angriffe, sondern um Attacken durch Viren und Würmer, *Denial-of-Service (DoS)*-Attacken oder Ausführung von anderem Schadcode. Dadurch wird das Thema „Sicherheit“ in Informatik-Kreisen immer wichtiger. Mangelnde IT-Sicherheit hat oft weitreichende Folgen für die betroffenen Firmen.
Dass der Wunsch nach mehr IT-Sicherheit nicht aus dem Nichts kommt, zeigt die Anzahl der IT-Angriffe, die in den letzten Jahren exponentiell gestiegen ist (siehe Abbildung 2.1).

Dabei kommen die Angreifer nicht nur von außen, sondern sind vielfach unter den eigenen Mitarbeitern zu finden. Der Angreifer macht dies aber nicht immer absichtlich - ein unsicher programmiertes Stück Software ist das Sprungbrett für einen erfolgreichen Angriff und kann enormen Schaden im Unternehmen anrichten.

Bevor es im nächsten Kapitel um die Java-spezifischen Sicherheitsmechanismen geht, werden zunächst die Ziele der IT-Sicherheit erklärt und ein Überblick über einen Teil der allgemeinen existierenden Sicherheitsmechanismen gegeben.

2.1 Sicherheitsziele

Für Unternehmen ist ein hoher Grad an Sicherheit sehr wichtig. Aus diesem Grund werden oftmals Strategien für die IT-Sicherheit entwickelt. Claudia Eckert nennt in [10] sechs wichtige Ziele, die erreicht werden sollten. Die Einhaltung dieser Ziele ist sehr wichtig für den reibungslosen Ablauf von IT-Systemen. Neben der Datenintegrität und Informationssicherheit gehören auch Verbindlichkeit und Verfügbarkeit zu den Zielen der IT-Sicherheit.
Nachfolgend werden die einzelnen Ziele erläutert.

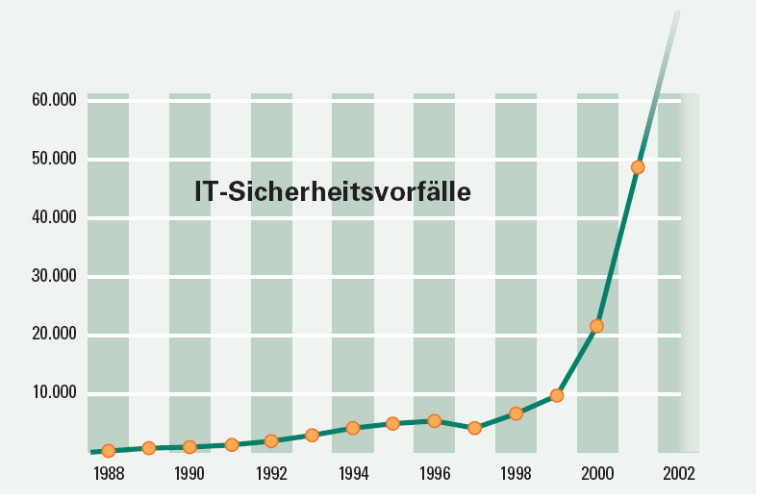


Abbildung 2.1: Bei www.cert.org gemeldete IT-Sicherheitsvorfälle [26]

2.1.1 Authentizität

Mit der Authentizität soll die Echtheit einer Information (z.B. eines Users) gewährleistet werden. Dies wird mit einer eindeutigen Identität und charakteristischen Eigenschaften überprüft - bei einem User ist dies häufig ein Benutzername und ein Passwort. Mittels der Authentifizierung muss für eine angenommene Identität nachgewiesen werden, dass diese mit den charakteristischen Eigenschaften der Identität übereinstimmen.
Mit der Authentizität kann eine aufgaben- oder benutzerabhängige Zugangsberechtigung gewährleistet werden und so den Anwendungsbetrieb vor Missbrauch schützen. Nachdem die Authentizität des Users festgestellt wurde, ist die Autorisation (siehe Abschnitt 2.2.2) wichtig.
Zur Wahrung der Sicherheit während der Authentisierung werden Verschlüsselungssysteme eingesetzt.

2.1.2 Datenintegrität

Die Datenintegrität umfasst die Menge der Maßnahem, die sicherstellen, dass die zu schützenden Daten nicht ohne Autorisierung manipuliert werden können. Dazu müssen für die Daten Nutzungsrechte festgelegt werden. Als Beispiele kann man hier das Rechtesystem unter UNIX/Linux ansehen oder dass ein Nutzer auf ein Konto nur bis zu einer gewissen Obergrenze über Geld verfügen kann. Die benötigten Verfahren und Mechanismen, die die Datenintegrität gewährleisten sollen, gehören zum Bereich der Zugriffskontrolle.
Die Datenintegrität spielt vor allem bei der Datenfernübertragung eine wichtige Rolle, da die Daten ein geschlossenes System verlassen und über eine leicht angreifbare Leitung geschickt werden.
Um sicherzustellen, dass die Datenintegrität tatsächlich gewährleistet wird, müssen Manipulationen erkannt werden. Gerade in verteilten Systemen kann eine Manipulation nicht immer von vornherein ausgeschlossen werden. Das heißt, dass die Korrektheit der Daten im Nachhinein geprüft werden muss. Zur Erkennung von Datenmanipulationen werden kryptographisch sichere Hashfunktionen (siehe Abschnitt 2.2.3) verwendet.

2.1.3 Informationsvertraulichkeit

Die Informationsvertraulichkeit ist sichergestellt, wenn die unautorisierte Informationsgewinnung nicht möglich ist. Informationen sollen nur Subjekten (beispielsweise einem User) zugänglich sein, denen die Berechtigung dazu erteilt wurde. Vor allen anderen Subjekten sollen die Daten geheim gehalten werden. Dies wird zum Beispiel mit Verschlüsselungssystemen erreicht.

2.1.4 Verbindlichkeit

Durch die Verbindlichkeit soll es unmöglich sein, dass ein Subjekt im Nachhinein abstreiten kann, eine gewisse Aktion durchgeführt zu haben. Diese Eigenschaft wird beispielsweise im elektronischen Handel immer wichtiger, damit die Zuordnung von Kunden zu Aktionen möglich ist. Dadurch soll eine Rechtsverbindlichkeit von Transaktionen herbeigeführt werden. Dies wird häufig durch digitale Signaturen gelöst. Die Erreichung der Verbindlichkeit erfordert eine Überwachung und Protokollierung der Aktivitäten von Subjekten.

2.1.5 Verfügbarkeit

Ein System gewährleistet die Verfügbarkeit, wenn eine unbefugte Vorenthaltung von Daten ausgeschlossen wird. Jedem authentifizierten und autorisierten Subjekt soll ein gewisses Maß an Zugänglichkeit und Verfügbarkeit der Daten zugesichert werden. Weiterhin sollen die Betriebsmittel - zum Beispiel Rechner - für das Subjekt verfügbar sein. Zur Sicherstellung der Verfügbarkeit sind Reglementierungen für die Nutzung von CPU-Zeit und Systemressourcen oder mehrere Übertragungskanäle denkbar. Diese Reglementierungen sollen bestimmen, welches Subjekt welche Ressource in welchem Umfang nutzen darf.
Um die Verfügbarkeit zu gewährleisten, müssen *Denial-of-Service (DoS)* Attacken verhindert werden. Dies ist aber nur sehr schwer - wenn überhaupt - möglich. Es gibt Programme, die sogenannten „Bomben“ (Anfragen, die auf einen DoS-Angriff zielen) erkennen und diese Verbindungen verweigern. Diese sind aber meist recht speziell und bieten keinen allgemeinen Schutz.

2.1.6 Anonymisierung und Pseudomisierung

Anonymisierung bedeutet, dass personenbezogene Daten in der Form verändert werden, dass ein Rückschluss von Einzelangaben über die Person auf die natürliche Person nicht gemacht werden kann. Zumindest sollte dies nicht mehr ohne einen unverhältnismäßig großen zusätzlichen Aufwand an Zeit, Kosten und Arbeitskraft möglich sein.
Neben der Anonymisierung gibt es noch - die etwas schwächere - Pseudomisierung. Hierbei werden Zuordnungsvorschriften zur Veränderung der personenbezogenen Angaben herangezogen. Der Rückschluss auf die natürliche Person kann nur gemacht werden, wenn diese Zuordnungsvorschrift bekannt und genutzt wird.

2.2 Sicherheitsmechanismen

Nachdem die Ziele der IT-Sicherheit erklärt sind, werden in diesem Abschnitt die allgemeinen Sicherheitsmechanismen aufgezeigt, die für die Erreichung der genannten Ziele benutzt werden können. Diese Mechanismen sind unabhängig von der Programmiersprache.
Die Authentisierung ist die Überprüfung der notwendigen Rechte des jeweiligen Users. Die Authentifizierung beschäftigt sich mit der Frage, ob der User auch wirklich derjenige ist, für

den er sich ausgibt. Mit Hilfe der kryptographischen Dienste werden Nachrichten verschlüsselt, damit das Abhören der Informationen erfolglos bleibt.
Im Folgenden werden - ohne Anspruch auf Vollständigkeit im Allgemeinen - die Mechanismen aufgeführt, die bei SicAri eine wichtige Rolle spielen.

2.2.1 Authentifizierung

Albrecht Beutelspacher ([3]) beschreibt die Benutzerauthentifikation als eine Grundaufgabe in der Sicherheitstechnik. Grundsätzlich kann die Identität einer Person über drei Wege nachgewiesen werden:

- **Wissen**: Zum Beispiel mit Passwort oder PIN
- **Merkmal**: Ein körperliches Merkmal wie Fingerabdruck oder Iris
- **Besitz**: Zum Beispiel über eine SmartCard oder einen Schlüssel

Das Merkmal ist in der täglichen Authentifizierung ein so häufig genutztes Charakteristikum, dass wir es meistens nicht bewusst wahrnehmen: Wir erkennen unsere Bekannten an ihrem Aussehen. Bei polizeilichen Ermittlungen werden Fingerabdrücke und teilweise sogar DNA-Analysen von Verdächtigen genommen. Mittlerweile wird diese Methode auch in gewissen Bereichen bei der Interaktion mit Rechnern eingeführt - Irisscanner und Reisepass mit biometrischen Angaben, die mit Computern überprüft werden können.
Die beiden anderen Methoden - „Authentifikation durch Wissen“ und „Authentifikation durch Besitz“ - kommen im täglichen Leben ebenfalls immer häufiger vor: Beispiele für „Authentifikation durch Wissen“ sind die Verwendung von Passwörtern beim Arbeiten mit dem Computer oder das Geldabheben am Geldautomaten. Viele Gebäude-Zugänge sind mittlerweile durch Transponder gesichert und die Person braucht eine spezielle Karte oder einen speziellen Schlüssel, um Zugang zum Gebäude zu erhalten.

2.2.2 Autorisation

Nachdem ein Subjekt authentifiziert wurde, wird das Subjekt autorisiert. Mit der Autorisation werden Zugriffsrechte im System durchgesetzt. Bevor solche Zugriffsrechte durchgesetzt werden können, müssen diese definiert sein. Eine Rolle kann zum Beispiel „Administrator“ oder „User“ sein. Je nachdem, welche Rolle das Subjekt einnimmt, hat es unterschiedliche Rechte.
So kann zum Beispiel ein Administrator auf Dienste zugreifen, auf die einem normaler User kein Zugriff gewährt wird.

2.2.3 Kryptographische Dienste

Kryptographie wird schon seit der Antike verwendet, um Nachrichten zu verschlüsseln und wieder zu entschlüsseln. Mit der Kryptographie soll der Zugriff von Dritten auf geheime Daten und Informationen verhindert werden.
Man unterscheidet zwei verschiedene Ansätze in der Kryptographie: die symmetrischen und die asymmetrischen Verfahren ([10]). Diese beiden Ansätze werden in den folgenden Abschnitten erklärt.

2.2.3.1 Symmetrische Verfahren

Bei den symmetrischen Verfahren ist der Schlüssel für Ver- und Entschlüsselung identisch (Abbildung 2.2) - oder die beiden Schlüssel sind zumindest von einander ableitbar. Da die Ver- und Entschlüsselung relativ schnell ist, werden symmetrischen Verfahren vor allem in den Bereichen eingesetzt, in denen Leistungsfähigkeit eine Rolle spielt.

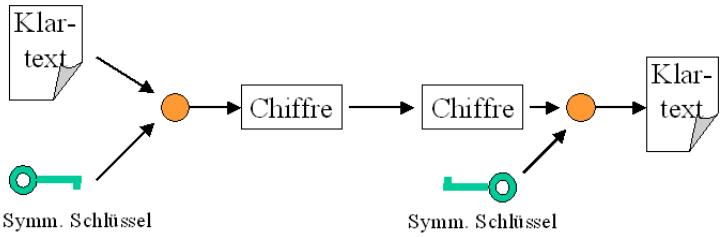


Abbildung 2.2: Symmetrische Verschlüsselung [15]

Um eine geheime Kommunikation zu ermöglichen, benötigen zwei Kommunikationspartner einen gemeinsamen, geheimen Schlüssel. Bevor die sichere Kommunikation stattfinden kann, müssen sich die Kommunikationspartner über den Schlüssel verständigen.
Bekannte symmetrische Verschlüsselungsverfahren sind der Data Encryption Standard (DES), Triple DES (3DES) und Advanced Encryption Standard (AES). Auf die verschiedenen Verfahren wird hier nicht näher eingegangen. Mehr Informationen gibt es in [10], [3] und [2].

2.2.3.2 Asymmetrische Verfahren

Bei asymmetrischen Verfahren hat jeder Kommunikationspartner ein Schlüsselpaar - einen privaten Schlüssel und einen öffentlichen Schlüssel, der anderen Kommunikationspartnern bekanntgegeben werden muss. Der große Unterschied zu den symmetrischen Verfahren besteht darin, dass der geheime (private) Schlüssel nicht ausgetauscht werden muss.
Ein Klartext wird mit dem öffentlichen Schlüssel des Empfängers verschlüsselt und der Empfänger entschlüsselt die Nachricht mit seinem privaten, geheimen Schlüssel (Abbildung 2.3). Damit Dritte nicht anstelle des Empfängers den eigenen öffentlichen Schlüssel veröffentlichen und damit die geheime Nachricht doch entschlüsseln können, muss die Authentizität des Schlüssels gewährleistet sein. Hierfür werden häufig Zertifikate eingesetzt.

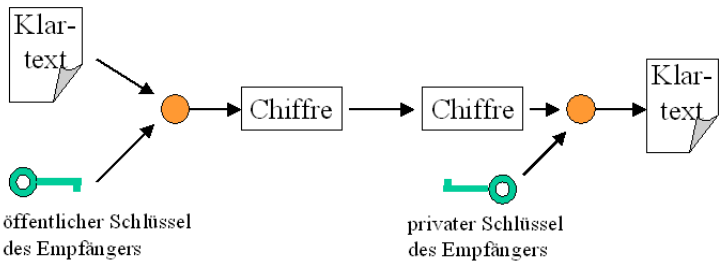


Abbildung 2.3: asymmetrische Verschlüsselung

Das bekannteste asymmetrische Verfahren ist das RSA-Verfahren ([10]).

In diesem Kapitel wurden die Grundlagen zur IT-Sicherheit gelegt. Welche Möglichkeiten Java bietet, um diese Ziele der IT-Sicherheit zu erreichen und welche Mechanismen von Java für die Sicherheit umgesetzt werden, wird im nächsten Kapitel erläutert.

Kapitel 3

Sicherheit in Java-basierten Systemen

> „Java is intended to be used in networked/distributed environments. Towards that end, a lot of emphasis has been placed on security. Java enables the construction of virus-free, tamper-free systems.“ ([5])

Seit der ersten Version von Java sind gewisse Sicherheitsmechanismen eingebaut. Diese haben sich jedoch von einem einfachen *sandbox*-Modell bis zu einem ausgereiften Sicherheits*framework* heute entwickelt. In diesem Kapitel wird erst die historische Entwicklung vorgestellt und dann der aktuelle Stand der in Java eingebundenen Sicherheitsmechanismen gezeigt.
Nicht nur in der zeitlichen Abfolge gibt es Unterschiede in den implementierten Sicherheitsmechanismen - auch innerhalb der verschiedenen Editionen der Java Plattform gibt es Unterschiede. Diese Unterschiede von *Micro Edition* und *Enterprise Edition* zur *Standard Edition* werden im Abschnitt 3.3 dargestellt.

3.1 Historie

Im Laufe der Zeit haben sich die Sicherheitsmechanismen in Java immer weiterentwickelt ([19], [12]). Bei den anfänglichen Konzepten merkte man schnell, dass diese nicht ausreichend und fehlerbehaftet waren. Aus diesem Grund wurden die Mechanismen immer weiter den neuen Bedürfnissen und Erkenntnissen angepasst. In den drei folgenden Abschnitten wird die Entwicklung von Java 1.0 bis zu Java 2 kurz dargestellt.
In den folgenden Abschnitten ist mehrfach von *Applets* und Anwendungen die Rede. Ein Applet wird als Code definiert, der nicht auf dem lokalen System gespeichert ist, sondern von einem anderen Rechner heruntergeladen werden muss. Eine Anwendung dagegen befindet sich auf dem lokalen System.

3.1.1 Sicherheit in Java 1.0

In der Version 1.0 von Java gab es nur zwei Stufen der Vertraulichlichkeit für Code. Applets wurden innerhalb einer *sandbox* ausgeführt und die Rechte der Applets waren eingeschränkt. So hatten Applets keinen Zugriff auf das lokale Dateisystem; es durften keine Dateien gelesen oder verändert werden, die sich auf dem lokalen Dateisystem befinden. Andere Anwendungen hingegen hatten völlige Freiheit - voller Zugriff auf das Dateisystem. Dies bedeutete, dass der Programmierer in dem Programm selbst eine Überprüfung von Rechten vornehmen musste.

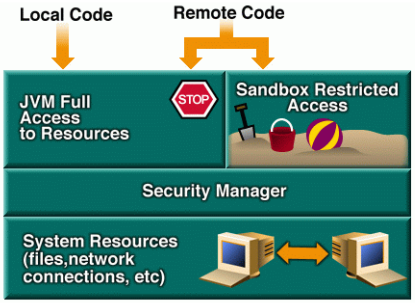


Abbildung 3.1: Sicherheitsmodell von Java 1.0 [19]

Die *sandbox* war wichtig, weil das Applet häufig ohne Kenntnis des Anwenders heruntergeladen wurde und weil weder das Applet noch der Autor bekannt war - jedenfalls in sehr vielen Fällen. Man darf einem Applet also nicht blind vertrauen.
In Java 1.0 gab es aber auch noch andere Mechanismen, die die Sicherheit gewährleisten sollen. Als erstes wäre die Sprache „Java“ an sich zu nennen mit der Typ-Sicherheit. Der *Bytecode-Verifier*, der *class loader* und die virtuelle Maschine sind weitere Mechanismen. Diese Mechanismen werden im Abschnitt 3.2 erläutert.
Die Abbildung 3.1 zeigt das Prinzip, das hinter dem Sicherheitsmodell von Java 1.0 steht. Zum einen gibt es den *local code*, also eine Anwendung, die auf dem lokalen System gespeichert ist. Diese bekommt von der virtuellen Maschine vollen Zugriff auf Ressourcen. Ein Applet läuft in der *sandbox*, die nur begrenzte Zugriffsrechte hat. Der *security manager* überprüft bei jedem Versuch, auf System-Ressourcen zuzugreifen, ob der aktuelle *Thread* auch die benötigten Zugriffsrechte hat. Der *security manager* ist daher der Teil, der die *sandbox* realisiert - jedem Applet werden die Zugriffsrechte auf System-Ressourcen verwehrt. Das bleibt auch in den nachfolgenden Versionen von Java erhalten.
Ein *Thread* stellt eine nebenläufige Ausführungseinheit innerhalb genau eines Prozesses dar.

3.1.2 Sicherheit in Java 1.1

Die Entwickler bei Sun haben erkannt, dass die Trennung von Applets und Anwendungen wenig praxisnah war. Es wurden auch Applets geschrieben, die mehr Rechte haben mussten als bisher. Mit der Version 1.1 kam eine wichtige Erweiterung: Der *trusted code* wurde eingeführt. Damit konnte über eine Signatur das Applet als vertrauenswürdig eingestuft werden, womit das Applet die gleichen Rechte bekommen hat wie eine normale Anwendung. Abbildung 3.2 zeigt das Sicherheitsmodell von Java 1.1 und dass der *trusted code* vollen Zugriff auf die Ressourcen bekommt.
Der lokale Code bekommt automatisch vollen Zugriff auf die System-Ressourcen. Bei Applets hat sich jedoch etwas geändert: Unsignierte Applets werden weiterhin in der *sandbox* ausgeführt. Ist das Applet jedoch signiert und wird der Signatur vertraut, dann wird das Applet mit den gleichen Rechten ausgeführt wie eine lokale Anwendung. Das Konzept des *security manager*, der die Rechte bei einem Zugriff auf System-Ressourcen überprüft, ist immer noch enthalten.
Weiterhin wurde mit Java 1.1 die *Java Cryptography Architecture (JCA)* eingeführt. Die JCA ist eine Architektur für kryptographische Funktionen in der Java Plattform. Diese Architektur ist in Abschnitt 3.2.2.1 eingehender beschrieben.

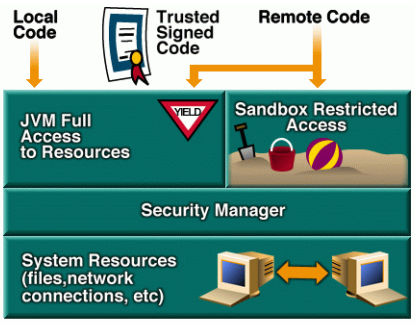


Abbildung 3.2: Sicherheitsmodell Java 1.1 [19]

3.1.3 Sicherheit in Java 2

Mit der Version 1.2 von Java - auch Java 2 genannt wurden weitere Sicherheitsmechanismen eingebaut. Hierbei sind vor allem die `Policy`- und `Permission`-Klassen zu nennen. Das heißt, dass jeder Code - egal ob Applet oder Java Anwendung - einer Sicherheitsrichtlinie unterliegt. Dabei kann ganz spezifisch festgelegt werden, auf welche Ressource (z.B. eine Datei) der Code mit welchen Rechten (z.B. Lese- bzw. Schreibrechte) Zugriff hat.
Weiterhin kamen *protection domains* hinzu, die in Abschnitt 3.2.1.5 genauer erläutert werden. Jede Domäne wird wie eine eigene *sandbox* behandelt, so dass Applets weiterhin in einer restriktiven Umgebung laufen und Java Anwendungen alle Rechte haben. Die Abbildung 3.3 zeigt die unterschiedlichen Möglichkeiten in Java 2:
Der Pfeil ganz links gilt für Code, der ohne Einschränkungen läuft, während der Pfeil ganz rechts zu Code gehört, der in der ursprünglichen *sandbox* läuft. Alle Pfeile zwischendrin zeigen auf spezifische *protection domains*, die eingeschränkte Rechte haben aber mehr Rechte als in der *sandbox*.

