Technische Universität Darmstadt
Fachbereich Informatik
Fachgebiet Theoretische Informatik

Optimierung des Flexsecures Online Certificate Status Protocol

Diplomarbeit von:

Younes Bennani

Betreuer:
Prof. Dr. J. Buchmann
M. Lippert

9. Juli 2007

Ehrenwörtliche Erklärung

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

Darmstadt, Juli 2007

Younes Bennani

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation . . . 1

1.2 Zielsetzung der Arbeit . . . 2

1.3 Aufbau der Arbeit . . . 2

2 Grundlagen **5**

2.1 Public Key Kryptographie . . . 5

2.1.1 Public Key Verschlüsselung . . . 6

2.1.2 Digitale Signatur . . . 7

2.2 Public Key Infrastruktur . . . 7

2.2.1 Aufbau einer PKI . . . 8

2.2.2 Zertifikat . . . 9

2.2.3 Gültigkeitsmodelle . . . 12

2.2.4 Certificate Revocation List . . . 13

2.3 Cache . . . 15

2.3.1 Organisation und Aufbau eines Caches . . . 16

2.3.2 Ersetzungsstrategie . . . 16

3 Online Zertifikat Revozierung **19**

3.1 Online Certificate Status Protocol . . . 19

3.1.1 Aufbau eines OCSP-Server . . . 20

3.1.2 Einzelheiten des OCSP-Protokolls . . . 21

3.1.3 Performanz- und Sicherheitsüberlegungen . . . 27

3.2 Lightweight-OCSP . . . 28

3.2.1 Das Verhalten in einem Lightweight-OCSP . . . 29

3.2.2 Sicherheitsüberlegungen . . . 31

3.3 Simple Certificate Validition Protocol . . . 32

3.3.1 Einführung in das SCVP Protokoll . . . 32

3.3.2 Protokollablauf . . . 33

4 Design und Entwurf **35**

4.1 Allgemeine Architektur . . . 35

4.2 Interne Datenstrukturen . . . 36

4.2.1 CRL Cache . . . 37

4.2.2 Issuer Cache . . . 38

4.2.3 Certificate Cache . . . 40
4.2.4 SignedResponse Cache . . . 42
4.2.5 Ersetzungsstrategie . . . 44
4.3 Interne Abläufe . . . 45
4.3.1 Verarbeitung eines OCSP-Requests . . . 45
4.3.2 Triggern . . . 49

5 Implementierung und Evaluierung **51**
5.1 Implementierung . . . 51
5.1.1 Realisierung des Cache-Konzepts . . . 51
5.1.2 Bearbeitung eines Requests . . . 55
5.2 Evaluierung . . . 58
5.2.1 Testsdurchführung . . . 58
5.2.2 Analysierung der Testergebnisse . . . 62
5.3 Installation und Konfiguration . . . 63
5.3.1 ocsp.properties . . . 63
5.3.2 scheduler-service.xml . . . 65
5.3.3 server.xml . . . 66
5.3.4 standardjboss.xml . . . 67
5.3.5 Installation der Testumgebung . . . 68

6 Ausblick **69**

Literaturverzeichnis **72**

Kapitel 1

Einleitung

1.1 Motivation

Durch das Internet ist das online einkaufen selbstverständlich geworden und der elektronische Handel wächst enorm. Der Anzahl der elektronischen Transaktionen steigen und das E-Commerce und das Online-Banking ist für jedermann selbstverständlich geworden. Fast jeder Haushalt verfügt über einen Internetanschluss. Zurzeit werden weitere Konzepte entwickelt und realisiert, beispielsweise die elektronische Gesundheitskarte und die E-Gouvernement.