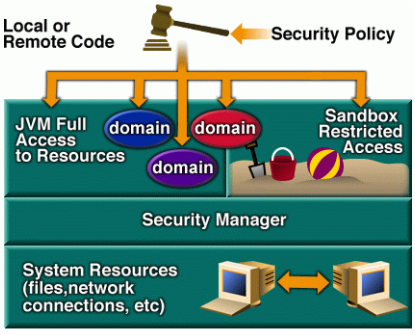


Abbildung 3.3: Sicherheitsmodell Java 2 [19]

3.2 Aktueller Stand

Über die Sicherheitsmechanismen, die in der aktuellen Java-Version eingebaut sind, gibt ein Artikel von Sun [32] Auskunft. Die einzelnen Bereiche sind zum Teil in [19] und [12] weiterge-

hend erklärt. Hier werden die einzelnen Sicherheitsmechanismen vorgestellt und kurz erläutert. Die hier gezeigten Sicherheitsmechanismen sind seit der Java-Version 1.4 implementiert und wurden in der Version 1.5 nur zum Teil erweitert.
Java stellt ein Bündel an Schnittstellen zur Verfügung, die die großen Sicherheitsbereiche abdeckt: Kryptographie, *Public-Key-Infrastructure*, Authentisierung, sichere Kommunikation und Zugriffskontrolle. Diese Schnittstellen ermöglichen eine schnelle und einfache Integration der Java-Sicherheitsmechanismen in Java Anwendungen. Diese Mechanismen werden in den kommenden Abschnitten erläutert.
Viele der hier erläuterten Mechanismen stellen nur Möglichkeiten dar, die Java bietet. Der Programmierer selbst hat jedoch noch viel zu tun, damit er Sicherheitslücken stopft. Standardmäßig erfolgen Abfragen auf Zugriffsrechte über den *access controller* und für die Verwendung des*security manager* (siehe Abschnitt 3.2.1.3) muss dieser beim Start der virtuellen Maschine aktiviert werden.
Im ersten Abschnitt werden die Mechanismen vorgestellt, die zur Standard-Sicherheitsarchitektur von Java gehören. Im zweiten Abschnitt werden die Mechanismen erläutert, die zur erweiterten Sicherheitsarchitektur gehören. Die im zweiten Abschnitt beschriebenen Mechanismen werden als erweiterte Sicherheitsarchitektur bezeichnet, weil sie über die Sicherheitsbedürfnisse von einfachen Anwendungen hinausgehen. Diese Mechanismen sind auch nicht in den Java-Kern integriert, sondern müssen vom Programmierer implementiert werden.

3.2.1 Standard-Sicherheitsarchitektur

Die Standard-Sicherheitsarchitektur muss nicht explizit verwendet werden, sondern wird automatisch angewendet. Dabei handelt es sich um eingebaute Sicherheitsmechanismen, die bei jedem Java-Programm zum Tragen kommen. Das geht von der Sprache an sich bis hin zu den *class loadern*. Diese Mechanismen werden in den kommenden Abschnitten erklärt.

3.2.1.1 Java-Sicherheit und Bytecode-Verifizierung

Die Programmiersprache Java hat einige Sicherheitsmechanismen nativ implementiert: Die virtuelle Maschine verwaltet Speicher, ist für die automatische *Garbage Collection* verantwortlich und überprüft die Grenzen von Arrays. Damit werden Pufferüberläufe vermieden. Java kennt weiterhin verschiedene Schlüsselwörter, die bestimmte Zugriffsrechte angeben: `private`, `protected` und `public`. Der restriktivste Zugriff erfolgt, wenn das Schlüsselwort `private` genannt wird.
Der Java-Kompiler übersetzt den Code in eine maschinenunabhängige *Bytecode*-Darstellung. Vor dem Start einer Java-Applikation wird dieser Bytecode überprüft, so dass nur legitimer Bytecode ausgeführt wird. Der Bytecode muss den Java-Spezifikationen genügen und darf die Sprachregeln nicht verletzen. Weiterhin wird überprüft, ob die Grenzen des *Stacks* eingehalten werden und ob es ungültige Typkonvertierungen gibt. Erst wenn der Bytecode verifiziert wurde, wird das Programm von der Java-Runtime zur Ausführung vorbereitet.

3.2.1.2 Class Loader

Der *class loader* ist in der *Java Virtual Machine (JVM)* dafür zuständig, die Java-Klassen zu laden und ist damit ein wichtiges Glied in der Sicherheitskette von Java. Der *class loader* arbeitet mit dem *security manager* und der Zugriffskontrolle zusammen, um die Sicherheitsbestimmungen durchzusetzen. Klassen, die in Java von der gleichen *class loader*-Instanz geladen

werden, gehören in den gleichen Namensraum (*namespace*). Die Kompatibilität zweier Klassen-Instanzen während der Laufzeit wird also durch die Identität des *class loader* und des kompletten Klassennamen inklusive *package* bestimmt.

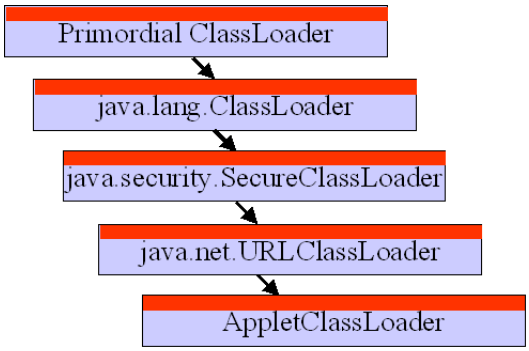


Abbildung 3.4: Vererbungshierarchie der Class Loader

Klassen können dynamisch nachgeladen werden, was eine Flexibilisierung von Anwendungen bedeutet, da Software-Komponenten während der Laufzeit nachinstalliert werden können. Ein Charakteristikum ist, dass Klassen nur nachgeladen werden, wenn sie benötigt werden - das sogenannte *lazy loading*. Das bedeutet, dass die Typsicherheit dann gesichert wird, wenn ein Link auf die Klasse gesetzt wird. Diese Überprüfung findet nur einmal statt, so dass weitere Überprüfungen während der Laufzeit vermieden werden.
Es können eigene *class loader*-Klassen geschrieben werden, wodurch der Programmierer bestimmen kann, von welchen Speicherorten Klassen geladen werden sollen und es können passende Sicherheitsattribute an den *class loader* vergeben werden. Die Vererbungshierarchie von *class loader*n ist in Abbildung 3.4 dargestellt. Der *Primordial Class Loader* ist in der JVM integriert und ist meistens in C geschrieben. Dieser *class loader* existiert in der virtuellen Maschine nur einmal. Die Klassen, die durch diesen *class loader* geladen werden, werden als vertrauenswürdig eingestuft und nicht mehr verifiziert.
Die anderen *class loader* in der Vererbungshierarchie sind in Java implementiert und können mehrfach in der JVM existieren. `java.lang.ClassLoader` ist eine abstrakte Klasse und alle, in der Vererbungshierarchie darunterliegenden *class loader* implementieren diese abstrakte Klasse. Eine besondere Bedeutung kommt dem `SecureClassLoader` zu, da er bei jedem Ladevorgang eine *CodeSource* bildet (aus *CodeBase* und Unterzeichner des Codes) und diese mit den *Permissions* aus der Sicherheitsrichtline verbindet.
Es existiert eine Hierarchie bei den *class loaders*, da in einer virtuellen Maschine mehrere *class loader* existieren können. Dadurch soll sichergestellt werden, dass immer die „richtige“ Klasse geladen wird. Der Ladeprozess für Klassen wird von einem *Primordial Class Loader* - oder *Bootstrap Class Loader* - angestoßen. Die Rangfolge der *class loader* ist in Abbildung 3.5 dargestellt. Daraus lässt sich ableiten, von welchem *class loader* einzelne Klassen geladen werden:

- ***Bootstrap Class Loader***- lädt die Java Basisklassen
- ***Extension Class Loader***- lädt die Klassen, die in dem Verzeichnis, das unter `sun.ext.dirs` definiert wurde, liegen
- ***System Class Loader***- lädt die Klassen, die im *Classpath* liegen

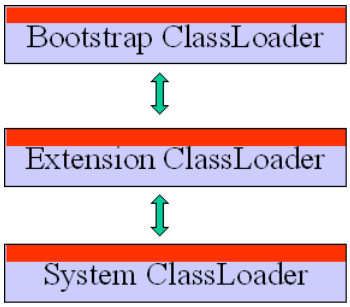


Abbildung 3.5: Rangfolge der ClassLoader

Wenn in der Anwendung eine Klasse geladen werden soll, wird die Suche nach der Klasse „nach oben“ delegiert bevor der *class loader* selbst nach der Klasse sucht. Erst wenn die übergeordneten *class loader* die Klasse nicht laden können, versucht es der untergeordnete *class loader* selbst. Wenn zum Beispiel eine vom Benutzer geschriebene Klasse `Test`, die im Klassenpfad liegt, geladen werden soll, wird die Anfrage zunächst vom System ClassLoader bis zum Bootstrap ClassLoader nach oben gereicht. Da der Bootstrap ClassLoader die Klasse aber nicht findet, wird die Anfrage an den Extension ClassLoader deligiert. Wenn auch der ExtensionClassLoader die Klasse nicht laden kann, so wird die Anfrage an den System ClassLoader gereicht, der die Klasse lädt.
Mit diesem Mechanismus soll verhindert werden, dass gewisse Klassen überschrieben werden können. So ist es zum Beispiel nicht möglich, `java.lang.String` durch eine eigene Klasse `java.lang.String` zu ersetzen. Das könnte sonst ein Angreifer ausnutzen um eine Klasse entsprechend seinen Vorstellungen zu implementieren. Damit könnte er jedes System, das die native Klasse verwendet, komprimittieren.
Wie der komplette Mechanismus in Java 1.2 funktioniert ist in [19] nachzulesen.

3.2.1.3 Security Manager und AccessController

Standardmäßig ist der Java-eigene *security manager* bei Applikationen aktiviert und kann im Programm durch ein `System.setSecurityManager()` oder durch die entsprechende Option durch einen eigenen *security manager*.
Diese Option ist `-Djava.lang.SecurityManager`.
Der *security manager* ist der wichtigste Teil bei der Zugriffskontrolle und wird bei jeder Entscheidung, ob ein Zugriff auf eine Ressource gewährt oder verweigert werden soll, aufgerufen. Dadurch ist es möglich, potentiell unsichere Operationen zu verweigern. Es besteht auch die Möglichkeit, das Programm mit einer Fehlermeldung abzubrechen. Der *security manager* implementiert diese Methoden jedoch nicht selbst, sondern ist die Schnittstelle zum *access controller*.
Der *security manager* ruft für jede Entscheidung die `checkPermission()`-Methode des *access controllers* auf. Dadurch ist der *access controller* die Instanz, die effektiv über die Gewährung von Zugriffsrechten entscheidet. In der Abbildung 3.6 wird gezeigt, dass in einem Programm die Methode `openFile()` implementiert ist, mit der eine Datei eingelesen werden soll. Dazu wird die Klasse `FileInputStream` aus dem `java.io`-Package verwendet. In Java werden durch die Verwendung von Systemklassen - zu denen `java.io.FileInputStream` gehört - nicht mehr Rechte zugewiesen. Daran wird das Prinzip der *protection Domain*s sichtbar.

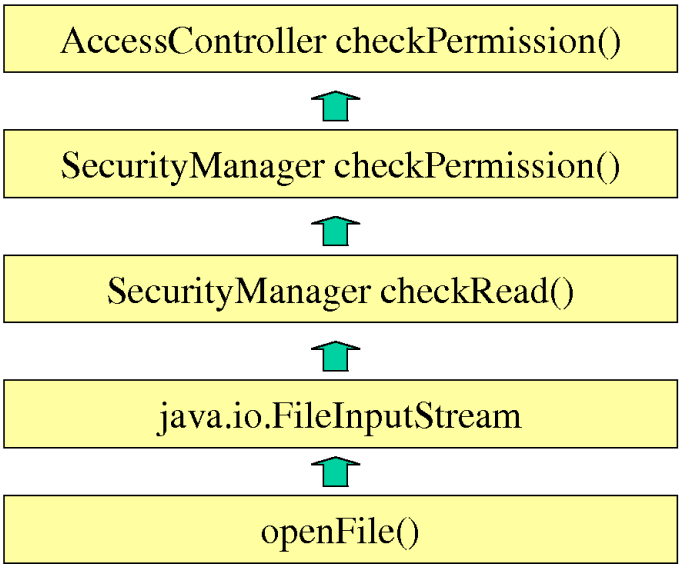


Abbildung 3.6: Entscheidung über Zugriffsrechte[1]

Im Szenario, das in der Abbildung gezeigt wird, ist der *security manager* aktiviert. Daher wird die Methode `checkRead()` des *security managers* aufgerufen, die wiederum die `checkPermission()`-Methode im *security manager* aufruft. Wie oben erwähnt, deligiert der *security manager* die Entscheidung über das Gewähren eines Zugriffsrechts an den *access controller* weiter.

3.2.1.4 Zugriffskontrolle (Permissions)

Die Architektur für Zugriffskontrollen in Java schützt den Zugriff auf Ressourcen wie z.B. lokal gespeicherte Dateien. Die gesamte Zugriffskontrolle wird über den Sicherheitsmanager abgehandelt. Der Sicherheitsmanager muss für die Kontrollen installiert sein; bei Applets und Java Web-Start Anwendungen ist dies standardmäßig installiert. Das Installieren des Sicherheitsmanager muss entweder direkt im Programm mit der Methode `setSecurityManager()` gemacht werden oder java muss mit der `-Djava.security.manager` Option gestartet werden.
In Java werden *permissions* und *policy* unterschieden. Die Sicherheitsrichtlinie wird in der sogenannten *policy* Datei definiert. Diese enthält entsprechende Zugriffsrechte (*permissions*) auf bestimmte Ressourcen.

Policy

Die *policy*-Datei enthält eine Reihe von Befehlen, die eine Zuordnung von Rechten mit Code darstellen. Ein einfaches Beispiel wäre

```
grant{
  permission java.io.FilePermission “<<ALL FILES>>“,“read“;
}
```

Dadurch wird ein Lesezugriff auf alle Dateien erlaubt.

Code	Zugriff
Renee Applets, signiert	lesend/schreibend auf /tmp
Renee Applets, signiert und unsigniert	Verbindung zu fraunhofer.de
lokale Anwendungen	lesend/schreibend auf /data
...	...

Tabelle 3.1: Policy Matrix [12]

Die *policy* kann also als eine Matrix verstanden werden, in der festgelegt wird, auf welche Ressourcen in welcher Art (z.B. lesend oder schreibend) unter welchen Umständen zugegriffen werden darf. Die Tabelle 3.1 soll dies verdeutlichen.
Ein kleines Set an Zugriffsrechten wird dem Code schon von den *class loadern* gewährt. Zu jedem Zeitpunkt kann es in der Laufzeitumgebung nur eine installierte *policy* geben.

Permissions
*Permissions*regeln den Zugriff auf Ressourcen.

3.2.1.5 Protection Domains

Bei der Festlegung von Zugriffsrechten mittels *policy* und *permission* ist die Erweiterbarkeit sehr eingeschränkt. Um die Erweiterbarkeit zu ermöglichen, wurde mit dem *Java Development Kit (JDK)* 1.2 die *protection domain* eingeführt. Nach dem klassischen Verständnis ist eine Domäne durch eine Gruppe von Objekten begrenzt, die für eine Instanz zugänglich sind. Dieser Instanz wurden vorher Rechte gewährt, die dann auf die Objekte übertragen wurden.
Seit der JDK 1.2 definiert man *protection domains*, denen man dann Rechte gewährt. Klassen und Objekte gehören dabei zu einer *protection domain* und bekommen automatisch die Zugriffsrechte, die die *protection domain* hat, zugewiesen. Die Zuordnung von Klassen und Objekten zu einer *protection domain* erfolgt einmal vor der Verwendung einer Klasse und bleibt dann während der Lebensdauer eines Klassenobjekts bestehen. Wenn zwei unterschiedliche *class loader* eine Klasse laden, enthält jede Domäne eine eigene Kopie der Klasse [16]. Wie die Zuordnung gemacht wird, ist in Abbildung 3.7 zu sehen.
Dass *permissions* nicht mehr direkt an Klassen gebunden werden, hat den Vorteil, dass man die Rechte abhängig von Authentizität und Autorisation eines Subjekts vergeben kann.

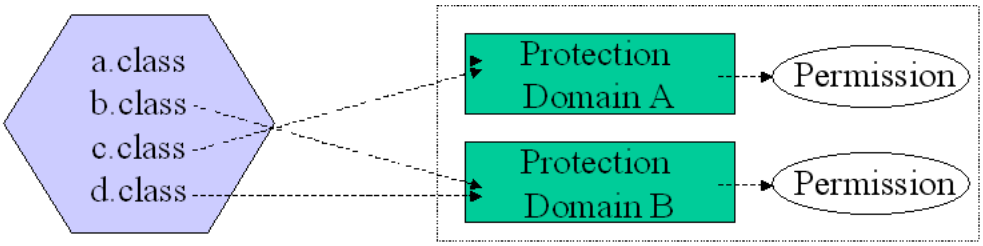


Abbildung 3.7: Zuordnung von Klassen zu ihren Protection Domains [16]

3.2.1.6 Signierte Applets

Jeder „fremde“ Code wurde als unsicher eingestuft. Dieses Raster war aber zu grob, so dass die signierten Applets [7] in Java 1.1 eingeführt wurden. Einige Applets funktionierten mit den strengen Restriktionen in der *sandbox* nicht. Als Beispiel kann man ein Unternehmen nennen, bei dem Mitarbeiter firmeneigenen Code herunterladen mussten. Wenn ein korrekt signiertes Applet heruntergeladen wird, so erhält dieses Applet die gleichen Rechte wie jede Java Anwendung. Die Signatur und der Code werden dabei in ein *Java Archiv (JAR)* gespeichert.

3.2.2 Erweiterte Sicherheitsarchitektur

Neben der Standard-Sicherheitsarchitektur stellt Java noch eine erweiterte Sicherheitsarchitektur zur Verfügung. Diese umfasst Standardschnittstellen, die Plattformunabhängigkeit gewährleisten und die Interoperabilität zwischen verschiedenen Hersteller-Implementationen sicherstellen. Abbildung 3.8 zeigt die erweiterte Sicherheitsarchitektur: Es besteht die Möglichkeit, dass die Anwendungen eines der vier Pakete verwendet, die wiederum die Kryptographie Pakete benötigen. Die Anwendung kann auch direkt auf die Kryptographie Pakete zugreifen. Die Provider stellen ein Set von Kryptographischen Diensten bereit.
Die Schnittstellen der erweiterten Sicherheitsarchitektur sind im *Java Development Kit (JDK)* seit der Version 1.4 enthalten. Die Architektur ist ausführlich in [7] beschrieben.
Diese Schnittstellen können für eine *Public-Key-Infrastructure (PKI)* verwendet werden. Eine PKI ist eine Kombination von Software, Verschlüsselungstechnologie und Diensten, die sichere Kommunikation und sachlichen Transaktionen über das Internet ermöglichen [31].

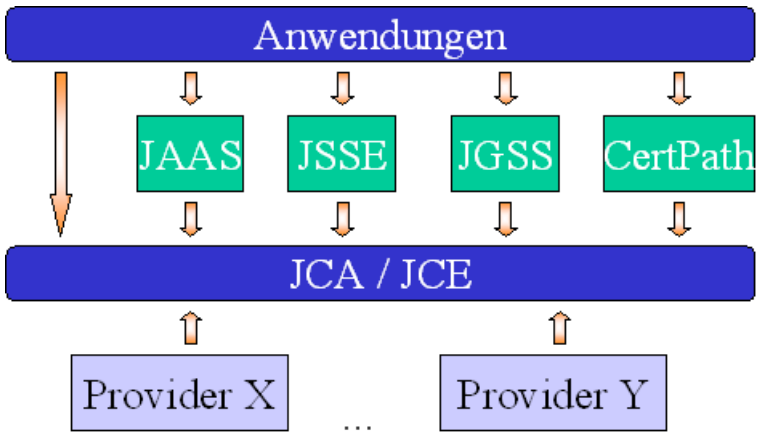


Abbildung 3.8: Die erweiterte Sicherheitsarchitektur von Java [7]

3.2.2.1 Kryptographie

Die *Java Cryptography Architecture (JCA)*, deren erste Version mit Java 1.1 ausgeliefert wurde, bietet ein großes Gerüst an kryptographischen Diensten, angefangen von *message digest* Algorithmen über Signatur-Algorithmen bis zu Zufallszahlengeneratoren und sogenannten *Message Authentication Codes (MAC)*. Die JCA besteht aus zwei Bereichen: den Kryptographie-Klassen des JDK und den sogenannten *Java Cryptography Extensions (JCE)*.
Das Design dieser Architektur folgt im Wesentlichen zwei Grundsätzen:

- Unabhängigkeit und Erweiterbarkeit des Algorithmus
- Unabhängigkeit und Interoperabilität der Implementierung

Damit wird sichergestellt, dass ein Zertifikat des einen *providers* auch von einem anderen *provider* verifiziert werden kann.

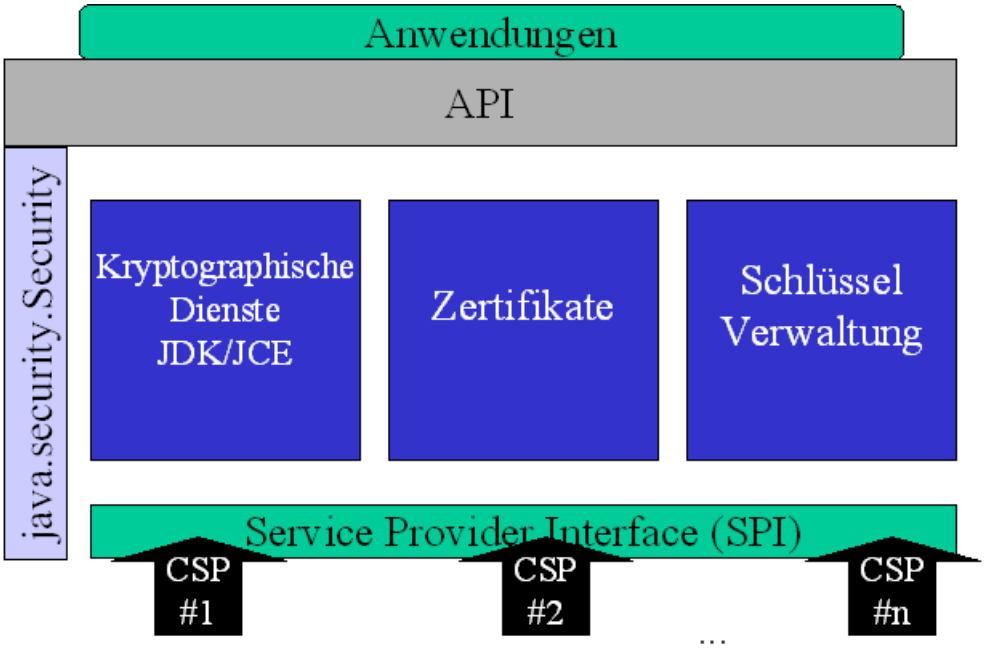


Abbildung 3.9: Die Module der Java Cryptography Architecture [19]

Ein *Cryptographic Service Provider (CSP)* ist ein Set von Modulen, die kryptographische Dienste implementieren.

3.2.2.2 Certificate Path

Die *Java CertPath API* stellt eine Schnittstelle zur Überprüfung von Zertifikatsketten - auch *certificate chains* genannt - zur Verfügung. Digitale Zertifikate spielen eine wichtige Rolle in Geschäftsanwendungen und anderen Transaktionen. Die Zertifikate werden von einer sogenannten *Certification Authority (CA)* ausgestellt und enthalten Angaben über den Besitzer des Zertifikats, eine Seriennummer, die Gültigkeit und andere Angaben. Zu den wichtigen Daten gehören der öffentliche Schlüssel und die Signatur, die zur Validierung des Zertifikats benötigt werden. Da auch die CAs Zertifikate benötigen, die die Echtheit der von der CA ausgestellten Zertifikate bescheinigen, entsteht eine Kette von Zertifikaten wie in Abbildung 3.10 zu sehen ist.
Die *Java Certification Path API (CertPath)* ist besonders für die *Public Key Infrastructure* wichtig. In der *Standard Edition von Java (J2SE)* bietet CertPath Methoden zum Parsen und Verwalten von Zertifikaten, auch das Erstellen von Zertifikaten ist möglich. Es werden die *Provider Interfaces* der *Java Cryptography Architecture (JCA)* benutzt, so dass CertPath in jede J2SE-Umgebung integriert werden kann.

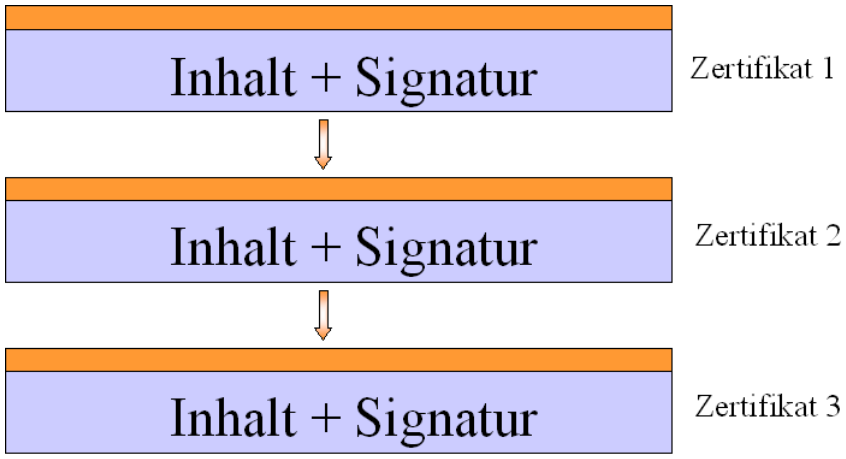


Abbildung 3.10: Kette von Zertifikaten [12]

Public Key Infrastructure (PKI)

Public Key Infrastructure beschreibt ein Gerüst, das den sicheren Austausch auf Basis der *Public Key* Kryptographie erlaubt. Mit PKI werden alle Komponenten zusammengefasst, die im Bereich der asymmetrischen Krytosysteme benötigt werden ([10]).

3.2.2.3 Authentisierung und Autorisierung

Mit den Zugriffsrechten (*permissions*) wird aufgrund einer Positivliste das Recht einer Anwendung definiert. Damit wird berücksichtigt, woher der Code stammt. Mit Hilfe der *Java Authentication and Authorization Services (JAAS)* [18] sollen die Rechte des Benutzers in den Vordergrund treten. Li Gong spricht statt von Benutzern von Subjekten ([6]), da nicht nur menschliche Benutzer sondern auch Services und Anwendungen gemeint sein können. Diese Architektur ist ab der Version 1.4 Bestandteil des JDK und bietet einen standardisierten Mechanismus der Benutzer-Authentifizierung. Das JAAS-*Framework* basiert auf PAM, einem *Pluggable Authentication Module* [33].

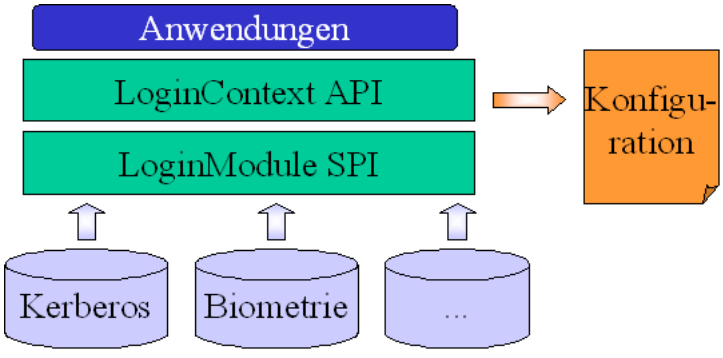


Abbildung 3.11: Login-Module als *Plug-Ins* des Authentisierungs-Frameworks [32]

Die Anwendungen agieren nur noch mit der `LoginContext` API (Abb. 3.11), während die verschiedenen Protokolle zur Authentifizierung wie Kerberos oder *Smart-Cards* einfach in das

`LoginModule` geladen werden können. Dadurch ist es möglich, neuere Authentifizierungsmechanismen zu implementieren ohne die gesamte Anwendung ändern zu müssen. Der `LoginContext` ruft die Login Module auf, die dann die Authentisierung durchführen.
Die Konfiguration dient dazu, die zu benutzenden Authentisierungsmechanismen oder Login Module festzulegen. Dies führt dazu, dass durch Änderung der Konfigurationsdatei ein neuer Mechanismus zur Authentisierung eingesetzt wird.
Die Interaktionen zwischen dem Subjekt und der Autorität werden durch die `Principal`-Klassen implementiert. Diese `Principal`s kennen nur ihren Namen und bieten einige überschriebene Methoden, wie ein Beispielcode in [14] zeigt.

3.2.2.4 Sichere Kommunikation

In Netzwerken – ob in einem kleinen Heimnetzwerk oder im Internet – besteht immer die Gefahr, dass die Verbindung „abgehört“ wird und Daten ausspioniert werden. Aus diesem Grund ist die sichere Kommunikation ein wichtiger Punkt in der Sicherheit. Denn was bringt der sicherste Computer, wenn der Weg der Daten absolut unsicher ist?
Über das Netzwerk werden nicht nur Daten ausgetauscht, die „unwichtig“ sind, sondern auch Passwörter und andere persönliche Informationen. Diese sollten nach Möglichkeit keinen unerwünschten Empfänger erreichen. Weiterhin ist es wichtig, dass die Daten, die beim erwünschten Empfänger eintreffen, nicht während des Datenaustauschs verändert worden sind.
Für die sichere Kommunikation wurden mehrere Protokolle entwickelt. Diese Protokolle kann man in zwei Gruppen aufteilen [10]:

- Kommunikationsprotokolle
- Anwendungsprotokolle

Die Kommunikationsprotokolle werden dann eingesetzt, wenn zur Sicherung der Kommunikation die eingesetzte Technik, z.B. Schlüssel für die Verschlüsselung, übermittelt werden. Sender und Empfänger müssen also erst interagieren und sich über die zu verwendenden Verfahren einigen.
Anwendungsprotokolle sind dagegen dadurch charakterisiert, dass einer der Kommunikationspartner alle Informationen, die für die sichere Kommunikation benötigt werden - wie zum Beispiel die verwendeten Schlüssel - mit der Nachricht kombiniert werden. Der Empfänger kann diese Informationen überprüfen und verarbeiten.

Als Basis für die Sicherung der Kommunikation dient die Kryptographie. Die Java Plattform stellt eine Reihe von Standard-Protokollen zur sicheren Kommunikation zur Verfügung. Diese werden in den nächsten Unterabschnitten kurz angesprochen.

SSL/TLS

Sowohl das *Secure Socket Layer (SSL)* Protokoll [1] als auch das *Transport Layer Security (TLS)* Protokoll [9] werden in der Transportebene des OSI-Referenzmodells eingesetzt.
SSL wurde 1994 von Netscape entwickelt und arbeitet transparent, so dass es nicht nur für HTTP-Verbindungen geeignet ist sondern auch für Protokolle wie SMTP, Telnet oder FTP. SSL codiert mit *Public Keys*, die von einer dritten Partei nach dem X.509-Standard [28] bestätigt werden. Mehr Informationen über SSL, z.B. über die Protokolle wie *Handshake*-Protokoll, sind in [10] und [29] zu finden.

TLS ist die Weiterentwicklung von SSL3.0 und hat sich mittlerweile als Standard etabliert. TLS verwendet ein anderes Verfahren zur Berechnung des *Message Authentication Codes (MAC)* und auch für die Schlüsselerzeugung wird ein anderes Verfahren angewendet.
Für beide Protokolle bietet Java eine Schnittstelle, die die Funktionalitäten wie Datenverschlüsselung, Nachrichtenintegrität und Server-Authentisierung implementiert. Diese Schnittstelle ist die *Java Secure Socket Extension (JSSE).*

GSS-API

GSS-API ist eine Schnittstelle, die Anwendungen ermöglicht, auf Sicherheits-Dienste zugreifen zu können. Für Java gibt es die *Java Generic Secure Services (JGSS)*, die eine Implementierung in Java sind. Dies wird in Abschnitt 3.2.2.5 behandelt.

SASL

Der *Simple Authentication and Security Layer (SASL)* [23] definiert ein Protokoll zur Authentisierung und Aufbau einer sicheren Schicht zwischen *Client* und *Server*. Es definiert nur das „Wie“ der Authentisierung aber nicht den Inhalt der Daten. SASL wird von Protokollen von wie dem *Lightweight Directory Access Protocol 3 (LDAP v3)* oder dem *Internet Message Access Protocol (IMAP)* verwendet.
Dem Entwickler wird durch die Schnittstellen geholfen, einen eigenen SASL-Mechanismus zu implementieren. Diese Mechanismen werden durch das Benutzen der JCA installiert.
SASL arbeitet mit anderen Schnittstellen wie JSSE und Java GSS zusammen. So kann der SASL-Mechanismus auf dem GSS-API-Mechanismus aufgesetzt werden, der von LDAP verwendet wird.

3.2.2.5 Java Generic Secure Services

Die *Java Generic Security Services (JGSS)* API ist die Java-Implementierung der *Generic Security Services API (GSS-API).* Die GSS-API ist eine einheitliche API für eine gegenseitige Authentifikation von *Client* und *Server* und eine API für den sicheren Austausch von Nachrichten - unabhängig von der darunterliegenden Technologie.
JGSS und JSSE stellen ähnliche Sicherheitsmechanismen zur Verfügung, wobei die verwendete Technologie unterschiedlich ist. So verwendet JGSS den Kerberos-Algorithmus anstatt SSL/TLS.

3.3 Sicherheitsmodelle der Java-Plattformen

Im vorangegangenen Abschnitt wurden die in Java implementierten Sicherheitsmechanismen kurz vorgestellt, die das Sicherheitsmodell in der Java 2 Standard Edition darstellen. Java gibt es in drei unterschiedlichen Entwicklungsplattformen. Applikationen für mobile Endgeräte werden mit der *Java 2 Micro Edition (J2ME)* entwickelt; wobei es hier Einschränkungen der Sicherheitsmechanismen gibt. Für Business-Anwendungen gibt es die *Java 2 Enterprise Edition (J2EE)*, die ein weiterreichenderes Sicherheitskonzept bietet als die oben genannten Sicherheitsmechanismen.
Der Schwerpunkt bei dieser Arbeit liegt jedoch bei der *Java 2 Standard Edition (J2SE)* mit der auch SicAri entwickelt wird. Aus diesem Grund wurden die J2SE Mechanismen im Abschnitt 3.2 vorgestellt. In den folgenden zwei Abschnitten werden die Sicherheitsmodelle der zwei Java-Plattformen vorgestellt, die nicht im Schwerpunkt behandelt werden. Als erstes wird die

Micro Edition (J2ME) behandelt, da sie das kleinste Sicherheitskonzept hat, danach kommt die *Enterprise Edition (J2EE)* mit dem breitesten Sicherheitsspektrum.

3.3.1 J2ME

Die Micro Edition der Java 2 Plattform ist hauptsächlich für mobile Endgeräte – wie Java-fähige Mobiltelefone oder *Personal Digital Assistants (PDA)* – gedacht. Diese Geräte haben weniger Ressourcen (Speicher, CPU) zur Verfügung als Desktop PCs und Server. Aus diesem Grund ist die oben dargestellte Sicherheitsarchitektur auf diesen Geräten nicht möglich.
Die J2ME *Connected Limited Device Configuration* (CLDC)-Architektur definiert 3 Schichten oberhalb des Betriebsystems des Gerätes (Abbildung 3.12):

- Profile Layer
- Configuration Layer
- Java Virutal Machine

Profile Layer

Der *Profile Layer* definiert auf Grund von einer bestimmten Konfiguration – die durch den *Configuration Layer* bestimmt wird – einen Satz von Schnittstellen (das sogenannte Profil). Anwendungen, die ein bestimmtes Profil unterstützen müssen auf allen Geräten laufen, die dieses Profil unterstützen. Geräte können mehrere Profile unterstützen.

Configuration Layer

Der *Configuration Layer* beinhaltet die Basisklassen, die für eine Anwendung auf einem bestimmten Endgerät vorhanden sein müssen.

Java Virutal Machine

Die *Java Virtual Machine* ist für J2ME wesentlich kleiner als im J2SE-Bereich und wird aufgrund des niedrigen Ressourcenverbrauchs auch *Kilo Virtual Machine (KVM)* genannt.

Auf der CLDC-Architektur ist das *Mobile Information Device Profile (MIDP)* aufgesetzt, das zur Erstellung von mobilen und vernetzten Java Anwendungen für mobile Endgeräte eingesetzt wird. Dafür stellt MIDP die entsprechenden Bibliotheken zur Verfügung - wie die `javax.microedition.io` für die Ein- und Ausgabe von Daten.
Die Entwickler von Anwendungen mit J2ME haben mit der CLDC die Möglichkeit, Rechte für die einzelnen Anwendungen zu definieren. Das Sicherheitsmodell von CLDC kennt – wie in [22] beschrieben – drei unterschiedliche Sicherheitsebenen:

- *Low Level Security*
- *Application Level Security*
- *End-to-End Security*

Die drei Sicherheitsebenen werden in den folgenden Abschnitten erläutert.

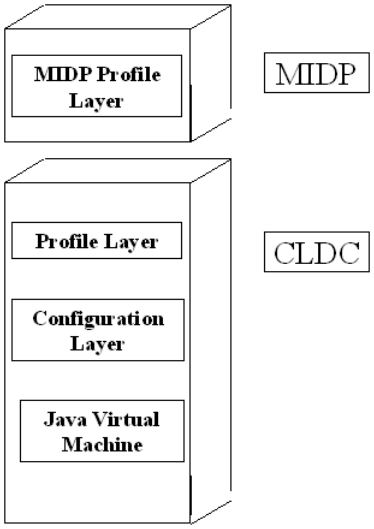


Abbildung 3.12: The J2ME architecture

3.3.1.1 Low Level Security

Dieser Level ist für die Sicherheit in Bezug auf die Virtuelle Maschine zuständig. Allgemein kann gesagt werden, dass die *low level security* verantwortlich dafür ist, dass die Class-Dateien, die in die Virtuelle Maschine geladen werden, auch nur auf dem erlaubten Wege ausgeführt werden, wie es auch in der Spezifikation der Java Virtuellen Maschine beschrieben ist.

3.3.1.2 Application Level Security

Die *application level security* beschreibt, dass die Anwendung nur auf die Bibliotheken, System-Ressourcen und anderen Komponenten zugreifen kann, auf die der Zugriff durch Gerät und Anwendungsumgebung erlaubt ist.

3.3.1.3 End-to-End Security

Die *end-to-end security* hat einen größeren Bereich abzudecken: Das Hauptziel der *End-to-end security* ist es, den sicheren Datenaustausch zwischen Server und Clients zu gewährleisten.

Bei der J2ME CLDC-Plattform ist die *low level security* und die *application security* bei CLDC angesiedelt und die *end-to-end security* im MIDP.

3.3.2 J2EE

Die Java 2 Enterprise Edition zielt auf Geschäftsanwendungen und hat ein besonders ausgefeiltes Sicherheitsmodell. Es werden zwar die Pakete und Mechanismen verwendet, die es auch in der J2SE Plattform gibt, aber bei J2EE wird schon spezifiziert, wann welcher Mechanismus eingesetzt werden muss. Die J2EE Plattform ist eine verteilte Umgebung, die eine Mehr-Schichten-Architektur definiert. Diese Schichten fassen logische Einheiten zusammen und stellen Funktionalitäten über Schnittstellen zur Verfügung. Wie diese Mehr-Schichten-Architektur bei der

J2EE Plattform aussieht, zeigt Abbildung 3.13. Die Sicherheit für die einzelnen Komponenten wird von den Containern gewährleistet. Es gibt dabei zwei Arten von Sicherheit: Die

- deklarative und
- programmatische

Sicherheit.
Die deklarative Sicherheit drückt die Sicherheit der Anwendungsarchitektur aus. Diese Sicherheit wird nicht in der Anwendung selbst, sondern durch eine externe Form ausgedrückt. Die programmatische Sicherheit ist in der Anwendung eingebettet. Die programmatische Sicherheit ist nützlich, wenn die deklarative Sicherheit nicht zur Beschreibung des Sicherheitsmodells ausreicht.
Innerhalb der J2EE Umgebung werden Zugriffsrechte über die Verbindung zwischen Nutzern, Gruppen, Rollen und Bereichen zu Methoden oder Endpunkten definiert.

- **Nutzer** - Eine einzelne Identität, die im *Application Server* definiert wurde. Nutzer können einer Rolle zugeteilt sein.
- **Gruppe** - Eine Menge von authentisierten Nutzern, die im *Application Server* definiert wurden.
- **Rolle** - Ein abstrakter Name für die Erlaubnis, auf eine bestimmte Ressource zuzugreifen.
- **Bereich** - Eine Menge von Nutzern und Gruppen, die durch die gleiche Authentisierungsrichtlinie kontrolliert wird

Für jede der Schichten einer J2EE Anwendung werden in [7] Sicherheitsmaßnahmen genannt, nur beim *Client-Tier* kann die Anwendung keine Sicherheitsmaßnahmen ergreifen.

3.3.2.1 Client Schicht

In der Client Schicht sind die Komponenten angesiedelt, die mit der J2EE-Anwendung interagieren. In den meisten Fällen ist dies ein *Web Browser* oder eine Netzwerk-Anwendung. Da diese Komponenten nicht unbedingt vom Programmierer der J2EE-Anwendung programmiert werden, ist es kaum möglich, für diese Komponenten Sicherheitsmaßnahmen durchzusetzen. Es besteht nur die Möglichkeit, die Verbindung zu sichern. Hierfür gibt es die Möglichkeit eine HTTPS-Verbindung aufzubauen.

3.3.2.2 Präsentationsschicht

Die Präsentationsschicht wird auch Web-Schicht genannt und repräsentiert die Komponenten, die für die Präsentationslogik zuständig sind (z.B. ein *Servlet* oder *Java Server Pages (JSP)*). Diese Komponenten sind für das *session management* und die Abarbeitung der Anfragen – zum Beispiel Anfragen über den *Browser* – zuständig. Diese Schicht ist das Bindeglied zwischen dem Client und der Anwendungslogik.
Für diese Schicht sind Sicherheitsmechanismen für die Authentisierung, Autorisierung, Interaktionssicherung, Sicherheit der Transportschicht vorgesehen. Für die Authentisierung werden in [7] mehrere Mechanismen genannt - wie zum Beispiel die *HTTP Basic Authentication* oder die *Form-Based Authentication.*

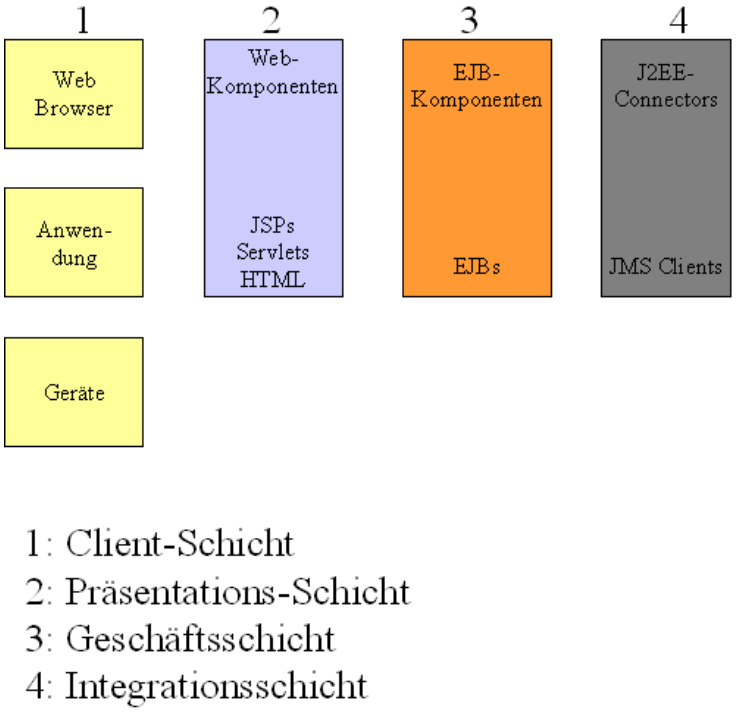


Abbildung 3.13: logische Schichten der J2EE Plattform [7]

3.3.2.3 Anwendungsschicht

In der Anwendungsschicht - auch EJB Schicht (von *Enterprise Java Beans*) genannt - liegt der Schwerpunkt der Sicherheit auf der Autorisierung und Zugriffskontrolle. Es gibt Ähnlichkeiten bei den Authentisierungs- und Autorisierungsmodellen im Vergleich zur Präsentationsschicht, aber die EJB Spezifikation beschriebt keine unterstützten Schemas. Dies liegt in der Verantworteung des J2EE Anbieters. In dieser Schicht spielt die Möglichkeit der deklarativen und programmatischen Sicherheit eine wichtige Rolle, während es bei anderen Schichten eine untergeordnete Rolle spielt.