Die Public Key Infrastruktur (PKI) ermöglicht es mit ihren Mechanismen und Verfahren die Transaktionen auf einem sicheren Niveau zu betreiben, um Betrugsmöglichkeiten zu verhindern. Die PKI bietet für jeden Anwender unter anderem ein Zertifikat in der elektronischen Welt, als ein Art persönlicher Ausweis, in dem eine digitale Signatur, den jeweiligen Anwender zuzuordnen ist.
Da ein Zertifikat kompromotiert oder missbraucht werden kann, bietet die PKI ihren Anwendern die Möglichkeit die Zertifikate auf ihre Gültigkeit zu überprüfen. Bei Ungültigkeit des Zertifikates wird es revoziert, d.h. der Anwender wird über die Vertrauenswürdigkeit des Zertifikates informiert. Die Revokation eines Zertifikates ist ein sehr wichtiger Aspekt für die Glaubwürdigkeit des PKI Anbieters und für die Sicherheit des Anwenders. Die PKI bietet ihren Anwendern verschiedene Methoden um ein Zertifikat auf seine Gültigkeit zu überprüfen. Es wird grundsätzlich eine der folgenden Verfahren angewandt oder miteinander kombiniert:

- Certificate Revocation List (CRL)
- Online Certificate Status Protocol (OCSP)
- Simple Cerificate Validation Protocol (SCVP)

Die Verwaltung der Revokationsinformationen ist eine der Hauptaufgaben die eine PKI durchführt. Das OCSP hat sich als Standard durchgesetzt, jedoch stellt die Entwicklung von immer breiteren PKIs Umgebungen ein Problem dar. Es müssen immer neue Lösungen entwickelt und optimiert werden, die die Kosten, Schnelligkeit und Skalierbarkeit betreffen.

1.2 Zielsetzung der Arbeit

Das Ziel der Diplomarbeit ist die Erweiterung und Weiterentwicklung des von der Firma Flexsecure [flexsec] implementierten OCSP-Responders. Es sollen neue Ideen entworfen und implementiert werden, die zur Verbesserung der Skalierbarkeit und der Performanz des OCSP-Responders beitragen. Die Sicherheitaspekte und Richtlinien des OCSPs werden durch die Änderungen nicht beeinträchtigt. Es wird hauptsächlich einen bzw. mehrere Caches realisiert, die die folgenden Vorteile zum OCSP mit sich bringen:

- Schnelle Verarbeitung der OCSP-Anfragen (Requests)
- Beantwortung von mehr Anfragen pro Zeit Einheit als bis jetzt der Fall ist
- Verwaltung des Speicherplatzes des OCSP-Servers

Im Rahmen dieser Arbeit soll das Verhalten des OCSP Responders geändert werden. Beim Starten des OCSP-Server werden alle im MasterLDAP gespeicherten Zertifikate zum Speicher des OCSPs geholt und im Certificatescache gespeichert. Nach der Änderung werden nicht mehr alle Zertifikate zum Hauptspeicher geholt, sondern nur einzelne im Verlauf der Beantwortung der Anfragen. Eine wichtige Rolle spielt die Ersetzungsstrategie und die vorher definierte Größe des Caches. Zudem wird ein Art *Pre-Produced* Cache implementiert, der die bereits beantwortete Anfragen speichert, um diese gegebenenfalls wieder zu verwenden. Es wird hierbei nicht die Antwort im voraus berechnet, sondern auch im Verlauf der Zeit wird der Cache aufgefüllt.

1.3 Aufbau der Arbeit

Die vorliegende Diplomarbeit ist wie folgt aufgebaut:

Im Kapitel 2, *Grundlagen*, wird zunächst Public Key Kryptographie und Public Key Infrastrukturen erklärt. Es werden die für diese Arbeit nötigen Begriffe und Themen wie digitale Signatur, Zertifikat, Gültigkeitsmodelle und Certificate Revocation List (CRL) behandelt. Am Ende des Kapitels wird der Aufbau und die Ersetzungsstrategien des Caches erklärt.

Das Kapitel 3, *Online Zertifikat Revozierung*, widmet sich den Online Revokation Protokolle. Es werden drei Verfahren vorgestellt und diskutiert, Online Certificate Status Protocol (OCSP), Lightweight OCSP und Simple Certificate Validition Protocol (SCVP).

Das Kapitel 4, *Design und Entwurf*, beschreibt das Design des vorhandenen OCSP-Servers sowie die getroffene Entscheidungen zur Optimierung und die Verbesserung des Performanz des OCSPs. Es werden die internen Datenstrukturen sowie die internen Abläufe erläutert.