3.3.2.4 Integrationsschicht

Die Integrationsschicht ist die Verbindung der Anwendungsschicht zu den Ressourcen, die hinter der Anwendung stehen wie zum Beispiel Datenbank Anwendungen oder *Enterprise Information System (EIS)*. J2EE Komponenten wie die der *Java DataBase Connectivity (JDBC)* sind in dieser Schicht angesiedelt. Hier gibt es über die *J2EE Connector Architecture (J2EE CA)* die Möglichkeit, solche Verbindungen zu sichern. Diese Sicherheitsmechanismen beinhalten die Identifikation und Authentisierung des Nutzers und die Sicherung der Verbindung zu den Ressourcen. Dazu werden Mechanismen wie HTTP Verbindungen über das SSL/TLS-Protokoll verwendet.

3.3.2.5 Ressourcenschicht

Alle back-end Ressourcen sind in der Ressourcenschicht angesiedelt.

3.4 Übersicht Sicherheitsmechanismen

In diesem Abschnitt wird aufgezeigt, welche Klassen für die einzelnen Sicherheitsmechanismen zuständig sind und wann die einzelnen Sicherheitsmechanismen in Java eingeführt wurden.

3.4.1 Einführung der Sicherheitsmechanismen

Einige Sicherheitsmechanismen wurden schon von Beginn an in Java implementiert, während die meisten Mechanismen erst im Nachhinein eingebaut wurden. Die einzelnen Sicherheitsmechanismen wurden auch nach ihrer Einführung in Java immer weiterentwickelt. Die Tabelle 3.2 gibt einen Überblick über die Einführung und Weiterentwicklung der Sicherheitsmechanismen.

3.4.2 Packages für Sicherheitsmechanismen

Für die verschiedenen Sicherheitsmechanismen wurden einzelne *packages* eingeführt. Diese sind in der Tabelle 3.3 aufgeführt, wobei immer nur das Hauptpaket genannt ist. Es gibt noch Erweiterungen, die in anderen *packages* liegen, aber im Hauptpaket sind die Basisklassen für eine Implementierung zu finden.

Sicherheitsmechanismus	Package
JSSE	javax.net.*
JAAS	javax.security.auth.*
JCE	javax.crypto.*
JCA	java.security.*
Security Manager	java.lang.SecurityManager
Permissions	java.security.Permission
Class Loader	java.lang.ClassLoader
CertPath	java.security.cert.*
Kerberos	javax.security.auth.kerberos
GSS-API	org.ietf.jgss

Tabelle 3.3: Packages der Sicherheitsmechanismen

3.5 Alternative Ansätze und Erweiterungen zur Java-Sicherheit

Java-Sicherheit wird nicht nur von Sun weiterentwickelt. Es gibt ein paar Ansätze aus der Java-Community oder von anderen Gruppierungen, die die Java-Sicherheit auf eigenem Wege erweitern. Zwei bekannte Projekte sind das OSGi-*Framework*[2] und die *Isolate-API*[3], die in den nächsten Abschnitten kurz erläutert werden.
OSGi ist - ähnlich wie SicAri - ein *Framework* zum Publizieren von Diensten und beschränkt sich bei der Umsetzung von Sicherheitsmaßnahmen nicht auf die Java-eigenen Sicherheitsmechanismen sondern erweitert sie. OSGi wird hier aufgeführt um aufzuzeigen wie in anderen Projekten, die SicAri ähnlich sind, mit dem Thema „Sicherheit“ umgegangen wird.

[2] http://www.osgi.org
[3] http://jcp.org/en/jsr/detail?id=121

Sicherheitsmechanismus	**JDK 1.0**	**JDK 1.1**	**JDK 1.2**	**J2SDK 1.3**	**J2SDK 1.4**	**J2SDK 1.5**
Sicherheitsmechanismen der Sprache	O	X	X	X	X	X
Virtuelle Maschine	O	X	X	X	X	X
Sandbox-Modell	O	X	X			
Signierter Inhalt		O	X	X	X	X
Sicherheits-Tools			O	X	X	X
Sicherheits-Policy			O		X	X
Java Cryptography Architecture (JCA)		O	X	X	X	X
Java Cryptography Extension (JCE)			O		X	X
Java Authentication and Authorization Sercive (JAAS)				O	X	
Java Secure Socket Extension (JSSE)			O		X	X
Java Certification Path API					O	X
Java GSS-API					O	

O : Einführung des Sicherheitsmechanismus
X : Weiterentwicklung

Tabelle 3.2: Einführung der Sicherheitsmechanismen [24]

3.5.1 OSGi

Das *Open Services Gateway Initiative (OSGi)-Framework* ist eine „*Service Delivery Plattform*“, die komplett in Java geschrieben wurde und einen Standard für Komponenten-orientierte Software definiert. Das Ziel von OSGi war es, ein vernetztes Haus zu schaffen, in dem alle Geräte miteinander kommunizieren können. Die Komponenten, die in diesem *Framework* Dienste anbieten, werden *bundles* genannt und sind als JAR-Dateien gekapselt. Die Komponenten können am *Framework* angemeldet und wieder abgemeldet werden. Angemeldete Komponenten können auch von anderen Diensten in Anspruch genommen werden. Sven Haiges nennt in [13] das Beispiel, dass ein GPS-Dienst in einem Auto, der ständig die aktuelle Position des Fahrzeugs ermittelt, von einem Navigationsdienst sowie von einem Dienst zur Anzeige der Position im Boardcomputer verwendet werden kann.
Dadurch entstehen Abhängigkeiten, die beachtet werden müssen. Dafür hat OSGi den *ServiceEvent* eingeführt. Ein *ServiceEvent* ist ein Ereignis, das immer ausgelöst wird, wenn ein Service sich an- oder abmeldet. Dadurch können abhängige Dienste dieses Ereignis behandeln. Mehr über OSGi ist unter [13], [17] und [27] zu finden.
Bei einem solchen *Framework* spielt die Sicherheit eine große Rolle. Im OSGi-*Framework* werden dafür die Sicherheitsmechanismen, die Java mit den oben beschriebenen Mechanismen bietet, implementiert und um eigene Ansätze ergänzt:
Anwendungen laufen auf der OSGi Service Plattform unter der Kontrolle eines Management Systems. Das Framework nutzt sehr extensiv die Permission-Funktionalitäten von Java. OSGi legt viel Wert auf die Sicherheit der kritischen Operationen. Ein Beispiel hierfür ist das Suchen eines Services, so dass das suchende Bundle gewisse Rechte haben muss. Die Rechte können über den *PermissionAdminService* angepasst werden und sind an einen sogenannten *Bundle-Location-Identifier* gebunden. Die Vergabe der Rechte basiert nicht auf der Signatur einer JAR-Datei.
Im OSGi-Framework sind drei verschiedene Sicherheitsrechte definiert [27]:

- AdminPermission
- ServicePermission
- PackagePermission

3.5.1.1 AdminPermission

*AdminPermissions*werden für hochgradig kritische Aufgaben - wie zum Beispiel Rechtekonfiguration - benötigt. Diese sollten nur an *Management Bundles* vergeben werden, denen vertraut werden kann.

3.5.1.2 ServicePermission

Diese Rechte erlauben es Services, sich zu registrieren, oder andere Services von der *Service-Registry* anzufordern. Die *ServicePermissions* werden benutzt, um nur den geeigneten Bundles das Anbieten und Benutzen von Services zu erlauben.

3.5.1.3 PackagePermission

Die *PackagePermissions* erlauben nur vertrauenswürdigen Bundles, Packages zu exportieren oder zu importieren. Das Exportieren eines Packages durch ein bösartiges Bundle könnte Attacken erlauben. Mit den *PackagePermissions* soll das verhindert werden.

3.5.2 Isolation API

Die *Isolation API* hat das Ziel, mehrere Java Anwendungen unter einer virtuellen Maschine laufen zu lassen. Zur Zeit ist es so, dass es für jede Java Anwendung, die ausgeführt wird, eine eigene virtuelle Maschine gibt (siehe Abbildung 3.14).

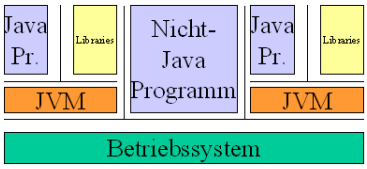


Abbildung 3.14: Mehrere Java Anwendungen - mehrere Virtuelle Maschinen [4]

Die in dem *Java Specification Request 121 (JSR 121)*[4] beschriebene Isolatione API löst dieses Problem. Jede Java Anwendung läuft innerhalb eines sogenannten Isolate. Ein Isolate ist eine Einheit, die unabhängig von anderen Java Anwendungen ist, also keine Anwendungsdaten teilt. Die Isolates haben den gleichen Sicherheitslevel als wenn sie als eigenständige Betriebssystem-Prozesse laufen würden.
Abbildung 3.15 zeigt, dass es nur noch eine virtuelle Maschine gibt, die mehrere Java Anwendungen startet.

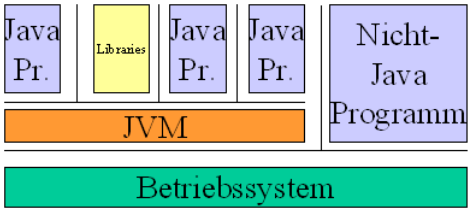


Abbildung 3.15: Mehrere Java-Anwendungen - Eine virtuelle Maschine [4]

Jeder Isolate kann eigenständig gestartet und gestoppt werden; der Lebenszyklus einer Anwendung kann vollständig von Java aus überwacht und kontrolliert werden. Die Begrenzungen zwischen den Isolates werden mit Mitteln von Java umgesetzt.

3.5.2.1 Zugewinn an Sicherheit

Isolates und Threads haben einige Ähnlichkeiten: Sie können miteinander kooperieren und der Lebenszyklus kann von Java aus gesteuert werden. Der große Unterschied besteht in der Sicherheit.
Die Isolation API ist eine Erweiterung der Java Sicherheit, da es die Java Anwendungen schützt. Threads können Objekte teilen, während Isolates keine Objekte teilen können. Sie können nur Endpunkte einer Kommunikation (z.B. Sockets oder Links) implementieren. Durch diese Abschottung kann es nicht passieren, dass ein Isolate durch „Absturz“ oder stoppen die Daten eines anderen Isolates in einem inkonsistenten Zustand bringt. Diese Gefahr besteht bei Threads.

[4] http://www.jcp.org/en/jsr/detail?id=121

Kapitel 4

Bedrohungs- und Angriffsanalyse

Eine Bedrohungs- und Angriffsanalyse ist notwendig, um eine Systematik in die Auffindung und Identifikation von möglichen Schwachstellen in der Anwendung zu bekommen. Durch diese Systematik ist es möglich, schneller und effizienter auf ein Gefährdungspotential zu reagieren. Das ist der Fall, weil an Stellen, die auf den ersten Blick als Gefahrenpunkt gedeutet werden, aber bei einer Bedrohungsanalyse als ungefährlich eingestuft werden, Zeit gespart wird.
Aber auch mit solch einer Methode, wie sie in diesem Kapitel vorgestellt wird, wird es nicht gelingen, ein 100%ig sicheres System zu schaffen. Es kann nur versucht werden, die möglichen Schwachstellen nach einer Prioritätenliste zu bearbeiten.
In den folgenden Abschnitten wird erst eine allgemeine Begriffsbestimmung gemacht, bevor auf die allgemeinen Ursachen für Gefahren eingegangen wird. Im darauffolgenden Abschnitt werden verschiedene Angriffsarten vorgestellt, bevor im abschließenden Abschnitt eine Methode zur Auffindung von Sicherheitslücken gezeigt wird.

4.1 Begriffsbestimmung

Bei der Bedrohungsanalyse werden bestimmte Begriffe verwendet, die hier kurz erläutert werden sollen.

Datenbestand

Als Datenbestand werden alle Daten, die von dem System erzeugt oder benötigt werden, bezeichnet. Dies sind zum Beispiel Daten aus einer Datenbank oder Dateien im Dateisystem.

Bedrohung

Eine Bedrohung ist ein mögliches Ereignis (bösartig oder nicht), durch das der Datenbestand beschädigt oder gefährdet werden kann.

Sicherheitslücken

Mit Sicherheitslücken werden Schwachstellen in der Anwendung oder im System bezeichnet, die eine Bedrohung ermöglichen.

Angriff

Ein Angriff ist die Aktion, bei der eine Sicherheistlücke ausgenutzt wird oder bei der einer Bedrohung nachgegangen wird.

Gegenmaßnahme

Um sich vor Angriffen zu schützen oder um die Gefährdung zu entschärfen, werden Gegenmaßnahmen umgesetzt.

4.2 Ursachen für Gefahren

Der größte Anteil von Sicherheitslücken wird durch drei Ursachen begründet: Fehlerhafter Code, Menschen und fehlerhafte Administration. Diese drei Ursachen werden in den drei folgenden Abschnitten genauer beschrieben.

4.2.1 Fehlerhafter Code

Fehlerhafter Code ist die Ursache für Gefahren, die am schwierigsten zu bekämpfen ist. Es gibt keine Tools, die fehlende Überprüfungen im Code erkennen. Ein häufiger Fehler im Code sind nicht überprüfte Arraygrenzen, die durch den Angreifer zu *Buffer Overflows* ausgenutzt werden können. Solche Pufferüberläufe können zwar durch Verwendung von Programmiersprachen wie Java oder Perl - die ohne Pointer-Arithmetik auskommt - verhindert werden, aber viele Programme werden in C geschrieben und Anwendungen stützen sich auf solchen Programmen. Allgemein sollte jede Benutzereingabe überprüft werden, da nie sichergestellt werden kann, dass immer verwertbare Eingaben getätigt werden. Diese fehlende Überprüfung von Benutzereingaben wird häufig in PHP-Applikationen ausgenutzt. So wurden einige Exploits bekannt, in denen der Angreifer bestimmen konnte, welche mit Hilfe des PHP-Befehls `popen` eigenen Code auf dem Server ausführen konnten.
Solch fehlerhafter Code wird meist für *Code-Injections* und/oder *SQL-Injections* ausgenutzt.
Die Gefahr von fehlerhaftem Code wächst mit dem Umfang und der Komplexität einer Anwendung. Umfangreiche und komplexe Anwendungen sind nicht mehr überschaubar und so bleiben viele Fehler - die von dem Compiler nicht entdeckt werden - von den Programmierern unentdeckt.
Programmverifikation mittels detailliert definierter Methoden-Schnittstellen (Vor- und Nachbedingungen sowie Invarianten) also auch zum Beispiel Unit-Tests sind Ansätze, um diese Ursache mehr oder weniger aufwändig zu minimieren.

4.2.2 Der Mensch

Der Mensch als Sicherheitsrisiko! Die Gruppe „Mensch“ kann in zwei Personenkreise eingeteilt werden:

- Interne Personen
- Externe Personen

Die größte Gefahr für Anwendungen geht durch interne Benutzer aus. Nach einem Artikel in der Computerwoche [20] wird ein Großteil aller Angriffe auf ein System von sogenannten Innentätern - den internen Mitarbeitern - ausgeführt. Dies wird auch durch einer Studie des *Computer Security Institutes (CSI)* von 2003 [30] gestützt. Bei vielen Systemen wird besonders auf Schutz nach außen geachtet - zum Beispiel mit *Firewalls*, das Schließen von Ports und Virenscannern - aber der Schutz nach innen bleibt unberücksichtigt.
Neu eingeführte Programme und Techniken überfordern teilweise den Nutzer und führen zu Fehlverhalten. Die Masse von Anwendungen, die man nur nach Authentitisierung nutzen kann, nimmt zu. Durch die steigende Zahl von Account-Daten, die der Nutzer sich behalten muss, werden die ausgewählten Passwörter einfacher, um sie sich einprägen zu können. Hier hilft der *Single-Sign-On (SSO)* Mechanismus, bei dem alle Anwendungen mit einem Account zugänglich sind. Die SSO-Lösung bedeutet jedoch indirekt ein größeres Risiko, da bei einem Passwort-Diebstahl gleich mehrere Anwendungen betroffen sind.
Ein weiterer Faktor ist die Technik, die zum Beispiel einen Datendiebstahl erleichtert. *Universal Serial Bus (USB)*-Sticks werden in vielen Unternehmen als Speichermedium verwendet. Ein Verlust des USB-Sticks bedeutet dann häufig auch ein Verlust von Unternehmensdaten. Eine fehlende Verschlüsselung der Daten hat dann oft weitreichende Folgen. Solche mobilen Speichermedien sind in vielen IT-Sicherheitsstrategien von Unternehmen nicht berücksichtigt [35].
Tanja Wolff teilt die Nutzer in vier Gruppen ein [34]:

- Sicherheits-Softie - Macht sich über die Sicherheitsbelange keine Sorgen
- Gadget-Freak - Über 50% der Umfrageteilnehmer schließen private mobile Endgeräte wie Digitalkamera oder USB-Stick an Firmen-Rechner an
- Der Illegale - Viele Nutzer laden unerlaubte und/oder persönliche Inhalte auf Firmen-Rechner
- Saboteur - eine kleine Gruppe von Nutzern, die versucht vorsätzlich in Bereiche einzudringen, für die sie nicht autorisiert ist

Eine größere Aufmerksamkeit genießen die externen Personen. Diese Gruppe teilt Claudia Eckert in [10] in vier Typen ein:

- Hacker
- Cracker
- *Skript Kiddie*
- Wirtschaftsspionage

Als Hacker wird ein Angreifer bezeichnet, der technisch versiert ist und versucht, Schwachstellen in einem IT-System zu finden. Ein *Hacker* verfolgt jedoch keine finanziellen Ziele oder einen persönlichen Vorteil. Er wendet sich vielmehr an die Öffentlichkeit, um eine Schwachstelle bekannt zu machen. Dem gegenüber steht der *Cracker*, der Angriffe ausführt, um einen persönlichen Vorteil zu erlangen. Diese Gruppe ist also ein wesentlich größeres Risiko als die Gruppe der Hacker.
Als *Skript Kiddie* werden Personen bezeichnet, die keine sehr tiefgehenden technischen Kenntnisse besitzen. *Skript Kiddies* versuchen bekannte *Exploits* auszunutzen und werden dabei häufig von Neugier getrieben.

Wirtschaftsspione hören Datenleitungen ab und versuchen an möglichst viele Informationen zu gelangen und dabei unerkannt zu bleiben. Diese Gruppe sorgt weniger für Datenverlust als für Integritäts- und finanziellen Verlust.

4.2.3 Fehlerhafte Administration

Die Administration eines Systems ist sehr entscheidend für die Sicherheit eines IT-Systems. Administratoren können die Art der Authentisierung, die Architektur des Systems und die Maßnahmen zur Sicherung bestimmen. Ein Beispiel fehlerhafter Administration sind die meisten Computer, die mit einem vorinstallierten Windows ausgeliefert werden. Viele Privatanwender richten keine Benutzerkonten ein oder alle eingerichteten Benutzerkonten haben Administratorenrechte. Dadurch wird es für Angreifer einfach, fremde Programme zu installieren ohne dass dies entdeckt wird.
Administratoren sind außerdem dafür zuständig, regelmäßig Patches zu installieren und Sicherungsmaßnahmen einzurichten. Die Sicherheit zu gewährleisten ist ein andauernder Prozess, das heißt, dass ein System zu keinem Zeitpunkt als sicher angesehen werden kann. Das *Bundesamt für Sicherheit in der Informationstechnik (BSI)* schreibt im Gefährdungskatalog 3[1], dass jede Modifikation an Einstellungen auch potentielle neue Schwachstellen sind.

4.3 Angriffsarten und -abwehr

Angriffe auf das System können in zwei Gruppen unterschieden werden; Zum einen in die passiven Angriffe und zum anderen in die aktiven Angriffe [10]. Während die passiven Angriffe auf den Verlust der Vertraulichkeit (siehe Kapitel 2.1.3) und die Informationsgewinnung zielen, betreffen die aktiven Angriffe eine nicht autorisierte Änderung von Daten und bedeuten somit einen Verlust der Datenintegrität (siehe Kapitel 2.1.2) oder der Verfügbarkeit des Systems (siehe Kapitel 2.1.5).
Für beide Arten von Angriffen gibt es - wenn teilweise auch nur eingeschränkt - die Möglichkeit, Angriffe zu verhindern. Diese Möglichkeiten werden im Abschnitt 4.3.2 beschrieben.

4.3.1 Angriffsarten

Die passiven Angriffe werden häufig nicht bemerkt, da keine unmittelbaren Auswirkungen auf das System zu erkennen sind und teilweise normalen Netzwerkproblemen ähneln. Ein typischer passiver Angriff ist das Abhören von Datenleitungen - das sogenannte *Sniffen*. Durch *Sniffen* von ungesicherten HTTP-Verbindungen ist es zum Beispiel möglich, Daten von verschiedenen Benutzerkonten zu bekommen. Eine weitere Art des passiven Angriffs ist die Verkehrsflussanalyse. Dabei analysiert der Angreifer, zu welchem Zeitpunkt welche Datenmengen über die Leitung gesendet werden.
Bei aktiven Angriffen werden Daten verändert. Beispiele hierfür sind das Entfernen und das Verändern von Datenpaketen, die über eine Leitung gesendet werden. In [10] werden die aktiven Angriffe in zwei Klassen geteilt:

- Maskierungsangriffe
- *Denial-of-Service*-Attacken

[1] http://www.bsi.de/gshb/deutsch/g/g03009.htm

Maskierungsangriffe - auch als *Spoofing* bezeichnet - versuchen, dem Opfer der Attacke eine falsche Identität zu zeigen. Häufig werden solche Maskierungsangriffe in der Form als *IP-Address-Spoofing* betrieben. Hierbei versucht der Angreifer dem Opfer vorzutäuschen, ein anderer *Host* zu sein. Damit kann der Angreifer Anfragen des *Clients* selbst beantworten. Ziel dieser Angriffe ist es, das Opfer dazu zu bringen, vertrauliche Daten bekannt zu geben - zum Beispiel Passwörter.
Die *Denial-of-Service*-Attacken (DoS-Attacken) zielen auf die Verfügbarkeit des Systems. Bei diesen Attacken versucht der Angreifer, das Zielsystem mit Nachrichten zu überschwemmen, so dass das Zielsystem die Anfragen nicht mehr bearbeiten kann und somit das System nicht mehr verfügbar ist.