Im Kapitel 5, *Implementierung und Evaluierung* werden die Implementierung des Desingsentwurfs sowie die Beschreibung und die Durchführung der Testfälle zur Evaluierung des OCSPs sowie die Installation und Konfiguration des Softwares beschrieben.

Im Kapitel 6, *Ausblick*, wird am Schluss dieser Arbeit das Wesentliche zusammengefasst.

In dieser Ausarbeitung werden wichtige Aspekte oder Bezeichner kursiv hervorgehoben, Literaturverweise werden in eckigen Klammern "[...]" aufgeführt. Am Ende dieser Ausarbeitung ist das Literaturverzeichnis zu finden.

Kapitel 2

Grundlagen

Dieses Kapitel, behandelt das notwendige Grundwissen für diese Ausarbeitung. Es wird auf die Public Key Kryptographie, Public Key Infrastruktur (PKI), die Zertifikate und ihre Anwendungen sowie die Revokationsmethoden eingegangen. Am Ende des Kapitels werden die Grundlagen des Caches kurz erläutert. Es wird keine detaillierte Einführung durchgeführt, besonders in der mathematischen Grundlagen und Algorithmen, da der Kern dieser Diplomarbeit eine praktische Anwendung behandelt, und zwar die Optimierung eines Online Certificate Status Protocol (OCSP), der im Kapitel 3 ausführlich erläutert wird.

2.1 Public Key Kryptographie

Schon die alten Griechen, Ägypter oder Römer haben alle ihre Botschaften verschlüsselt, um die Nachricht gegenüber Dritte unsichtbar und vertraulich zu halten. Damals wie heute spielt die Kryptographie eine immense Rolle für die Kommunikation zwischen den Kommunikationspartnern. Da die heutige Informationsgesellschaft, die elektronische Kommunikation in vielfältiger Weise nutzt, beispielsweise bei Homebanking, E-Mail, E-Commerce etc., ermöglicht die moderne Kryptographie im Gegensatz zur klassischen, nicht nur die Vertraulichkeit was unter Verschlüsselung bekannt ist, sondern auch andere Schutzziele um die Anforderungen des Anwenders zu realisieren. Diese Schutzziele werden in [JohEi] wie folgt definiert:

Vertraulichkeit garantiert, dass nur Berechtigte Zugang zu den Informationen haben. Sie wird mit Hilfe von Verschlüsselungsverfahren gewährleistet.

Authentizität bezeichnet die Echtheit und Glaubwürdigkeit elektronischer Nachrichten. Kryptographische Techniken geben Aufschluß über die Identität des Absenders und garantieren damit die Authentizität der Nachricht.

Integrität der Daten wird gewährleistet, wenn es unmöglich ist, diese unbemerkt zu manipulieren. Mit kryptographischen Methoden kann überprüft werden, ob die Daten verändert wurden.

Zurechenbarkeit elektronischer Dokumente bedeutet, dass es einem Dritten gegenüber nachgewiesen werden kann, das ein Dokument von einem bestimmten Absender kommt. Dafür verwendet man digitale Signaturen.

2.1.1 Public Key Verschlüsselung

Durch die Vertraulichkeit wird sicher gestellt, dass kein unerwünschter Dritter die Nachricht abhören kann. Dies wird entweder durch: die symmetrische, asymmetrische oder hybride Verschlüsselungsverfahren (mehr dazu in [JohEi]) erreicht.

Das symmetrische Verfahren ist die älteste Verschlüsselungsmethode, sie basiert auf einen gemeinsamen geheimen Schlüssel, d.h. Alice und Bob benutzen in ihrer Kommunikation den gleichen Schlüssel, zur Ver- und Entschlüsselung ihrer geheimen Nachrichten. Der Nachteil dieses Verfahrens ist, dass der Schlüssel auf einem sicheren Weg dem Kommunikationspartner zugestellt werden muss. Ein weiteres Problem stellt die Anzahl der Schlüssel dar, da für jeden Kommunikationspartner ein eigener Schlüssel existiert. Zu den symmetrischen Verschlüsselungsverfahren gehören: DES/ 3DES, IDEA und AES.

Das asymmetrische Verfahren wird auch Public Key Verfahren genannt. In der Public Key Verschlüsselung besitzt jeder Teilnehmer ein Schlüsselpaar, d.h. zwei Schlüssel, die eine gemeinsame mathematische Basis (z.B. Primzahl mit 200 Dezimalstellen) haben. Einer dieser Schlüssel ist der öffentliche Schlüssel (Public Key), dieser ist für alle Teilnehmer bekannt - daher öffentlich-. Der andere ist der private Schlüssel, dieser ist nur dem Eigentümer bekannt und muß unter allen Umständen geheimgehalten bleiben. Wobei der private Schlüssel aus dem öffentlichen Schlüssel nicht effizient berechenbar ist. Wenn Alice eine verschlüsselte Nachricht an Bob schicken will, nimmt sie seinen Public Key und verschlüsselt damit die Nachricht. Um den Klartext zu bekommen wendet Bob auf den Chiffretext seinen privaten Schlüssel an. So kann nur Bob die Nachricht lesen. Ein Vertreter der asymmetrischen Verschlüsselung Algorithmen ist der RSA Algorithmus ([PKCS1]) (benannt nach seinen Entwicklern 1977, Ron Rivest, Adi Shamir, Leonard Adleman).
Im Gegensatz zum symmetrischen Verfahren, wird neben der Verschlüsselung auch die Zurechenbarkeit mit den asymmetrischen Verfahren erreicht. Der Nachteil des asymmetrischen Verfahrens besteht darin, dass es deutlich langsamer ist, als die symmetrische Verschlüsselung (RSA ist ca. 1000 Mal langsamer, als z.B. DES) und aufgrund der mathematischen Abhängigkeiten der beiden Schlüssel werden auch höhere Schlüssellängen benötigt. Aus diesem Grund wird in der Praxis im Allgemeinen nur die hybride Verschlüsselung eingesetzt.

Das hybride Verfahren ist eine Kombination aus symmetrischer und asymmetrischer Verschlüsselung. Dabei wird die zu verschlüsselnder Nachricht durch Alice zunächst mit einem geheimen Schlüssel (Session Key) symmetrisch verschlüsselt. Anschließend wird der Session Key selbst mit dem öffentlichen Schlüssel von Bob asymmetrisch verschlüsselt und übertragen. Bob entschlüsselt nun asymmetrisch mit Hilfe seinem privaten Schlüssel den Session Key und schließlich symmetrisch mit dem Session Key die eigentliche Nachricht. Da nur der symmetrische Schlüssel verschlüsselt wird, bleibt der Rechenaufwand bei der asymmetrischen Verschlüsselung relativ gering.

Es bleibt noch zu wissen:

- woher Alice sicher gehen kann, dass der benutzte Public Key wirklich vom Bob ist und nicht von einen dritten, der sich als Bob ausgibt?
- Woher bekommt Alice den öffentlichen Schlüssel von Bob?

- Wie findet Alice heraus, ob der öffentliche Schlüssel gültig ist:
  - Wenn es sich um einen Signaturschlüssel handelt, ist er gültig zum Zeitpunkt der Signatur und nicht kompromittiert?
  - Wenn es sich um einen Verschlüsselungs- oder Authentifikationsschlüssel handelt, ist der jetzt gültig?

Im Kapitel 2.2 gehe ich darauf ein, wie diese Probleme in der Public Key Infrastructure gelöst werden.