4.3.2 Abwehr der Angriffe

Das Verhindern von passiven Angriffe ist teilweise nicht möglich, da es in Systemen auch verdeckte Kanäle gibt, die man kaum vollständig ausschließen kann. Mittels kryptographischer Verfahren kann man jedoch verhindern, dass vertrauliche Daten durch Abhören der Datenleitung verloren gehen. So kann zum Beispiel im Internet über *Secure-Socket-Layer*-Verbindungen (SSL) verhindert werden, dass ein Angreifer mit *Sniffern* Passwörter ausgespäht kann.
Aktive Angriffe, die auf den Verlust der Datenintegrität abzielen, können durch das Setzen von minimalen Rechten (zum Beispiel nur Leserechte für *group* und *all* unter Unix/Linux) meist verhindert, zumindest aber begrenzt werden. Durch das Beobachten (*Monitoring*) von gefährdeten Ressourcen können Angriffe zwar nicht verhindert werden, aber gezielte Eingriffe verhindern eine Nichtverfügbarkeit eines Systems.

4.4 Methode zur Bedrohungsanalyse

Die Bedrohungsanalyse für eine Anwendung sollte nicht nur einmal durchgeführt werden, sondern sollte ein ständiger Prozess sein. Da sich die Anwendung und die Erkenntnisse bei den Sicherheitsmechanismen ständig ändern, ändern sich die Ergebnisse der Bedrohungsanalyse. Microsoft beschreibt die Bedrohungsanalyse als einen Prozess mit sechs Teilschritten (vergleiche Abbildung 4.1 und [21]):

- Identifizieren zu schützender Daten/Komponenten
- Erstellen einer Architekturübersicht
- Gliedern der Anwendung
- Identifizieren der Bedrohungen
- Dokumentieren der Bedrohungen
- Bewerten der Bedrohungen

4.4.1 Identifizieren zu schützender Daten/Komponenten

Im ersten Schritt der Bedrohungsanalyse werden alle Datenbestände identifiziert, die als schützenswert gelten. Wie im Fall SicAri können das Nutzerdaten oder die *Policy*-Dateien sein. Bei großen Unternehmen gehören die Auftragsdatenbank und Kundendaten zu den schützenswerten Daten.

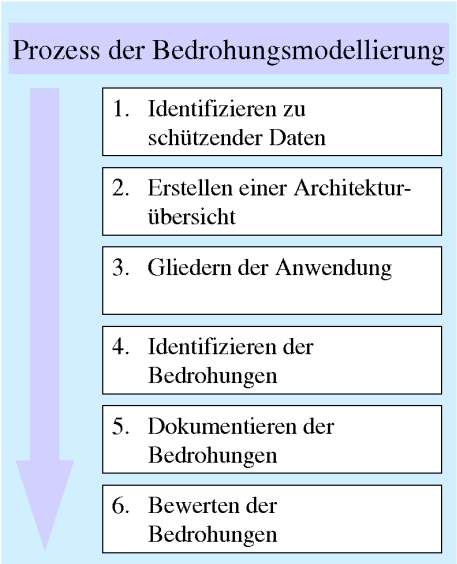


Abbildung 4.1: Prozess der Bedrohungsanalyse [21]

4.4.2 Erstellen einer Architekturübersicht

Der nächste Schritt besteht darin, eine Übersicht der Anwendungsarchitektur zu erstellen. Diese Übersicht enthält die Subsysteme und Vertrauensgrenzen. Dabei sollte nach möglichen Sicherheitslücken in der Architektur der Anwendung gesucht werden. In diesem Schritt sollen Funktion der Anwendung, das Architekturschema und die verwendeten Technologien bestimmt werden. Die Funktion der Anwendung kann durch Anwendungsfälle dargestellt werden. In SicAri wären das zum Beispiel

- Nutzer loggt sich ein
- Nutzer startet einen Dienst

Im Architekturschema werden die Zusammensetzung und die Struktur der Anwendung beschrieben. Dabei kann es bei komplexen Systemen notwendig sein, sich auf Subsysteme zu beschränken.
Durch die Identifizierung der Technologien ist es möglich, auch nach Technologiespezifischen Schwachpunkten zu suchen. So gelten einige ältere Kryptographiealgorithmen als unsicher.

4.4.3 Gliedern der Anwendung

Die Gliederung der Anwendung ist nötig, um ein Sicherheitsprofil der Anwendung erstellen zu können. Generell gilt, dass es umso einfacher ist, Schwachpunkte und Bedrohungen für die Anwendung aufzudecken, je mehr über die Abläufe der Anwendung bekannt ist. Dieser Schritt besteht aus fünf Unterschritten:

- Identifizieren von Vertrauensgrenzen
- Identifizieren des Datenflusses

- Identifizieren von Einstiegspunkten
- Identifizieren von privilegiertem Code
- Dokumentieren des Sicherheitsprofils

4.4.3.1 Identifzieren von Vertrauensgrenzen

Bei der Identifizierung der Vertrauensgrenzen werden alle Datenbestände der Anwendung berücksichtigt. Hierbei ist es notwendig, zu überprüfen, ob der Datenfluss oder Benutzereingaben vertrauenswürdig sind. Gegebenenfalls muss überlegt werden, wie der Datenfluss beziehungsweise die Benutzereingabe überprüft werden können. Weiterhin muss überprüft werden, ob der Code, der auf die Daten zugreift, vertrauenswürdig ist.

4.4.3.2 Identifizieren des Datenflusses

Der Datenfluss wird am besten von oben nach unten hin analysiert. Das heißt, dass zuerst das Gesamtsystem betrachtet wird und danach die einzelnen Subsysteme. Ein Beispiel für den Datenfluss in SicAri ist der Datenfluss zur Authentisierung: Passwort-Eingabe - Überprüfung des Passworts - Authentisierung.
Der Datenfluss über Vertrauensgrenzen hinweg ist besonders wichtig, da hier die Gefahr eines Angriffs am größten ist.

4.4.3.3 Identifizieren von Einstiegspunkten

Als Einstiegspunkte werden die Stellen bezeichnet, bei denen Subsysteme aufeinander zugreifen. Diese Stellen sind auch Einstiegspunkte für Angreifer. Als Beispiele für einen Einstiegspunkt seien eine Front-End-Webanwendung oder eine Datenbankverbindung genannt. Diese Einstiegspunkte sollten besonders bedacht werden, da hier die Eingaben und Zugriffe besonders überprüft werden müssen.

4.4.3.4 Identifizieren von privilegiertem Code

Privilegierter Code ermöglicht es, auf eine bestimmte Art auf sichere Ressoucen - wie Drucker oder Dateisysteme - zuzugreifen. Dieser Code erhält mehr Rechte als andere Codeteile und darf daher nur vertrauenswürdigem Code ausgesetzt sein.

4.4.3.5 Dokumentieren des Sicherheitsprofils

Der anschließende Schritt beinhaltet einen Katalog von Fragen (siehe Tabelle 4.1), mit dem Ansätze für verschiedene Implementierungsansätze identifziert werden können.

4.4.4 Identifizieren der Bedrohungen

In diesem Schritt der Bedrohungsanalyse sollen die Bedrohungen identifiziert werden, die für die Anwendung zutreffend sind. Dabei werden für die Arten der Angriffe die Szenarien durchgespielt.
Eine anerkannte Methode ist die Verwendung von Angriffsbäumen. Mit den Angriffsbäumen ist es möglich, die verschiedenen Angriffsszenarien in einer anschaulichen Art und Weise darzustellen. Für die Darstellung des Angriffsbaums kann ein hierarchisches Diagramm (siehe Abbildung 4.2) oder eine einfache Gliederung (siehe Listing 1) verwendet werden.

Kategorie	**Überlegungen**
Eingabeüberprüfung	Werden Eingabedaten überprüft? Könnte ein Angreifer Befehle oder bösartige Daten in die Anwendung schleusen?
Authentifizierung	Sind die Anmeldeinformationen bei der Übertragung geschützt? Werden sichere Kennwörter erzwungen?
Autorisierung	Wird ein sicherer Abbruch ermöglicht und der Zugriff nur nach erfolgreicher Bestätigung der Anmeldeinformationen gewährt?
Konfigurationsverwaltung	Welche Konfigurationsspeicher werden verwendet und wie sind diese geschützt?
Sicherheitsrelevante Daten	Welche sicherheitsrelevanten Datenwerden von der Anwendung behandelt? Welche Art von Verschlüsselung wird verwendet?
Sitzungsverwaltung	Wie werden Sitzungscookies erzeugt? Wie werden diese geschützt?
Kryptographie	Welche Algorithmen und kryptographischen Techniken werden verwendet? Wie lang sind die Verschlüsselungsschlüssel und wie sind diese geschützt?
Parametermanipulation	Werden verfälschte Parameter von der Anwendung erkannt?
Ausnahmeverwaltung	Wie werden Fehlerbedingungen von der Anwendung behandelt? Werden allgemeine Fehlermeldungen verwendet, die keine verwertbaren Informationen enthalten?
Überwachen und Protokollieren	Werden Aktivitäten von der Anwendung überwacht? Wie werden Protokolldateien geschützt?

Tabelle 4.1: Fragenkatalog zur Erstellung des Sicherheitsprofils [21]

Die Knoten eines Angriffsbaums können mit UND und ODER verbunden werden. Damit wird gekennzeichnet, ob zwei Bedingungen zusammen zutreffen müssen oder ob die Erfüllung einer Bedingung ausreicht. Die Angriffsbäume zu erstellen ist zeitaufwändig und kann schnell komplex werden, deshalb wird häufig die Textdarstellung vorgezogen.

Listing 4.1

1. Ziel eins
    (a) Unterziel eins.eins
    (b) Unterziel eins.zwei
2. Ziel zwei
    (a) Unterziel zwei.eins

(b) Unterziel zwei.zwei

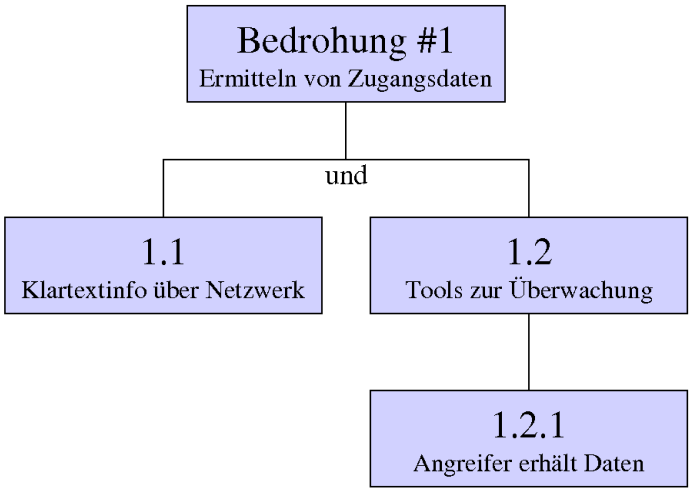


Abbildung 4.2: Beispiel eines Angriffsbaums [21]

4.4.5 Dokumentieren der Bedrohungen

Die Dokumentation der Bedrohung kann relativ einfach in tabellarischer Form erfolgen, in der die Attribute und deren Werte eingetragen werden. Diese Form der Dokumentation ist übersichtlich und kann durch Vorlagen sehr schnell gemacht werden. In Tabelle 4.2 wird eine solche Dokumentation gezeigt.

Beschreibung der Bedrohung	**Angreifer erhält Anmeldeinformationen durch Überwachen des Netzwerks**
Ziel der Bedrohung	Benutzerauthentifizierungsprozess bei SicAri
Risiko	wird erst später ermittelt
Angriffstechniken	Verwendung von Programmen zur Netzwerküberwachung
Gegenmaßnahmen	Verwendung von SSL, um einen verschlüsselten Kanal bereitzustellen

Tabelle 4.2: Dokumentation der Bedrohung [21]

4.4.6 Bewerten der Bedrohungen

Beim Abschluss der Bedrohungsanalyse werden die einzelnen Bedrohungen noch bewertet, um eine Priorisierung zu bekommen. Durch die Bewertung wird es möglich, schwerwiegende Bedrohungen mit hoher Eintrittswahrscheinlichkeit von leichten Bedrohungen mit niedriger Eintrittswahrscheinlichkeit zu unterscheiden. Eine sehr einfache Möglichkeit, das Risiko einer Bedrohung zu bewerten, ist folgende Formel:

$$Risiko = Wahrscheinlichkeit \times Schadenspotenzial$$

Es ist jedoch ersichtlich, dass diese Formel sehr wenig Differenzierungsmöglichkeiten bietet. Deshalb kann eine Bewertungsmatrix verwendet werden. Bei Microsoft wird das DREAD-Modell verwendet [21]. Dabei werden folgende Kriterien bewertet:

- **D**amage (Schadenspotential): Wie groß ist der Schaden wenn die Sicherheitslücke ausgenutzt wird?
- **R**eproducibility (Reproduzierbarkeit): Wie leicht kann der Angriff reproduziert werden?
- **E**xploitability (Ausnutzbarkeit): Wie leicht kann ein Angriff gestartet werden?
- **A**ffected User (betroffene Benutzer): Wie viele Benutzer können betroffen sein?
- **D**iscoverability (Auffindbarkeit): Wie leicht kann die Sicherheitslücke aufgefunden werden?

Für jedes Kriterium werden Punkte nach einem einfachen Schema wie niedrig (1), mittel (2) und hoch (3) vergeben. Beispielhaft wird das in Tabelle gezeigt.

Bedrohung	**D**	**R**	**E**	**A**	**D**	**Gesamt**	**Bewertung**
Angreifer erhält Anmeldeinformationen durch Überwachen des Netwerkes	3	3	2	2	2	12	hoch

Tabelle 4.3: DREAD-Modell - Beispiel

Kapitel 5

Sicherheitsplattform SicAri

Die SicAri-Plattform ist eine Plattform, die dynamisch Dienste lädt und zur Verfügung stellt. Diese Dienste laufen innerhalb einer Umgebung, die die Sicherheit gewährleisten soll. Das Hauptziel ist, den sicheren ubiquitären Internetzugang zu ermöglichen. Dabei bietet SicAri, für den Benutzer transparent und integriert, verschiedene Sicherheitsdienste. In diesem Kapitel wird SicAri vorgestellt und wie die Plattform in [25] spezifiziert ist. Als erstes wird ein Überblick gegeben, aus welchen Komponenten die Plattform besteht und welchen „Gültigkeitsbereich“ die Plattform hat. In den darauffolgenden Abschnitten werden die Basisdienste von SicAri eingeführt. In den abschließenden Abschnitten wird auf einzelne Sicherheitsmechanismen in SicAri eingegangen.

5.1 Überblick

SicAri ist in einer Mehrschicht-Architektur aufgebaut. Die oberste Schicht ist die Anwendungsschicht. Auf dieser sind alle Anwendungen angesiedelt, die der Benutzer von SicAri benutzt. Diese Anwendungen wiederum nutzen die Dienste von SicAri, wie den Authentisierungdienst oder die Einbindung von Datenbanken anbieten. Die Dienste von SicAri sind in der Serviceschicht angesiedelt. Die *Middleware* ist für die Integration der Dienste und Anwendungen verantwortlich (vergleiche Abbildung 5.1).
Die gesamte Plattform wurde in Java implementiert. Auch wenn alle Java-Plattformen (J2ME,

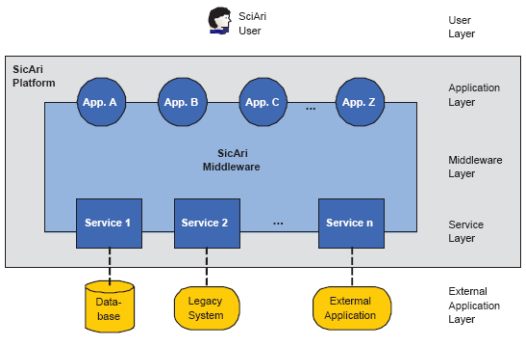


Abbildung 5.1: Architektur-Überblick von SicAri [25]

J2SE, J2EE) abgedeckt werden sollen, sind die Einschränkungen der Sicherheitsmechanismen bei J2ME zu groß. Aus diesem Grund wird nur die J2SE-Plattform berücksichtigt. In SicAri werden die von Java implementierten Sicherheitsmechanismen eingesetzt, wobei diese teilweise durch eigene Implementierungen ersetzt wurden.

5.2 Anwendungsbereich

Die Hauptfunktion der Plattform ist es, Schnittstellen für den Zugriff auf Dienste, die von SicAri bereitgestellt werden, zur Verfügung zu stellen. Zusammen mit der *Middleware* wird auch eine Infrastruktur für die Kommunikation zwischen verteilten SicAri-Komponenten bereitgestellt.
Die Plattform bietet dem Nutzer die Möglichkeit, selbst Dienste über die Plattform zur Verfügung zu stellen.

Bei SicAri werden die sicherheitsrelevanten Aspekte durch eine Sicherheitsrichtlinie geregelt. Wenn mehrere Plattformen innerhalb einer Infrastruktur interagieren, verwenden alle Plattformen die gleiche Sicherheitsrichtlinie. Als SicAri-Infrastruktur wird der Verbund mehrerer Plattformen bezeichnet. Jede Plattform kümmert sich selbständig um die Durchsetzung dieser Richtlinien. Versuche, auf Dienste zuzugreifen, werden auf die Einhaltung der Richtlinien hin überprüft.
Laufen jedoch zwei unterschiedliche SicAri-Infrastrukturen und haben diese unterschiedliche Sicherheitsrichtlinien, muss über eine gemeinsame Richtlinie verhandelt werden. Es existiert ein Unterprojekt, das sich mit der Entwicklung eines entsprechenden Protokolls beschäftigt.

5.3 SicAri-Architektur

Die Abbildung 5.2 zeigt die Architektur von SicAri:

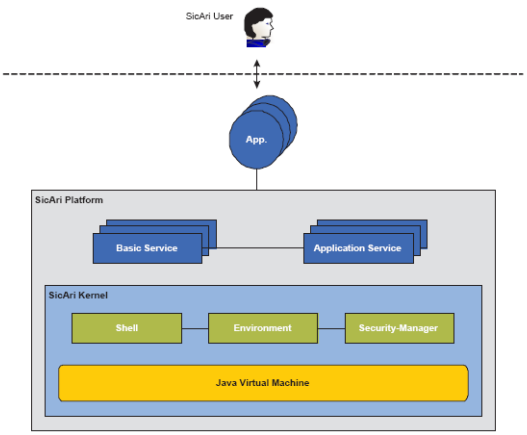


Abbildung 5.2: Architektur von SicAri [25]

Der Kernel läuft in der virtuellen Maschine und hat noch die drei Komponenten

- Shell
- Environment

- SecurityManager

Um den Kernel herum wird dann die Plattform aufgebaut. Außerhalb des Kernels laufen die Basisdienste und die Anwendungsdienste. Diese Schicht zusammen mit dem Kernel bildet die SicAri-Plattform. Externe Anwendungen agieren mit der Plattform, indem zum Beispiel ein Basisdienst aufgerufen wird. Diese externen Anwendungen wiederum interagieren mit dem SicAri-Nutzer.
Der Kernel ist für das *Bootstrapping* der Dienste und deren Verwaltung zuständig. Die Diensteverwaltung beinhaltet die Suche nach Diensten und die Registrierung von Diensten. Auch die Kontrolle der Zugriffsrechte gehört in das Aufgabengebiet des SicAri-Kernels. Neben den genannten Eigenschaften hat der Kernel noch folgende Aufgaben:

- Konfiguration der Plattform
- Überprüfung von Abhängigkeiten zwischen Diensten
- Sicheres Laden von Klassen und Trennung von Threads
- Administration über eine Shell-basierte Schicht
- Monitoring und logging

Die drei Hauptaufgaben des SicAri-Kernels sind Identitätsmanagement, Durchsetzung der Sicherheits-Policy und das Management der Services, die in Abschnitt 5.4 eingehender erläutert werden.

5.3.1 Kernel-Komponenten

Wie zuvor erwähnt, besteht der SicAri-Kernel aus den drei Komponenten Shell, Environment und Security Manager, die in den folgenden Abschnitten beschrieben werden.

5.3.1.1 Shell

Die Shell bietet die Oberfläche für den Administrator der SicAri-Instanz. Diese Shell ist der UNIX-Shell sehr ähnlich. Die angebotenen Funktionalitäten werden durch Java-Klassen und -Methoden implementiert. Über die Shell kann der Administrator während der Laufzeit Dienste laden, löschen und konfigurieren.
Eine weitere Analogie zur UNIX-Shell ist die Möglichkeit, Skripte für die SicAri-Shell zu schreiben und es gibt SicAri-eigene Befehle. Die Syntax der SicAri-Shell ist in [25] beschrieben. Mittels der Skripte ist ein automatisiertes *Bootstrapping* möglich (siehe Kapitel 7.1).
Zusätzlich zum lokalen Login kann sich ein *Client* auch auf einem entfernten Rechner anmelden (siehe Abbildung 5.3 - (1) und (3)). Dazu gibt es einen Dienst, der einen SSH-ähnlichen *Server* implementiert.

5.3.1.2 Environment

Als Environment wird der hierarchisch strukturierte Namensraum der Dienste bezeichnet. Dieser Namensraum ist dem Dateisystem nachempfunden, wobei die Pfade der Dienste nur virtuelle Pfade sind. Die Pfade können - ähnlich wie im Dateisystem - mit Rechten versehen werden. Auch im SicAri-Environment gibt es drei Arten von Rechten:

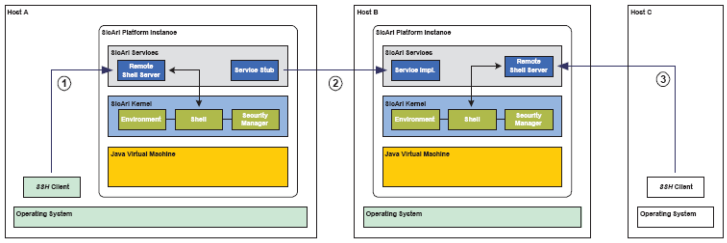


Abbildung 5.3: Interaktionen mit und zwischen SicAri-Instanzen [25]

- lookup (Leserecht)
- publish (Schreibrecht)
- retract (Schreibrecht)

Das Environment ist die für das Dienstemanagement zuständig. Die Interaktionen zwischen Diensten, Kernel und Dienst oder Anwendung und Dienst passiert nur über das Environment und somit ist das Environment ein wichtiger Bestandteil bei der Durchsetzung der Sicherheitsrichtlinien.
Durch sogenannte dynamische Proxies werden Dienste voneinander getrennt. Es gibt drei verschiedene Arten von Proxies:

Security Proxy

Wenn eine Methode über den *Security Proxy* aufgerufen wird, wird die Autorisation des Aufrufs überprüft. Ein weiterer Grund für die Implementierung des *Security Proxies* ist es, Dienste im Kontext des Nutzers aufzurufen, der den Dienst gestartet hat. Wenn ein Dienst über den *Security Proxy* aufgerufen wird, wird zuerst überprüft, ob der Aufrufende die benötigten Rechte besitzt. Danach wird der Kontext in den Kontext des Nutzers geändert, der den Dienst gestartet hat.

Plain Proxy

Der *Plain Proxy* leitet die Methodenaufrufe direkt an den gekapselten Dienst weiter. Durch ein bestimmtes Signal löscht der *Proxy* die Referenz zu dem gekapselten Objekt und überlässt es dem *Garbage Collector*. Werden danach Methoden aufgerufen, die zu dem gelöschten Objekt weitergeleitet werden sollen, gibt der *Proxy* einen Fehler aus.

Asynchronous Proxy

Dieser Typ von *Proxy* startet einen neuen *Thread* beim Aufruf einer Methode. Der aufrufende Thread wartet, bis das Ergebnis der Methode verfügbar ist. Allerdings kann der aufrufende Thread einen *Timeout*-Wert setzen und so den *Thread* löschen, selbst wenn das Ergebnis noch nicht vorliegt.

Bei dem *Plain Proxy* und dem *Aynchronous Proxy* werden nur die öffentlichen Methoden des Dienstes gekapselt, wodurch eine Verminderung der Gefahr von DoS-Attacken erreicht wird, da

eventuell schlecht programmierte oder fehlerhafte Methoden des Dienstes nicht angesprochen werden können.

5.3.1.3 Sicherheitskontext

Das dritte Element des SicAri-Kernels ist der Sicherheitskontext. Die Shell und das Environment implementieren einige Sicherheitsmechanismen, zusätzlich dazu hat der SicAri-Kernel diesen Sicherheitskontext. Der Sicherheitskontext ist für die Durchsetzung der Sicherheits-Policy über den Sicherheits-Manager notwendig.
Dienste können mit ihrem eigenen Sicherheitskontext gestartet werden. Normalerweise werden die Dienst mit dem gleichen Sicherheitskontext wie der Administrator gestartet. Mit dem Sicherheitskontext ist es möglich, die Rechte des Dienstes zu beeinflussen.
Der Sicherheitsmanager von SicAri überlässt die Durchsetzung der Sicherheits-Policy einem anderen Modul. Die Sicherheits-Policy ist vom Benutzer abhängig, der in dem Moment auf den Dienst zugreift.

5.3.2 Sicherheitsarchitektur

Die Sicherheitsarchitektur von SicAri besteht aus Hard- und Softwarekomponenten, die wichtige Ressource - wie Datenbanken oder E-Mail-Dienste - schützen und kontrollieren. Dieser Abschnitt gibt eine Einführung in das *Policy-Management* und in die Autorisierungsmechanismen in SicAri. SicAri legt für alles sicherheitsrelevanten Angelegenheiten eine Sicherheits-Policy fest. Alle Aktionen - wie z.B. das Starten eines neuen Dienstes - innerhalb der SicAri-Plattform werden auf die Einhaltung dieser Richtlinie hin überprüft.

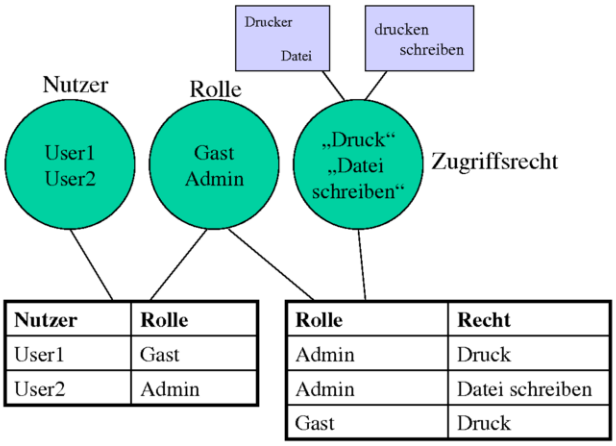
5.3.2.1 Autentisierung und Zugriffskontrolle

Zwei zentrale Mechanismen des Sicherheits-*Frameworks* der SicAri-Plattform sind Authentisierung und Zugriffskontrolle. Der Zweck von Authentisierung und Autorisierung wurde in Abschnitt 2.2.1 und Abschnitt 2.2.2 erläutert. In diesem Abschnitt wird darauf eingegangen, was diese Mechanismen in SicAri bedeuten.
Bei Computersystem ist ein Subject eine Einheit, die eine Anfrage initiieren kann. Bei SicAri kann dies ein Prozess, ein *Thread*, eine Komponente oder ein Agent sein. Ein Object ist die Einheit, auf die diese Anfrage ausgeführt wird - zum Beispiel eine Datei, eine Datenbank oder andere Systemressource wie ein Drucker.
Der Vorgang der Authentisierung wird in den Abschnitten 5.4.3 und 5.5.1 genauer beschrieben. Für die Zugriffskontrolle wurden viele verschiedene Ansätze entwickelt, zum Beispiel die Zugriffsmatrix. In SicAri wird der *Role-based Access Control (RBAC)* Mechanismus verwendet.

5.3.2.2 Rollenbasierte Zugriffskontrolle

Der RBAC Mechanismus wird in diesem Abschnitt nur in den Grundzügen beschrieben. Mehr Details sind in der Literatur zu finden ([11]). Dieser Mechanismus beschreibt vier Komponenten mit steigender Anzahl von Eigenschaften und Anforderungen.
Die Grundbegriffe bei RBAC sind *user*, *role* und *permission*. Mit diesen Begriffen sind auch Mengen verbunden. In einem System gibt es also eine Menge „user“, eine Menge „role“ und eine Menge „permission“. Aus diesen Mengen werden die Beziehungen *user* x *role* und *permission* x *role* gebildet. Damit werden die Zuordnungen von Nutzern zu Rollen und Rolle zu

Zugriffsrecht abgebildet (siehe Beispiel 5.4). Ein Zugriffsrecht bestimmt sich aus der Art des Zugriffs (Operation) und dem Objekt, auf das dieser Zugriff ausgeübt werden soll.
Durch RBAC können die Zugriffsrechte eines Nutzers sehr einfach erweitert oder eingeschränkt werden, indem dieser Nutzer eine neue Rolle bekommt oder eine Rolle gelöscht wird. Es können auch Hierarchien von Rollen und Rechten gebildet werden. So kann zum Beispiel ein Gruppe „Autor“ das Recht haben, eine Datei zu schreiben und eigene Dateien zu verändern. Eine Gruppe „Editor“ erbt diese Rechte von „Autor“ und bekommt zusätzlich noch die Rechte, Dateien von anderen „Autoren“ zu bearbeiten.
Die Zugriffskontrolle basiert hier nicht mehr auf einer einfachen Nutzer-Recht-Zuordnung, sondern auf der Zuordnung von Nutzer zu Rolle und welche Rechte diese Rolle hat.



Nutzer	Rolle
User1	Gast
User2	Admin

Rolle	Recht
Admin	Druck
Admin	Datei schreiben
Gast	Druck

Abbildung 5.4: RBAC Beispiel

5.3.2.3 Kontextabhängige Zugriffskontrolle

In SicAri wird die rollenbasierte Zugriffskontrolle durch die Dimension „Kontext“ erweitert. In diesem Parameter wird zum Beispiel gespeichert, welche Methode beim Login verwendet wurde. Je sicherer die Methode ist (z.B. X.509 Zertifikate gegenüber User/Passwort), desto mehr wird dem authentisierten Subjekt vertraut. Damit können zum Beispiel mehr Rechte verbunden sein.

5.3.2.4 SicAri Referenz-Monitor

Der Referenz-Monitor ist ein abstraktes Konzept, bei dem jeder Zugriff eines Subjekts auf ein Objekt durch den Referenz-Monitor gestattet werden muss. Der Referenz-Monitor besteht dabei aus zwei Teilen - zum einen aus dem *Policy Enforcement*, das die Überprüfung der Zugriffsrechte an eine *Policy Decision*-Komponente weiterleitet. Mit dem Referenz-Monitor wird keine festgelege Sicherheitsrichtlinie durchgesetzt. Die Spezifikation dieser Richtlinie wird außerhalb des Monitors in einer extra Komponente festgelegt (vergleiche Abbildung 5.5).

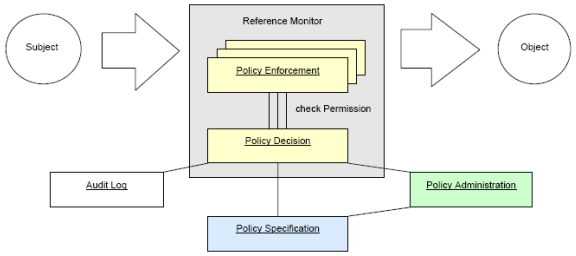


Abbildung 5.5: SicAri-Referenz-Monitor [25]

5.4 Basisdienste in SicAri

Basisdienste sind für die Durchführung der Sicherheitsaufgaben notwendig. In den nachfolgenden Abschnitten werden diese Basisdienste beschrieben.

5.4.1 Durchsetzen der Policies und SecurityManager

Dieser Abschnitt erläutert die Komponenten des SicAri-Referenz-Monitors (siehe Abschnitt 5.3.2.4) eingehender und in welche Prozesse diese Komponenten eingebunden sind.
Sobald auf eine sensitive Ressource zugegriffen wird, überprüft die *Enforcement*-Komponente, ob der Nutzer die benötigten Rechte hat.
Es gibt zwei Arten wie diese Komponente aktiviert werden kann. Zum einen kann die Aktivierung explizit geschehen. In diesem Fall muss jede SicAri-Komponente sicherstellen, dass die *Policy*-Komponente verwendet wird, sobald auf eine Ressource zugegriffen wird. Da in SicAri jeder Nutzer Komponenten zu SicAri hinzufügen kann, kann nicht sichergestellt werden, dass auch tatsächlich immer die *Policy*-Komponente verwendet wird. Aus diesem Grund ist die explizite Aktivierung der *Enforcement*-Komponente eine fehleranfällige Art.
Im Gegensatz dazu steht die implizite Aktivierung. In diesem Fall können die SicAri-Komponenten die Policy-Komponente nicht umgehen, da diese im Hintergrund immer läuft. Dieser Ansatz wird auch in SicAri eingesetzt.
Die implizite Aktivierung wird dadurch erreicht, dass in SicAri ein eigener *security manager* gesetzt wird. In Java läuft der *security manager* im Hintergrund und kontrolliert alle Zugriffe. Dafür gibt es in der Klasse verschiedene `checkXXX`-Methoden. Da der Java-eigene *security manager* nicht alle Operationen in einer gewünschten Art und Weise überprüft, wird in SicAri ein eigener *security manager* implementiert.
Der *security manager* leitet alle Entscheidungen bezüglich der Zugriffe an die Entscheidungskomponente des Referenz Monitors weiter.
Ein großer Vorteil dieses Ansatzes ist es, dass keine Anwendungen an den SicAri *security manager* angepasst werden muss, da der *security manager* automatisch eine `checkPermission`-Methode aufruft, die entweder nichts zurückliefert oder einen Fehler auswirft.

Policy Decision Komponente
Der einzige Zweck dieser Komponente ist es, Entscheidungen bezüglich der Zugriffe auf Basis der zugrundeliegenden Sicherheitsrichtlinie zu treffen. Dazu werden die User-ID, die Ressourcen-ID und die ID der Operation benötigt. Die User-ID ist eine eindeutige Kennung für einen SicAri-Nutzer, die Ressourcen-ID bezeichnet die Ressource, auf die der Zugriff erfolgen soll, eindeutig.

Die ID der Operation gibt den Typ der Operation an, zum Beispiel Lesezugriff oder Schreibzugriff.
Um die Entscheidung treffen zu können, benötigt die hier beschriebene Komponente Zugriff auf die Komponente, die die Sicherheitsrichtlinie bereithält (vergleiche Abbildung 5.5).

5.4.2 Kontextmanagement

Der Kontextmanager verwaltet Informationen zum Environment. Dazu gehören Informationen über User und Services. Da der Kontext Manager auch den Kontext der aktuellen Sitzung eines Nutzers bereitstellt, ist dieser Teil wichtig für die Entscheidungen bezüglich der *Policy*.

5.4.3 Authentisierungsmanagement

Der Authentisierungs Manager arbeitet eng mit dem Identitätsmanagement und dem Schlüsselmanagement zusammen, um die Authentizität eines Subjekts innerhalb der SicAri-Plattform zu gewährleisten. Dabei stellt der Authentisierungs Manager die Mechanismen zur Authentisierung zur Verfügung. Ein Subjekt gilt als authentisiert, wenn es den Besitz einer Identität bewiesen hat, die im Identitätsmanagement hinterlegt ist. Im *Keystore* ist der private Schlüssel des Subjekts hinterlegt.
Grundsätzlich gibt es drei Formen der Authentisierung:

- Benutzername und Passwort
- Token und PIN
- Authentisierung mit privatem Schlüssel

Für jeden Nutzer wird ein X.509-Zertifikat erzeugt und gespeichert.

5.4.4 Identitätsmanagement

Als Identitätsmanagement wird die zentrale Verwaltung von Nutzern - inklusive Rechtevergabe und Rollenzuweisung - bezeichnet. Das Identitätsmanagement soll sicherstellen, dass jeder SicAri-Service den Nutzer auf sichere Art und Weise identifizieren kann. Um dies zu gewährleisten, wird jeder User einem eindeutigen Zertifikat zugeordnet.
Das Identitätsmanagement ist auch dafür verantwortlich, eine Zuordnung unterschiedlicher Repräsentationsmöglichkeiten eines Users herzustellen. Diese unterschiedlichen Repräsentationsmöglichkeiten können zum Beispiel E-Mail-Adressen, User ID und Namen sein.

5.4.5 Schlüsselmanagement

Beim Schlüsselmanagement werden die privaten Schlüssel von Nutzern gespeichert. Dieser Dienst stellt zwei Funktionen nur für den Authentisierungsmanager zur Verfügung: Zwischenspeichern von Schlüsseln für einen *Single-Sign-On* Mechanismus und Ausführen von Kryptographischen Operationen wie Signierung und asymmetrische Verschlüsselung.

5.5 Sicherheitsmechanismen in SicAri

In Abschnitt 5.3.2 wurden die Konzepte der Sicherheitsarchitektur in SicAri angesprochen. Wie einzelne Konzepte in SicAri umgesetzt sind, wird in diesem Abschnitt besprochen. Bei diesen Konzepten handelt es sich um die Authentisierung, die Autorisation und der *Policy Service*.

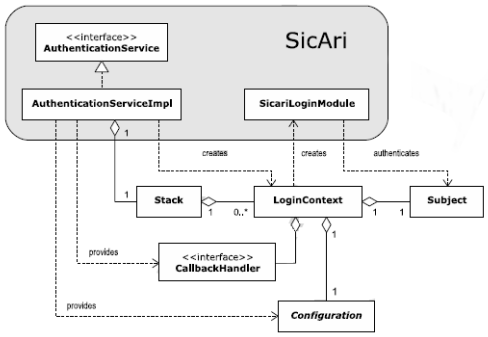


Abbildung 5.6: Klassendiagramm des Authentisierungsmechnismus

5.5.1 Login mittels JAAS

Für den Login-Mechanismus wird die JAAS-Architektur (siehe Kapitel 3.2.2.3) von Java erweitert. Nach dem Start von SicAri wird der Authentisierungsdienst - repräsentiert durch die Klasse `AuthenticationServiceImpl` - gestartet. Abbildung 5.6 zeigt das Klassendiagramm des Authentisierungsmechanismus in SicAri.
`AuthentisierungsService` ist eine Schnittstelle *interface* für Authentisierungsdienste in SicAri und `AuthenticationServiceImpl` ist die aktuelle Implementierung dieser Schnittstelle. In der `AuthenticationServiceImpl` wird auch ein Stack geführt, durch den eine Login-Hierarchie wie unter UNIX möglich ist. Der `LoginContext` beschreibt die Methoden, mit denen ein Subjekt authentisiert wird. Durch den `LoginContext` sind diese Methoden unabhängig von den darunterliegenden Mechanismen.
Über eine Konfigurationsdatei wird festgelegt, welche Klasse die Methoden zur Authentisierung implementiert. In diesem Fall ist es die Klasse `SicariLoginModule`. In dieser Klasse wird eine Überprüfung der Zugriffsrechte veranlasst und das Passwort über den `CallbackHandler` abgefragt. Weiterhin wird der *Principal* des Subjekt-Logins erzeugt.

5.5.2 PolicyService mittels Permissions

Für die Rechtevergabe wird das XACML-*Framework* verwendet. XACML steht hierbei für *eXtensible Access Control Markup Language* und ist ein Subtyp der *eXtensible Markup Language (XML)*. Mit diesem Framework sollen die Autorisierungs-Richtlinien standardisiert werden. Für die Verwendung in Java-Applikationen kann das Paket `sunxacml` [1] verwendet werden.
Die Klasse `PDP` (*Policy Decision Point*) verwendet ein paar Klassen (`xxxFinderModule`), die für das Parsen der XACML-Dateien und das Bereitstellen der Informationen zuständig sind. Diese XACML-Dateien enthalten Informationen über die User und deren Zuordnung zu Gruppen.
Abbildung 5.7 zeigt das Klassendiagramm des *Policy Services*. Das *Interface* für den Service verwendet den SicAri-eigenen *Security Manager*, damit - wie in Abschnitt 5.4.1 beschrieben - die Sicherheitsrichtlinien durchgesetzt werden können. `PolicyServiceImpl` ist die konkrete Implementierung des *Interfaces* und verwendet die Klasse `PDP`, die wiederum die Finder-Module verwendet, um an die XACML-Informationen zu kommen.

[1] http://sunxacml.sourceforge.net

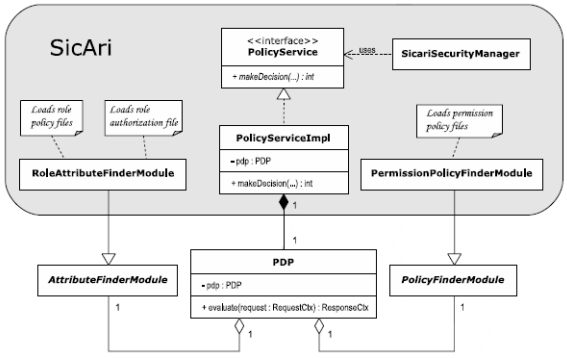


Abbildung 5.7: Klassendiagramm des PolicyServices

Insgesamt werden in diesem *Policy Service* drei Arten von XACML-Dateien benötigt:

- Datei mit der Zuordnung Nutzer - Rolle (Listing 5.1)
- Datei mit Sicherheitsrichtlinien für die Rollen (Listing 5.2)
- Zuordnung der Zugriffsrechte (Listing 5.3)

Listing 5.1

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <Policy xmlns="..." xmlns:xsi="..."
 3     xsi:schemaLocation="..."
 4     PolicyId="Role:Assignment:Policy"
 5     RuleCombiningAlgId="...">
 6   <Rule RuleId="guest:role:requirements" Effect="Permit">
 7     <Target>
 8       <Subjects>
 9         <Subject>
10           <SubjectMatch MatchId="...">
11             <AttributeValue
12               DataType="...">renee4</AttributeValue>
13             <SubjectAttributeDesignator
14                   AttributeId="..." DataType="..."/>
15           </SubjectMatch>
16         </Subject>
17       </Subjects>
18       <Resources>
19         <Resource>
20           <ResourceMatch MatchId="...">
21             <AttributeValue DataType="...">
22           urn:renee:role-values:guest</AttributeValue>
23             <ResourceAttributeDesignator
24                    AttributeId="..." DataType="..."/>
```

```
25          </ResourceMatch>
26        </Resource>
27      </Resources>
28      <Actions>
29        <Action>
30          <ActionMatch MatchId="...">
31            <AttributeValue DataType="...">
32               attr
33            </AttributeValue>
34            <ActionAttributeDesignator AttributeId="..."
35                                       DataType="..."/>
36          </ActionMatch>
37        </Action>
38      </Actions>
39    </Target>
40  </Rule>
41 </Policy>
```

In Zeile 4 wird mit `PolicyId` ein eindeutiger Bezeichner für diese Policy festgelegt. Mit `RuleId` (Zeile 6) wird ein eindeutiger Bezeichner festgelegt, der nach Möglichkeit der entsprechenden Rolle entspricht (in diesem Fall `guest`). In Zeile 12 wird dann die ID des Users (hier: renee4) einer Rolle zugewiesen. Über die `Resource`-Angabe (Zeile 22) wird die Rollen-URI bezeichnet, die in einer der Policy Dateien (siehe Listing 5.2) bezeichnet ist.

Listing 5.2

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2   <PolicySet xmlns="..." xmlns:xsi="..."
 3              xsi:schemaLocation="..."
 4              PolicySetId="RPS:guest:role"
 5              PolicyCombiningAlgId="...">
 6     <Target>
 7       <Subjects>
 8         <Subject>
 9           <SubjectMatch MatchId="...">
10             <AttributeValue DataType="...">
11           urn:renee:role-values:guest</AttributeValue>
12             <SubjectAttributeDesignator
13                     AttributeId="..." DataType="..."/>
14           </SubjectMatch>
15         </Subject>
16       </Subjects>
17     </Target>
18   <PolicySetIdReference>
19                 PPS:guest:role</PolicySetIdReference>
20 </PolicySet>
```

Über `PolicySetId` (Zeile 4) wird der eindeutige Name der Policy festgelegt. Beim `Subject`

im `AttributeValue` (Zeile 11) wird die URI der Rolle festgelegt. Diese dient für die Verbindung zur Rollenzuweisung des Nutzers (vergleiche mit Listing 5.1, Zeile 22). In Zeile 19 wird die Referenz zu der Datei festgelegt, in der die Rechte vergeben werden (dargestellt in Listing 5.3).