2.1.2 Digitale Signatur

Die Zurechenbarkeit und Authentizität werden mit Hilfe der digitalen Signatur (elektronische Signatur nach der EU Richtlinien) gewährleistet. Diese ersetzen die Funktionalität der handlichen Unterschrift in der digitalen Welt, d.h. wenn Alice eine Nachricht *digital signiert*, kann sie nicht im nachhinein bestreiten, dass diese Nachricht von ihr stammt und dass sie, sie signiert hat. Die Integrität der Nachricht wird auch mit der digitalen Signatur gewährleistet, in dem der Fingerabdruck der Nachricht signiert wird. Der digitale Fingerabdruck (Message Digest) wird mit Hilfe einer Einweg Hash Funktion ([JohEi]) aus der Nachricht berechnet. Analog zur Verschlüsselung kann man jeden deterministischen umkehrbaren und sicher eingestuften Public Key Algorithmus für den Zweck der Signatur konstruieren. Wobei ein symmetrisches Verfahren hier nicht geeignet ist, da zwei Teilnehmer im Besitz des geheimen Schlüssels sind und beide damit Signaturen erstellen können. Der Verlauf der Signatur ist dann anders, als der der Verschlüsselung. Will Alice eine Nachricht digital signieren, so wendet sie ihren privaten Schlüssel auf die Nachricht bzw. den Fingerabdruck und erhält damit die digitale Signatur der Nachricht. Als nächste überträgt sie die Nachricht und die Signatur an Bob. Will Bob sicher gehen, dass die Nachricht von Alice ist, so muß er die Nachricht bzw. den Fingerabdruck mit dem für ihn bekannte Public Key von Alice verschlüsseln und das Ergebnis mit der Signatur von Alice vergleichen. Stimmen beide überein, dann ist die Nachricht tatsächlich von Alice, andernfalls stellt er fest, dass es um ein Täuschungsmanöver handelt. Hier tritt auch das Problem der Zugehörigkeit des öffentlichen Schlüssels zu seinem Besitzer auf. Mehr dazu in den Abschnitten 2.2 und 2.2.2.

2.2 Public Key Infrastruktur

Ein Angreifer kann Bob seinen öffentlichen Schlüssel unterschieben und behaupten der Schlüssel sei von Alice. Daher reicht ein sicheres Public Key Verfahren allein, für das Erstellen einer digitalen Signatur, die die Zurechenbarkeit und die Authentizität einer elektronischen Datei sicherstellen muss, oder für die Verschlüsselung, die die Vertraulichkeit gewährleisten soll (siehe Kapitel 2.1) nicht aus. Um dieses Problem zu lösen ist eine Infrastruktur erforderlich, die das Schlüsselmanagement organisiert und regelt.

Die Public Key Infrastruktur ([pki]) ist ein System das sicherstellt, dass die privaten Schlüssel geheim bleiben und die öffentlichen Schlüssel vor Mißbrauch und Fälschung geschützt werden. Die PKI stellt die fehlende Sicherheit für die praktische Nutzung kryptographische Dienste sicher.

Eine PKI agiert als eine vertrauenswürdige Instanz, die die folgenden Aufgaben seinen Teilnehmern bietet:

- Erzeugung und Speicherung von Schlüsselpaaren (Public Key, Private Key)
- Zweifelsfreie Zuordnung einer Identität zu einem öffentlichen Schlüssel
- Verteilung der öffentlichen Schlüssel
- Schutz der privaten Schlüssel
- Außerbetriebnahme und Vernichtung von Schlüsselpaaren
- Archivierung und Wiederherstellung von Schlüsselpaaren

2.2.1 Aufbau einer PKI

Die PKI als vertrauenswürdige Instanz ist eine Zentrale Komponente. Bekannt als Zertifizierungsstelle, stellt sie u.a. Zertifikate aus und signiert sie. Sie wird von einem PKI-Anbieter betrieben und unterliegt zahlreichen Schutzmaßnahmen.
Diese Zertifizierungsstelle oder Trust Center faßt sehr umfangreiche und komplizierte Prozesse mit sehr hohem Sicherheitsgrad zusammen. Um dies in der Praxis zu realisieren, wird dieser Trust Center in mehreren Komponenten aufgeteilt. Dem entsprechend besteht eine PKI aus den folgenden Komponenten:

Registrierungsinstanz (Registration Authority – RA)