Listing 5.3

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <PolicySet xmlns="..." xmlns:xsi="..."
 3            xsi:schemaLocation="..."
 4            PolicySetId="PPS:guest:role"
 5            PolicyCombiningAlgId="">
 6   <Target/>
 7   <Policy PolicyId="guest:policy001"
 8           RuleCombiningAlgId="...">
 9     <Target/>
10     <Rule Effect="Permit"
11        RuleId="guest:policy001:permit:Permission:*:*">
12       <Target>
13         <Resources>
14           <Resource>
15             <ResourceMatch MatchId="...">
16               <AttributeValue DataType="...">
17                java.lang.RuntimePermission .*
18               </AttributeValue>
19               <ResourceAttributeDesignator
20                           AttributeId="..."
21                           DataType="..."/>
22             </ResourceMatch>
23           </Resource>
24         </Resources>
25         <Actions>
26           <Action>
27             <ActionMatch MatchId="...">
28               <AttributeValue DataType="...">
29                 read
30               </AttributeValue>
31               <ResourceAttributeDesignator
32                           AttributeId="..."
33                           DataType="...">
34             </ActionMatch>
35           </Action>
36         </Actions>
37       </Target>
38     </Rule>
39   </Policy>
40 </PolicySet>
```

Auch in dieser Datei wird über `PolicySetId` (Zeile 4) der eindeutige Bezeichner für die Rechte-Richtlinie festgelegt. Dieser sollte - wenn möglich - mit der Rollenbezeichnung übereinstimmen (vergleiche Listing 5.2, Zeile 4). Für jede einzelnen `Policy` wird über `PolicyId` der Namen festgelegt (Zeile 7). Es gibt mehrere `Rules` in jeder `Policy`. Diese `Rules` auch einen eindeutigen Namen brauchen (Zeile 11). Innerhalb der `Rule` wird dann festgelegt, welche Rechte vergeben werden sollen und die dazugehörigen Parameter (Zeile 17). In Zeile 29 wird das Attribut gesetzt, das für alle vorhergenannten Ressourcen gilt. In diesem Fall wird der Lesezugriff erlaubt.

Kapitel 6

Bedrohungs- und Angriffsanalyse SicAri

In Kapitel 4 wurden allgemeine Gefahren genannt. In diesem Kapitel werden diese Gefahren auf SicAri angewendet und bewertet. Dazu wird die Methode zur Bedrohungsanalyse angewendet (vergleiche Kapitel 4). Es werden jedoch unter Berücksichtigung der Bearbeitunszeit nur drei Teilaspekte betrachtet. Als erstes wird das Problem besprochen, das in der Implementierung angegangen wird.
SicAri hat nicht den Anspruch, Sicherheitslücken im Betriebssystem oder in der Java Virtual Machine zu schließen. Aus diesem Grund betrachten wir für mögliche Implementierungen nur den Ablauf und die Mechanismen von SicAri.

6.1 Starten von SicAri

Beim Starten von SicAri werden die ersten Aktionen ausgeführt ohne dass sich ein Plattform-Administrator eingeloggt hat. Beim Start der Shell wird durch den `SicariSecurityManager` überprüft, ob sich die Shell-Klasse in einer Liste von vertrauenswürdigen Klassen befindet. Diese Liste ist aber für Angreifer leicht austauschbar. Nachdem eine Instanz der Shell erzeugt wurde, werden ein paar Basisdienste gestartet. Diese Basisdienste sind

- Authentisierungsdienst
- Identitätsmanagement
- Schlüsselverwaltung

Nach dem Laden dieser Dienste meldet sich der Plattform-Administrator an und es werden weitere Dienste gestartet. Die Shell läuft jedoch weiterhin ohne Kontext. Der gesamte Vorgang des *Bootstrappings* wird in Kapitel 7 erläutert. Bei einem Login eines Users, wird keine neue Instanz der Klasse `Shell` erzeugt, sondern es läuft immer noch die gleiche Shell wie beim Administrator. Bei dem Login wird der neu eingeloggte Nutzer einem *Stack* – der in der Klasse `Shell` gespeichert wird – hinzugefügt. Die gesamte Nutzerverwaltung läuft über diesen *Stack*, über den auch der aktuelle User für die Überprüfung von Zugriffsrechten ermittelt wird.
Aufgrund des beschriebenen Ablaufs ergeben sich folgende zu schützende Daten:

- Nutzerdaten (Logindaten)
- Klassen, die für das Starten von SicAri benötigt werden

- trusted.classes (siehe Abschnitt 6.1.1)

Zur Zeit behandelt SicAri diese Daten innerhalb einer Vertrauensgrenze, die in Abbildung 6.1 gezeigt wird.
Die Shell startet über ein Skript die beiden Dienste Identitätsmanagement und Authentisierungsmanagement. Die Aufgaben der beiden Dienste sind in Kapitel 5 beschrieben. Jeder Zugriff auf Ressourcen wird durch den `SicariSecurityManager` überprüft. Wenn eine Klasse instanziert wird, überprüft der *Security Manager*, ob diese Klasse in der Datei `trusted.classes` aufgeführt ist.
Bei jedem Login übernimmt der Authentisierungsservice die Überprüfung der Authentisierung und interagiert dabei über die Shell mit dem User. Im Identitätsmanagement werden die Identitäten aller Nutzer verwaltet. Der Nuter kann Services starten. Beim Start überprüft der *Security Manager*, ob die notwendigen Rechte erteilt wurden.

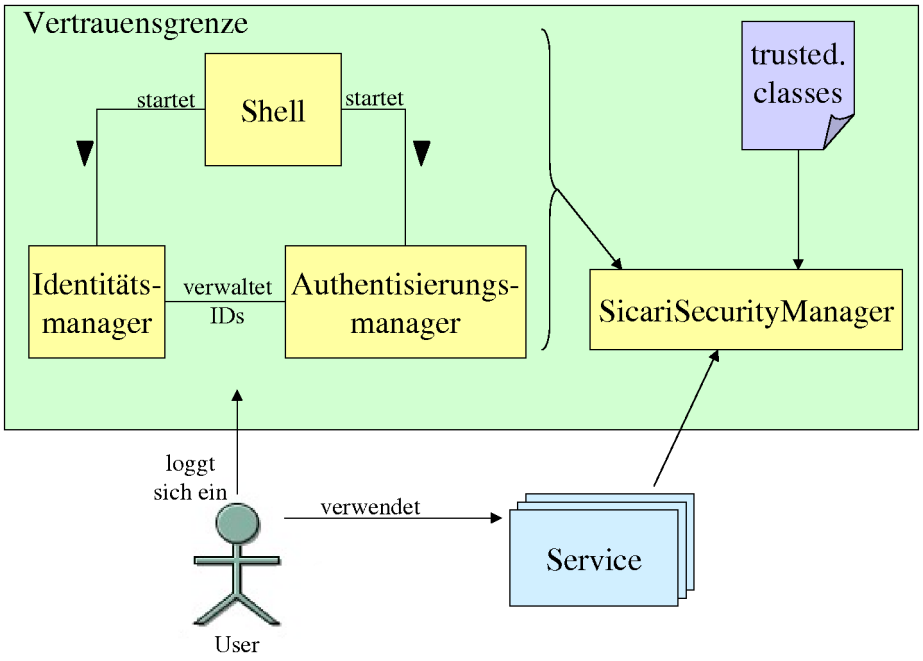


Abbildung 6.1: Architekturübersicht

Im Folgenden werden einige Punkte des *Bootstrappings* genauer betrachtet.

6.1.1 Vertrauenswürdige Klassen

Da eine Liste der Klassen, denen in SicAri ohne vorherige Prüfung vertraut wird, als Datei vorliegt, bestehen drei Möglichkeiten, anzugreifen:

- Erweitern der Liste
- Austausch der Datei
- Klasse in Package speichern

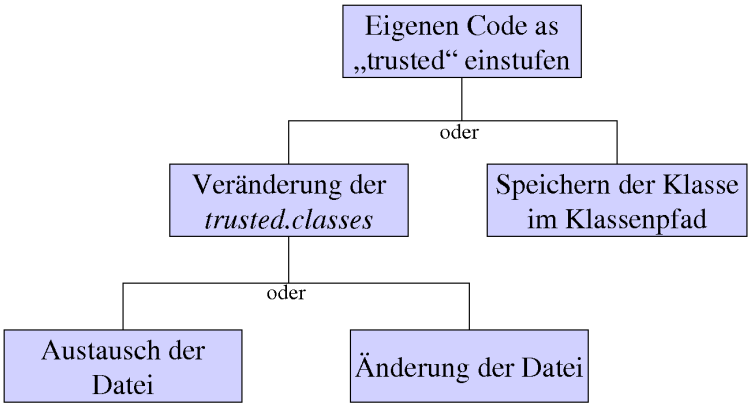


Abbildung 6.2: Bedrohungsbaum „Vertrauenswürdige Klassen“

Die Datei enthält Informationen über vertrauenswürdige Klassen in der folgenden Form:
`de.sicari.kernel.*`
`de.fhg.*`
Diese *package*-Namen werden durch eine Klasse in SicAri soweit ausgewertet, dass alle Klassen des angegebenen *packages* zu den vertrauenswürdigen Klassen zählen.
Durch eine Erweiterung der Liste mit z.B.

`my.evil.package.*`

werden alle Klassen des *packages* `my.evil.package` als vertrauenswürdig betrachtet.
Der Bedrohungsbaum (siehe Abbildung 6.2) für diesen Fall zeigt, dass ein möglicher Angriff auf drei Wegen passieren kann.

6.1.2 Userverwaltung

Wie beschrieben, geschieht die Userverwaltung für die Shell mit einem *Stack*, in dem der `LoginContext` des Nutzers gespeichert wird. Dadurch ist es möglich, dass ein Angreifer den *Stack* manipuliert und so den `LoginContext` des Administrators für eigene Operationen verwendet.

6.1.3 Starten der Sicherheitsrelevanten Dienste

Die aktuelle Version von SicAri sieht vor, dass der Plattform-Administrator über ein Skript steuert welche Dienste gestartet werden nachdem die erste Shell geöffnet wurde (siehe Listing 6.1). Dadurch ist es möglich, dieses Startskript so zu verändern, dass keine Dienste gestartet werden oder eigene Dienste gestartet werden (siehe Listing 6.2). Dies gilt auch für die Sicherheitsrelevanten Dienste `PolicyService`, `IdentityManager` und `AuthenticationService`. Ein Angreifer kann - vom Plattform-Administrator unbemerkt - das Startskript verändern und die sicherheitsrelevanten Dienste durch eigene Dienste ersetzen.

Listing 6.1 (rc.main)

```
###
# Startskript Plattform-Administrator
##
# Sicherheitsdienste starten
echo „Sicherheitsdienste starten...“
java de.sicari.security.AuthenticationService \
  -publish default
```

Listing 6.2 (rc.main.hacked)

```
###
# Startskript Plattform-Administrator
###
# Sicherheitsdienste starten
echo „Sicherheitsdienste starten...“
java de.sicari.security.MyHackedAuthenticationService \
  -publish default
```

6.2 Sicherheitslücken durch Java

Eine Sicherheitslücke in SicAri wird durch die native Java-Implementierung geschaffen:
Beim ersten Aufruf einer Anwendung, die das *Abstract Window Toolkit (AWT)* verwendet, wird eine *Event-Queue* gestartet, die auf alle zukünftigen Ereignisse, die von einer Anwendung mit AWT erzeugt werden, reagiert. Diese *Event-Queue* bekommt die Rechte der Person, die das erste Mal eine Anwendung mit AWT startet.

Kapitel 7

Implementierung

In diesem Kapitel wird beschrieben, welcher Vorgang in SicAri im Rahmen dieser Bachelor-Thesis geändert wurde. Vorher wird noch beschrieben, wie dieser Vorgang bisher abgelaufen ist.
Das SicAri-*Bootstrapping* ist der Vorgang, bei dem die Sicherheitsplattform SicAri und einige Basisdienste gestartet werden.

7.1 SicAri-Bootstrapping bisher

Das *Bootstrapping* beginnt mit dem Aufruf der Klasse Shell. Zu Beginn wird die SicAri-Shell (siehe Kapitel 5.3.1.1) initialisiert und das Startskript wird ausgeführt. In diesem Startskript wird auch festgelegt, welche Basisdienste gestartet werden sollen. Diese Basisdienste werden in keinem Benutzerkontext gestartet, da noch kein Login erfolgte. Das liegt daran, dass diese Basisdienste - zum Beispiel der Authentisierungsdienst - für das Login notwendig sind.
Nachdem diese Basisdienste gestartet sind, muss sich der Plattform-Administrator einloggen. Wenn die Authentisierung erfolgreich war, muss der *Keymaster* eingelesen werden. Dieser ist für die kryptographischen Dienste - die später gestartet werden - notwendig. Wie der *Bootstrapping* Vorgang bisher abgelaufen ist, ist in Abbildung 7.1 zu erkennen.

7.2 SicAri-Bootstrapping neu

In der bisherigen Implementierung von SicAri war die Shell die Einstiegsklasse. Für das neue Bootstrapping wurden folgende Punkte vorgesehen:

- Erstellen einer signierten JAR-Datei mit allen notwendigen Klassen
- Datei `trusted.classes` so weit wie möglich durch interne Liste ersetzen (siehe Abschnitt 7.2.5)
- Administrator-Login möglichst früh
- Festlegen von Diensten die gestartet werden müssen
- Konfigurationsdatei als Parameter
- Bei User-Login neue Shell erzeugen

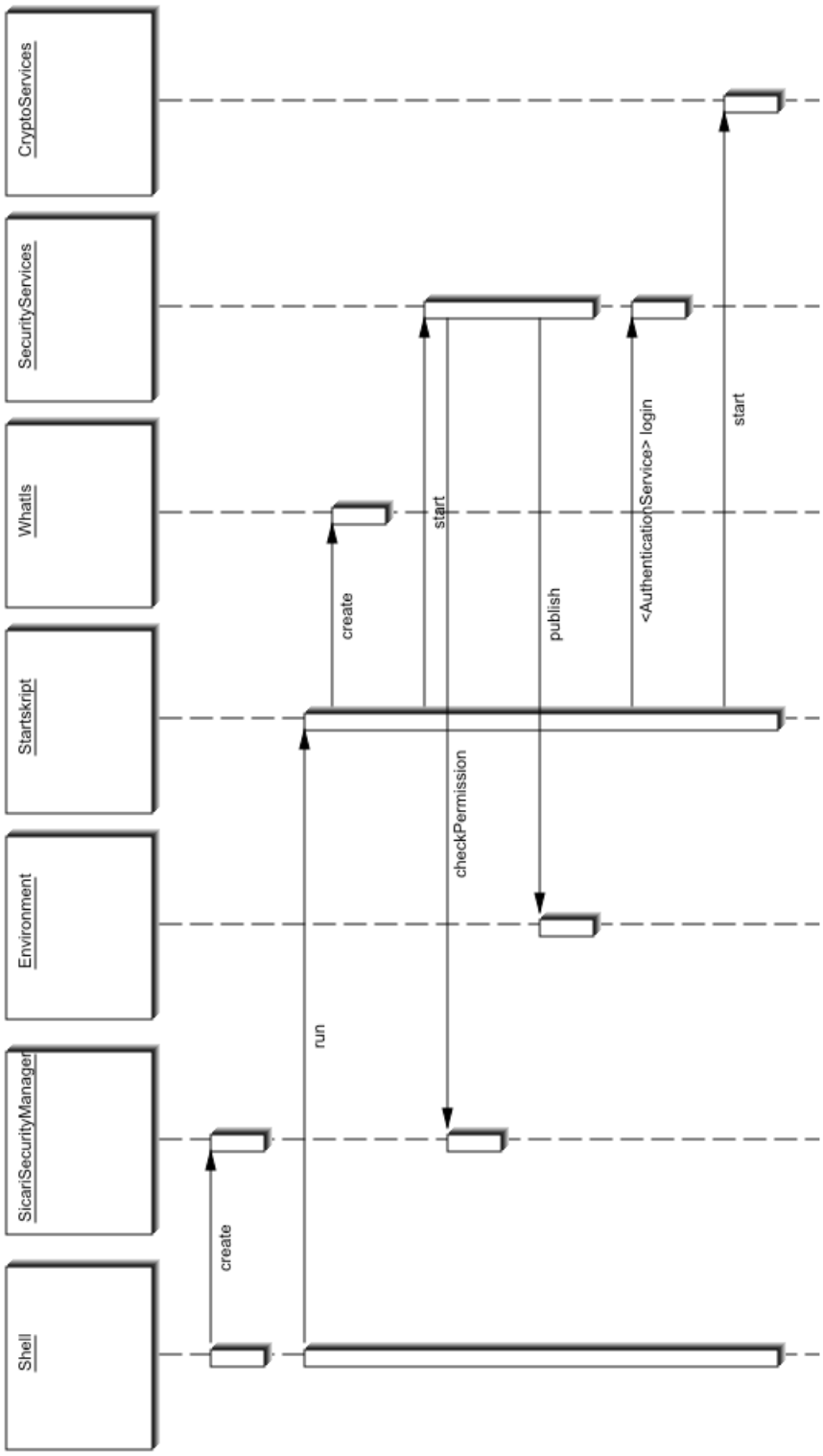


Abbildung 7.1: Bootstrapping „alt“

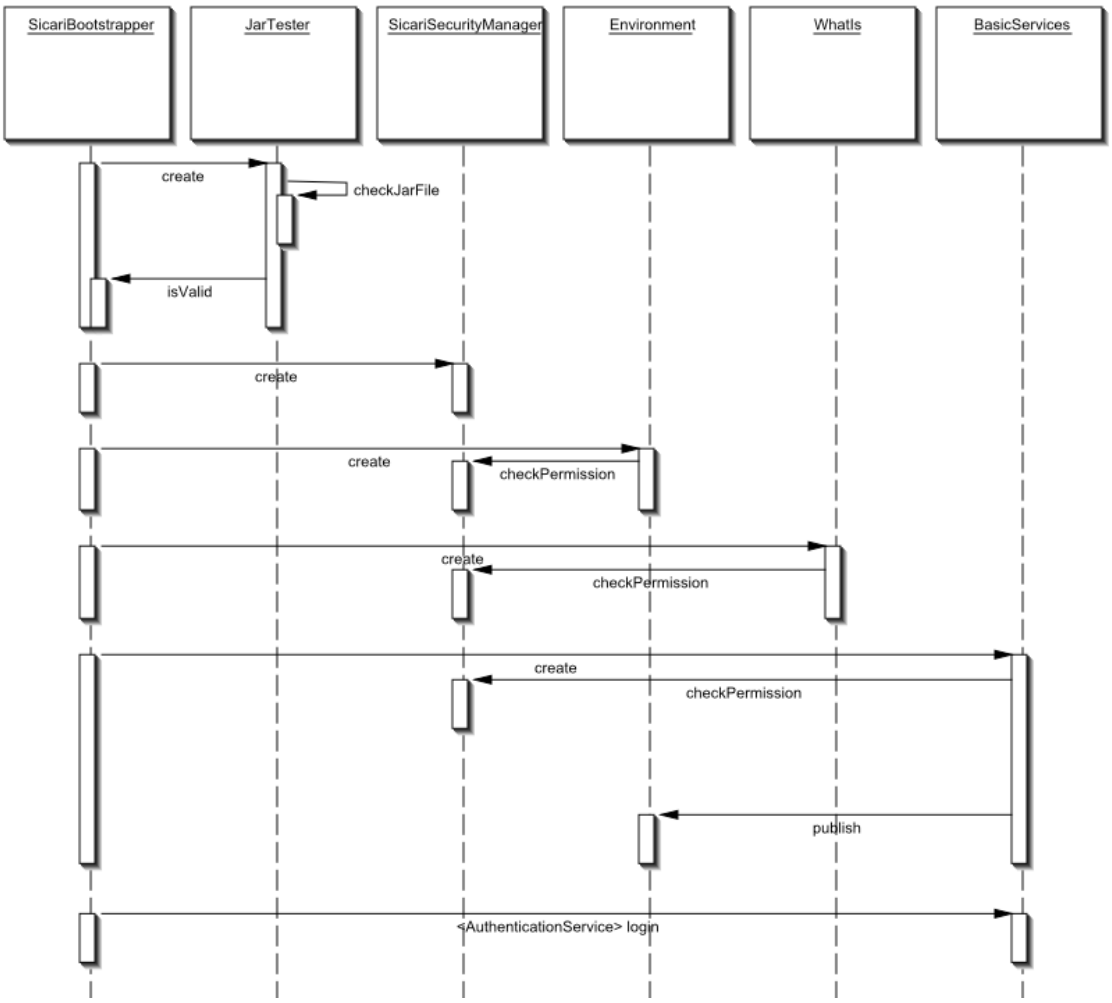


Abbildung 7.2: Bootstrapping „neu“

- Anpassen der Shell-Syntax
- Shell darf nicht ohne Subject laufen

Damit ergibt sich folgender Ablauf beim *Bootstrapping* (siehe Abbildung 7.2).
Beim Start von SicAri wird der `SicAriBootstrapper` aufgerufen, der die JAR-Datei testet. Dabei ist es wichtig, dass alle Dateien im Archiv auch durch die Signatur abgedeckt sind. Alle Dateien aus dieser signierten JAR-Datei werden als vertrauenswürdige Klassen angesehen, wenn die Signatur gültig ist. Nachdem das JAR-Archiv überprüft wurde, wird der Standard *Security Manager* durch den `SicariSecurityManager` ersetzt. Im Konstruktor wird zuerst das `WhatIs` gestartet, das für das spätere *Lookup* (Suche nach Diensten) notwendig ist. Danach werden die Basisdienste gestartet. Die Basisdienste sind:

- PolicyService
- IdentityManager
- AuthenticationService

Dies ist notwendig, damit der erste Login schon auf Basis der Dienste geschehen kann.
Nachdem diese Dienste gestartet sind, muss sich der Plattform-Administrator anmelden, damit er andere Dienste starten kann. Durch die Änderung des *Bootstrapping* Vorgangs wird bei jedem Login eines Nutzers eine neue Shell instanziert. Dadurch werden die Variablen und die *History*-Listen zurückgesetzt. Bei der alten Implementierung wurde auch immer der aktuelle Pfad, in dem der Nutzer sich befand, übernommen. Jetzt beginnt der neu eingeloggte Nutzer wieder im *root*-Verzeichnis.
Für die Implementierung wurden einige Klassen geändert oder neu geschrieben. In den folgenden Abschnitten werden die Klassen beschrieben, die für den *Bootstrapping*-Vorgang wichtig sind und geändert werden mussten.

7.2.1 SicariBootstrapper

Der `SicariBootstrapper` ist die neue Startklasse von SicAri. Im `static`-Block - der vor dem Konstruktor ausgeführt wird - wird der JarTester erzeugt, der die JAR-Datei überprüft. Der Administrator bekommt die Zertifikatskette angezeigt, damit er sich entscheiden kann, ob er den Klassen aufgrund der Zertifikate vertraut oder nicht. Als nächstes wird der *Security Manager* gesetzt.
Im Konstruktor wird zuerst eine Environment-Instanz erzeugt. Da Basisdienste gestartet werden sollen, die über das Environment auffindbar sein sollen, wird eine Environment-Instanz benötigt. Nach dem Environment wird die Klasse `WhatIs` instanziert. Diese Klasse speichert eine `HashMap`, in der die logischen Pfade zu den Diensten gespeichert sind.
Im Anschluss werden die Basisdienste gestartet und in das Environment eingetragen. Der nächste Schritt ist der Login des Plattform-Administrators. Dadurch wird eine neue Shell instanziert.

7.2.2 JarTester

Die Klasse `JarTester` ist komplett neu entstanden. Diese Klasse hat die Aufgabe *Java ARchive (JAR)*-Dateien zu testen. Dazu bekommt der Konstruktor den Pfad der zu testenden Datei übergeben. Danach wird das `Manifest` der JAR-Datei ausgelesen und alle dort gelisteten Dateien werden in eine `HashMap` gespeichert. Danach werden alle Einträge der Datei überprüft. Dies beruht nicht auf den Einträgen des Manifests, da es sonst zu Problemen kommen kann (siehe Kapitel 7.2.5).
Zum Schluss wird noch überprüft, ob alle Datein aus der JAR-Datei im `Manifest` stehen. Ist dies der Fall, ist die JAR-Datei valide.

7.2.3 AuthenticationServiceImpl

In der alten Version wurden die Nutzer über einen Stack verwaltet. Da dies ein Angriffspunkt ist und der aktuelle Nutzer über den *Access Controller* herausgefunden werden kann, wurde der Stack komplett entfernt. Die Methode `currentUser()` wurde dahingehend angepasst, dass der aktuelle Nutzer nicht mehr vom *Stack*, sondern vom `AccessControlContext` ausgelesen wird.
Die Anfrage des aktuellen Subject, an das öffentliche und private Daten eines Nutzers im Rahmen seines Sicherheitskontextes gebunden sind, ist also nur noch diesem Nutzer möglich. Eine Speicherung an einem anderen Ort – wie vorher in dem *Stack* geschehen – ist nicht mehr vorgesehen.

7.2.4 RunCommand

Nach dem erfolgreichen Login eines Nutzers lief die Shell weiterhin ohne Sicherheitskontext. Erst bei der Ausführung drei verschiedener Shell-Befehle wurden die entsprechenden Aktionen im Rahmen des aktuellen Sicherheitskontexts des authentisierten Nutzers ausgeführt. Das entfällt jetzt komplett, da die komplette Shell mit dem Subject des aktuellen Nutzers - und damit auch die Methoden von `RunCommand` - ausgeführt werden.

Damit ergibt sich folgendes Klassendiagramm (siehe Abbildung 7.3). Für die Sicherheit in SicAri bedeutet der neue *Bootstrapping*-Vorgang, dass jetzt jeder Befehl der Shell im Kontext des aktuell angemeldeten Nutzers ausgeführt wird. Vorher war der Subject, der über den `AccessControlContext` abgefragt wurde, immer null. Nach den Änderungen ist es jetzt der aktuelle User.

7.2.5 Auftretende Probleme

Während der Implementierung sind verschiedene Probleme aufgetreten, die auf Grund der kurz bemessenen Zeit für die Bachelor-Thesis nicht alle bearbeitet werden konnten. Die Probleme, die direkt Einfluss auf den *Bootstrapper* hatten, sind hier aufgeführt.

7.2.5.1 Problem mit JAR-Dateien

In diesem Abschnitt wird kurz beschrieben wie eine JAR-Datei erstellt wird, wie diese signiert wird und warum es zu Problemen mit signierten JAR-Dateien kommen kann. JAR-Dateien sind ZIP-Dateien änlich und stellen ein Archiv von Dateien dar. Der in Listing 7.1 gezeigte Befehl erzeugt eine JAR-Datei mit dem Namen `SicariBootstrapper.jar` und speichert alle Dateien des aktuellen und aller darunterliegenden Verzeichnisse. Bei der Erstellung von JAR-Dateien wird automatisch ein sogenanntes `Manifest` erzeugt. Diese `Manifest`-Datei kann Informationen über die gespeicherten Dateien beinhalten. In Listing 7.2 ist solch eine Datei beispielhaft gezeigt. Zum Signieren der Datei wird ein *Keystore* und ein *Alias* benötigt. In dem *Keystore* ist der private Schlüssel der signierenden Person enthalten. Java bietet mit dem `jarsigner`-Programm ein *Tool* zum Signieren von JAR-Dateien. Wie ein eigener Keystore erzeugt wird und wie die JAR-Datei dann signiert wird, ist in Listing 7.3 zu sehen.

Listing 7.1