Will sich ein neuer Teilnehmer ein Zertifikat (2.2.2) besorgen, so wendet er sich an der Registrierungsinstanz mit seinen Personalien z.B. Name, Kontaktadresse etc. an. Die RA überprüft dann die Benutzerdaten auf ihre Gültigkeit. Dafür verwendet sie verschiedene Methoden, wie z.B. das persönliche Erscheinen des Antragstellers. Ist die Datenüberprüfung erfolgreich verlaufen, so sorgt die RA für die sichere Übermittlung der benötigen Daten für die Erstellung des Zertifikates an die Zertifizierungsinstanz (CA), dies geschieht meist in Form eines PKCS#10-Antrags [PKCS10] an die CA. Nach der erfolgreichen Erstellung des Zertifikates erhält der neue Teilnehmer von der RA sein Zertifikat und seinen privaten Schlüssel.
Die Verlängerung oder Erhaltung eines Passworts für die Sperrung des vorhandenen Zertifikates gehören auch zur Aufgaben der RA.
Um die Vertraulichkeit, Zurechenbarkeit und Authentizität zu gewährleisten werden der private Schlüssel und die kryptographische Verfahren in einer *Persönlichen Sicherheitsumgebung (personal security enviroment PSE)* aufbewahrt. Die PSE-Umgebung kann als Softwarelösung gemäß dem PKCS#12 Standard[PKCS12] realisiert werden, diese wird auch als *softtoken* bezeichnet. Eine Alternative zur Softwarelösung ist eine Hardware Realisierung oder auch *hardtoken* genannt. Die Hardtoken werden meistens als *Hardware security modules* [hsm] oder *Chipkarten* [chipka] realisiert. Die Hardtoken haben den Vorteil, dass sie mehr Sicherheit bieten als die Softoken, dafür sind sie teurer.

Zertifizierungsinstanz (Certification Authority – CA)

Die Zertifizierungsinstanz stellt Zertifikate (2.2.2) aus und signiert sie für jeden neuen Teilnehmer oder Dienst in der PKI. Erst durch diese Signatur erlangt ein Zertifikat seine Gültigkeit.

Das gleiche gilt für Sperrinformationen bzw. Revozierungslisten (siehe Kapitel 2.2.4). Daraufhin hat das Schlüsselmanagement für den privaten Schlüssel der CA, die höchste Sicherheitsanforderung in einer PKI [SIGG01], diese Sicherheit wird z.B. durch physikalische Abschirmung oder Offline-Betrieb realisiert.

Eine Zertifizierungsstelle kann:

1. Eigenständig agieren und Zertifikate für Benutzer oder Dienste ausstellt, die nicht berechtigt sind selbst Zertifikate auszustellen. Sie signiert ihre eigenes Zertifikats selbst mit dem eigenen privaten Schlüssel.
2. Eine Hierarchie angehören mit einer Wurzel-CA und mehrere untergeordneten CAs. Die Wurzel-CA signiert sein Zertifikat selbst und die von den untergeordneten CAs. Solche CAs werden als besonderer Teilnehmer bezeichnet und können Zertifikate für andere CAs oder Benutzer ausstellen und signieren.

Bei der Überprüfung einer digitalen Signatur, wird der gesamte Zertifizierungspfad in der Hierarchie vom Signaturzertifikat bis hin zur Wurzel-CA betrachtet. Ein Zertifikat ist nur dann gültig, wenn alle Zertifikate im Pfad gültig sind.
Es gibt verschiedene Verfahren wie die Zertifikate im Zeitpunkt der Signatur auf ihre Gültigkeit überprüft werden können. Im Abschnitt 2.2.3 werden zu diesem Zweck die beiden Verfahren: Ketten- und Schalenmodell kurz vorgestellt.

Verzeichnisdienst (Certificate Management Authority – CMA)

Zertifikate und Sperrlisten (siehe Abschnitt 2.2.4) werden in einem Verzeichnisdienst gespeichert und zur Verfügung gestellt bzw. für alle Teilnehmer veröffentlicht. Ein Standard, der hierfür verwendet wird, ist der *Lightweight Directory Access Protocol* LDAP [rfc2251].
Im Fall der zentralen Erstellung des Schlüsselpaares, wird die Auslieferung der Hard- bzw. SoftToken vom Verzeichnisdienst übernommen, dies geschieht entweder per Post, E-Mail oder persönlich.