```
jar -cf SicariBootstrapper.jar *
```

Listing 7.2

```
Manifest-Version: 1.0
Created-By: 1.5.0_06 (Sun Microsystems Inc.)
Main-Class: Test
Name: Test.class
SHA1-Digest: ajp3Y3eYtj53OokReNMIgRwydFY=
```

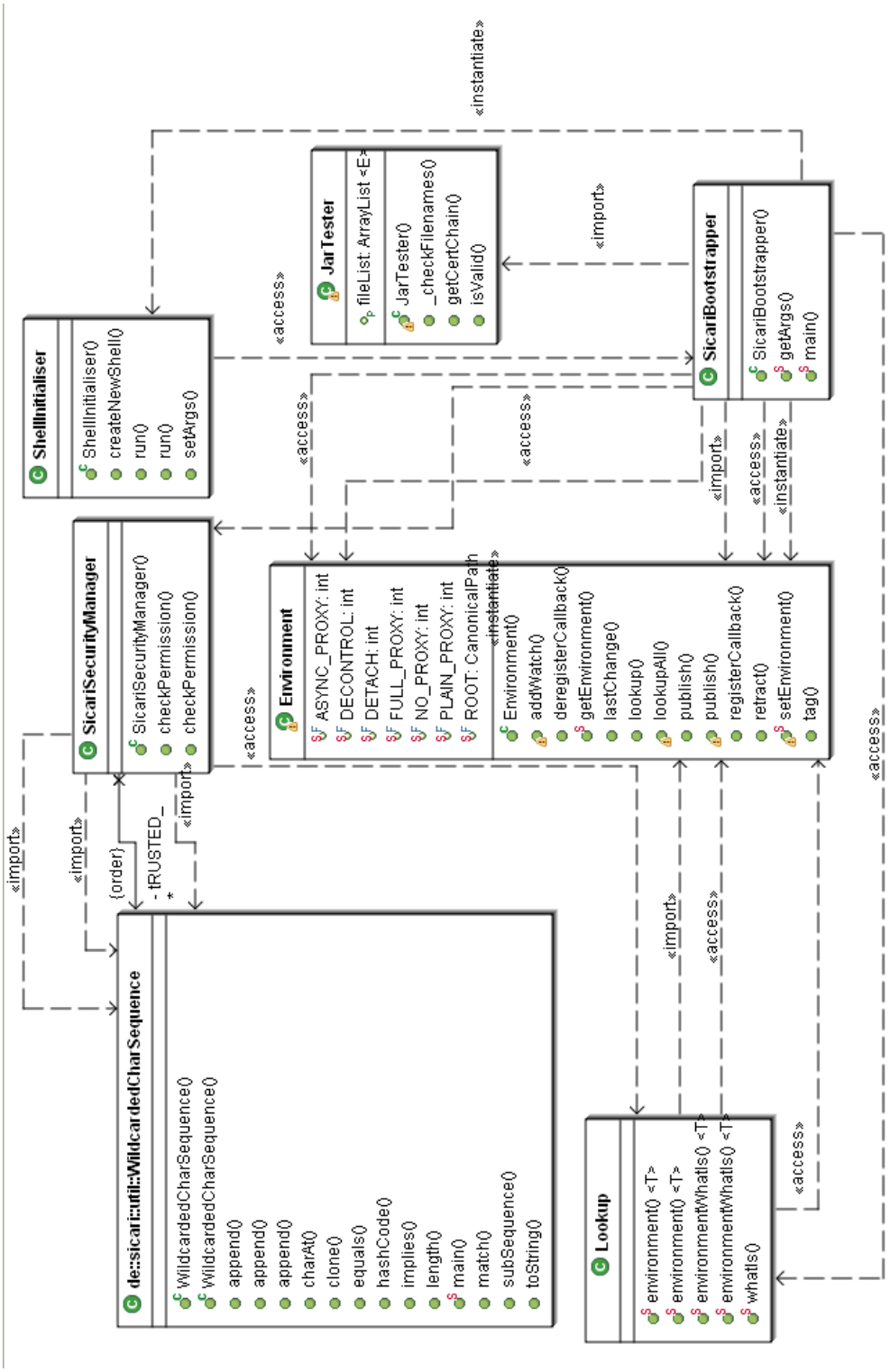


Abbildung 7.3: Klassendiagramm Bootstrapping (neu)

Listing 7.3

```
Key generieren:
C:\>keytool -genkey -alias ReneeB
Geben Sie das Keystore-Passwort ein: passwort
Wie lautet Ihr Vor- und Nachname?
[Unknown]: Renee
Wie lautet der Name Ihrer organisatorischen Einheit?
[Unknown]: Smart-Websolutions
Wie lautet der Name Ihrer Organisation?
[Unknown]: Perl-Programmierung
Wie lautet der Name Ihrer Stadt oder Gemeinde?
[Unknown]: Riedstadt
Wie lautet der Name Ihres Bundeslandes oder Ihrer Provinz?
[Unknown]: Hessen
Wie lautet der Landescode (zwei Buchstaben) für diese Einheit?
[Unknown]: DE
Ist CN=Renee, OU=Smart-Websolutions, O=Perl-Programmierung,
L=Riedstadt, ST=Hessen, C=DE richtig?
[Nein]: Ja
Geben Sie das Passwort für <ReneeB> ein.
(EINGABETASTE, wenn Passwort dasselbe wie für Keystore):
JAR signieren:
C:\>jarsigner SicariBootstrapper.jar ReneeB
Enter Passphrase for keystore: passwort
Warning: The signer certificate will expire within six months.
```

Es existiert ein Bugreport[1], in dem beschrieben wird, dass ein JAR-Archiv verifiziert wird, obwohl unsignierte Inhalte in der Datei enthalten sind. Um dies zu testen, wurde ein Beispiel-Archiv erstellt und signiert. Mit WinRAR wurde eine weitere Klasse in das Archiv gepackt und mit dem jarsigner überprüft (siehe Listing **7.**4). Dieser hat zwar eine Warnung ausgegeben, aber das Archiv wurde dennoch als verifiziert erkannt. Mit den JCE-Mitteln wird dieser Fehler gar nicht erkannt. Aus diesem Grund wurde die Klasse `JarTester` geschrieben und um entsprechende Prüfroutinen erweitert.

Listing 7.4

```
C:\>jarsigner -verify Test.jar
jar verified.
Warning: This jar contains unsigned entries which
have not been integrity-checked. Re-run with the
-verbose option for more details.
This jar contains entries whose signer certificate
will expire within six month. Re-run with the -verbose
and -certs options for more details.
```

[1] https://bugs.eclipse.org/bugs/show_bug.cgi?id=83349

Das gleiche Archiv mit dem JarTester überprüft, gibt die Meldung aus, dass die Datei nicht valide ist (siehe Listing **7.**5)

Listing 7.5

```
C:\>java Test.jar
4: Test2.class
The intgerity of the JAR file is not warranted
```

7.2.5.2 trusted.classes

Im ersten Entwurf des neuen Bootstrapping Vorgangs war es vorgesehen, dass die Datei mit den vertrauenswürdigen Klassen - `trusted.classes` - komplett wegfällt um Angreifern keine Chance haben, eine kompromittierte Klasse als vertrauenswürdig zu kennzeichnen. Die Idee war es, nur über den JarTester und die Zertifikate zu bestimmen welche Klassen vertrauenswürdig sind. Diese Idee musste verworfen werden, weil Bibliotheken wie `rt.jar` - in der die Standard-Bibliotheken von Java gespeichert sind - nicht signiert sind.
In der jetzigen Fassung, gibt es die `trusted.classes` noch. Da diese Datei aber in der signierten JAR-Datei von SicAri ist, wird diese Datei und damit auch der Inhalt als vertrauenswürdig angesehen.

7.2.5.3 Klassen laden

Zum Test wurde schon sehr früh während der Implementierung ein Beispiel JAR-Archiv erstellt, das die Klassen enthielt, die für das *Bootstrapping* notwendig waren. Alle Zusatzdienste und andere Klassen wurden in ein andere JAR-Archiv gespeichert. Bei einem Test zeigte sich schnell, dass dies zu Problemen führte, da keine Klasse außerhalb des *Bootstrapper*-JARs geladen werden konnten. Das lag daran, dass die Klassen aus dem zweiten JAR-Archiv nicht im *CLASSPATH* des *Bootstrappers* lagen. Und durch das *class loader*-Konzept von Java konnten die Klassen nicht geladen werden. Aus diesem Grund wurden alle bisher existierenden Klassen von SicAri in das *Bootstrapper*-JAR-Archiv gespeichert.

Kapitel 8

Zusammenfassung und Ausblick

In diesem abschließenden Kapitel sollen die Schwerpunkte und die erreichten Ziele dieser Arbeit zusammengefasst werden. Dazu werden die Zielsetzungen aus dem einleitenden Kapitel mit den Ergebnissen verglichen. Im zweiten Abschnitt dieses Kapitels wird ein Ausblick auf mögliche zukünftige Entwicklungen gegeben.

8.1 Zusammenfassung

In Abschnitt 1.1 wurden die Ziele und die Motivation dieser Arbeit genannt. Als Zieldefinitionen wurden folgende Punkte benannt:

- Literaturrecherche
- Entwickeln von Angriffsszenarien
- Implementierung neuer Sicherheitsmechanismen

In Kapitel 2 wurde eine allgemeine Einführung in die IT-Sicherheit gegeben. Da neue Schutzmechanismen in SicAri implementiert werden sollen, ist ein Grundverständnis für die IT-Sicherheit sehr wichtig. Deshalb wurden allgemeine Ziele von IT-Sicherheit erläutert.
Welche Sicherheitsmechanismen es in der Programmiersprache Java gibt, wurde in Kapitel 3 erläutert. Die Einführung in diese Mechanismen ist nötig, da in dem Projekt Java als Programmiersprache verwendet wurde und SicAri einige der Sicherheitsmechanismen von Java verwendet.
Um einen Ansatzpunkt für Verbesserungen an einer Anwendung zu bekommen, ist es nötig eine Bedrohungs- und Angriffsanalyse zu machen. Die theoretischen Grundlagen dazu wurden in Kapitel 4 genannt. In Kapitel 5 wurde die Anwendung, in deren Rahmen diese Arbeit erstellt wird, - SicAri - vorgestellt.
Im anschließenden Kapitel wurde für den Teilbereich „*Bootstrapping*“ eine Bedrohungs- und Angriffsanalyse durchgeführt. Im Kapitel 7 wurde dann aufgezeigt, welche Änderungen an SicAri vorgenommen wurden.

8.2 Ausblick

Während der Implementierung sind einige Probleme aufgetreten, die es in Zukunft zu bearbeiten gilt. Durch das *class loader*-Prinzip von Java (vergleiche Abschnitt 3.2.1.2) ist es nicht möglich,

Klassen zu laden die außerhalb des initialen JAR-Archivs liegen. Durch dieses Problem ist es Nutzern nicht möglich, eigene Dienste zu starten. Da dies jedoch ein zentraler Bestandteil von SicAri ist, ist hier der Bedarf an einer Änderung des *Class Loader*-Prinzips sehr hoch.
Ein weiterer Punkt für die zukünftige Entwicklung ist die Problematik mit *Graphical User Interface (GUI)*-Anwendungen. Dieses Problem (vergleiche Abschnitt 6.2) ist ein natives Java-Problem. Hier sollte ein Konzept erarbeitet werden, wie man die Aufrufe in der *Event-Queue* in mit dem Kontext des Aufrufenden ausführen kann.

Anhang A

Akronyme

3DES	Triple Data Encryption Standard
AES	Advanced Encryption Standard
API	Applications Programming Interface
AWT	Abstract Window Toolkit
BSI	Bundesamt für Sicherheit in der Informationstechnik
CA	Certification Authority
CLDC	Connected Limited Device Configuration
CPU	Central Processing Unit
CSI	Computer Security Institue
CSP	Cryptography Service Provider
DES	Data Encryption Standard
DNA	Desoxyribonucleic Acid
DoS	Denial-of-Service
DREAD	Damage-Reproducibility-Exploitability-Affected User-Discoverability
EIS	Enterprise Information System
EJB	Enterprise Java Beans
GPS	Global Positioning System
GSS-API	Generic Secure Services API
GUI	Graphical User Interface
HTTP	Hypertext Transport Protocol
ID	Identifier
IMAP	Internet Message Access Protocol
IP	Internet Protocol
IT	Informationstechnik
J2EE	Java 2 Enterprise Edition

J2EE CA	J2EE Connector Achitecture
J2ME	Java 2 Micro Edition
J2SE	Java 2 Standard Edition
JAAS	Java Authentication and Authorization Services
JAR	Java ARchive
JCA	Java Cryptography Architecture
JCE	Java Cryptography Extension
JDBC	Java Database Connectivity
JDK	Java Development Kit
JGSS	Java Generic Secure Services
JSP	Java Server Pages
JSR	Java Specification Request
JVM	Java Virtual Machine
KVM	„Kilo Virtual Machine“
LDAP	Lightweight Directory Access Protocol
MAC	Message Authentication Code
MIDP	Mobile Information Device Profile
OSI	Open Systems Interconnection
OSGi	Open Services Gateway Initiative
PAM	Pluggable Authentication Module
PDA	Personal Digital Assistants
PHP	PHP Hypertext Preprocessor
PIN	Personal Identification Number
PKI	Public-Key-Infrastructure
RBAC	Role Based Access Control
RSA	Rivest-Shamir-Adleman
SASL	Simple Authentication and Security Layer
SMTP	Simple Mail Transport Protocol
SQL	Structured Query Language
SSH	Secure Shell
SSL	Secure Socket Layer
SSO	Single-Sign-On
TLS	Transport Layer Security
USB	Universal Serial Bus
XML	eXtensible Markup Language
XACML	eXtensible Access Control Markup Language

Literaturverzeichnis

[1] P. Karlton und P.Kocher A. Freier. The ssl protocol version 3.0. *http://home.netscape.com/eng/ssl3/index.html*, 1996.

[2] Paul C. van Oorschot Alfred J. Menezes and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[3] Albrecht Beutelspacher. *Kryptologie*. vieweg, 2 edition, 2005. ISBN 3-8348-0014-7.

[4] Ciarán Bryce. Isolates: A new approach to multi-programming in java platforms. *OT Land*, 05.04, 2004.

[5] Gary Cornell Cay S. Horstmann. *Core Java 2 Volume I - Fundamentals*. Prentice Hall, 2005. ISBN 0-13-148202-5.

[6] Larry Koved Anthony Nadalin Roland Schemers Charlie Lai, Li Gong. User authentication and authorization in the java platform. 1999.

[7] Ray Lai Christopher Steel, Ramesh Nagappan. *Core Security Patterns*. Pearson - Prentice Hall, 2006.

[8] SicAri Consortium. The sicari code conventions. Technical report, 2004.

[9] T. Dierks. The transport layer security (tls) protocol version 1.1. *RFC4346 http://www.ietf.org/rfc/rfc4346.txt*, 2006.

[10] Claudia Eckert. *IT-Sicherheit*. Oldenbourg, München, 2005. ISBN 3-486-57676-3.

[11] David F. Ferraiolo, D. Richard Kuhn, and Ramaswamy Chandramouli. *Role-Based Access Control*. Artech House Publishers, 2003.

[12] Li Gong. *Inside Java 2 Platform Security*. Addison-Wesley, 1999.

[13] Sven Haiges. Residential gateway. *Javamagazin*, 07.2005:72 – 77, 2005.

[14] Stuart Halloway. Introduction to jaas and using jaas. *http://java.sun.com/developer/JDCTechTips/2001/tt0727.html*, 2001.

[15] CRYPTAS it Security GmbH AUSTRIA. http://www.cryptoshop.com/de/knowledgebase/cryptograph (12.06.2006).

[16] Roland Schemers Li Gong. Implementing protection domains in the java development kit 1.2. Technical report, Sun Microsystems, 1997.

[17] Mario Linke. Das osgi-framework. *JAVA Spektrum*, 02.2005:35 – 38, 2005.

[18] Qusay H. Mahmoud. Java authentication and authorization service (jaas)in java 2, standard edition (j2se) 1.4. *http://java.sun.com/developer/technicalArticles/Security/jaasv2/index.html*, 2003.

[19] Alison Huml Mary Campione, Kathy Walrath. *The Java Tutorial Continued*. Sun Microsystems, 1999.

[20] Jan-Bernd Meyer. Die bedrohung kommt von innen. *Computerwoche*, 22:6–7, 2006.

[21] Microsoft. Bedrohungsanalyse. 2004.

[22] Chamseddine Talhi und Sami Zhioua Mourad Dabbabi, Mohamed Saleh. Java for mobile devices: A security study. Technical report, Concordia Institue for Information Systems Engineering, Concordia Universität, Montreal, Kanada, 2004.

[23] J. Myers. Simple authentication and security layer (sasl). *http://www.ietf.org/rfc/rfc2222.txt*, 1:1, 1997.

[24] Dimo Milev Nikolaj Cholakov. The evolution of the java security model. *International Conference on Computer Systems and Technologies - CompSysTech' 2005*, 2005.

[25] Jan Oetting, Jan Peters, Ulrich Pinsdorf, Taufiq Rochaeli, and Ruben Wolf. Specification of the sicari architecture. 2004.

[26] Stephan Wolthusen Isabel Münch Hubertus Gottschalk Sebastian Hummel Olaf Jüptner, Christoph Busch. Hessisches ministerium für wirtschaft, verkehr und landesentwicklung. *IT-Sicherheit für den Mittelstand*, 38:1–30, 2002.

[27] OSGi Alliance. *About the OSGi Service Platform*, 3.0 edition, July 2004.

[28] W. Ford D. Solo R. Housley, W. Polk. Internet x.509 public key infrastructure. *http://www.ietf.org/rfc/rfc3280.txt*, 2002.

[29] M. Repges. Einführung in ssl. http://www.repges.net/SSL/ssl.html.

[30] Robert Richardson. Csi/fbi - computer crime and security survey 2003. 2003.

[31] Solidium. Public key infrastructure (pki). *http://www.solidium.nl/pages/publicaties/White paper PKI v01.5.pdf*, 1:1–7, 2003.

[32] Sun. Java security overview. Technical report, Sun Microsystems, April 2005.

[33] Charlie Lai Vipin Samar. Making login services independent of authentication technologies. *http://java.sun.com/security/jaas/doc/pam.html*, 1996.

[34] Tanja Wolff. Ignorante mitarbeiter gefährden it-sicherheit. *CIO*, 05.01.2006, 2006.

[35] Tanja Wolff. Sicherheitslücke usb-stick. *CIO*, 22.06.2006, 2006.