2.2.2 Zertifikat

Ein wichtiger Grundbegriff der PKI ist das digitale Zertifikat, das die Zuordnung eines öffentlichen Schlüssels zu seinem Besitzer organisiert. Es wird vom Trust Center signiert von dem es ausstellt wird. Das Zertifikat löst das Problem der authentischen Zugehörigkeit eines Schlüssels zu seinem Besitzer. Es muß nur noch die digitale Signatur des Zertifikates verifiziert werden, um das im Kapitel 2.1 erwähnte Schutzziel der Zurechenbarkeit zu gewährleisten.
Es gibt mehrere Standards für Zertifikate, das am weitesten verbreitete Standard ist das X509-Zertifikat ([rfc3280]). Ein X509 Zertifikate (siehe Abb. 2.1 hat drei Feldern, das *to-be-signed certificate* Feld, das *signature algorithm identifier* es definiert der vom Zertifikat Aussteller benutzte digitale Signatur Algorithmus für das Signieren des Zertifikates, und das *signature value* Feld, welches die digitale Signatur enthält.

Das to-be-signedcertificate Feld umfaßt folgende Angaben:

- Version – Versionsnummer dieses Zertifikatstyps
- Serial Number – Seriennummer des Zertifikats
- Signatur Algorithm – Signaturverfahren, mit dem dieses Zertifikat signiert wurde
- Issuer – Name der zertifizierenden Instanz
- Validity – Gültigkeitszeitraum
- Subject – Identität des Zertifikatsinhabers, was bei natürlichen Personen Land, Name und E-Mail-Adresse und bei Chipkarten die Chipkarten-Seriennummer sein kann
- Subject Public Key Info – Informationen zum öffentlichen Schlüssel des Zertifikatsinhabers, die aus dem öffentlichen Schlüssel und dem dazu gehörenden Signatur- oder Verschlüsselungsalgorithmus bestehen. Zum Signieren bzw. Verschlüsseln können dann verschiedene Verfahren genutzt werden, die diesen Algorithmus enthalten
- issuerUniqueID, subjectUniqueID – vorgesehen für die Behandlung der Wiederbenutzung des Subjekt-Name, diese beiden Feldern werden in der Praxis nicht benutzt
- Extensions – Erweiterungen, die beispielsweise den Verwendungszweck (Key Usage) des Schlüssels ("Non-Repudiation" für Nicht-Abstreitbarkeit, "Key-Cert-Sign" und "CRLSign" für spezielle Anwendungen von "Non-Repudiation", "Digital Signature" in Authentisierungsprotokollen, "Key-Encipherment" und "Key-Agreement" beim Schlüsseltransport und -austausch und "Data-Encipherment" für Datenentschlüsselung), alternative Namen oder Eigenschaften des Zertifikatsinhabers (Arzt oder Mitarbeiter von Firma anonymus.com) angeben.

Certificate Policy

Inwiefern der Benutzer einem Zertifikat vertrauen kann, hängt von mehreren Faktoren ab:

- Wie die CA die Zertifikate erzeugt und mit welchem Algorithmen sie signiert werden
- Wie die Sicherheit im Trust Center gewährleistet wird
- Wie die privaten Schlüssel verwahrt werden
- Welche rechtliche Bedingungen zur Geltung kommen

All diese Aspekte werden in einer *Zertifizierungspolicy* oder *Certificate Policy* – auch Policy oder *Certificate Practice Statement* (CPS) genannt – gehalten. Eine Policy besteht aus Regeln, die eine gewisse Sicherheitsanforderung erzielen. Wie eine Policy aussehen kann ist in [rfc2527] vorgegeben.

```
Certificate:
   Data:
       Version: 1 (0x0)
       Serial Number: 7829 (0x1e95)
       Signature Algorithm: md5WithRSAEncryption
       Issuer: C=ZA, ST=Western Cape, L=Cape Town, O=Twte GmbH,
               OU=Certification Services Division,
               CN=Twte Server CA/Email=server-certs@twte.com
       Validity
           Not Before: Jul  9 16:04:02 1998 GMT
           Not After : Jul  9 16:04:02 1999 GMT
       Subject: C=US, ST=Maryland, L=Pasadena, O=Brentt Balanca,
                OU=xyzsoft, CN=www.xyzsoft.org/Email=balanca@xyzsoft.org
       Subject Public Key Info:
           Public Key Algorithm: rsaEncryption
           RSA Public Key: (1024 bit)
               Modulus (1024 bit):
                   00:b4:31:98:0a:c4:bc:62:c1:88:aa:dc:b0:c8:bb:
                   35:19:d5:0c:64:b9:3d:41:b2:96:fc:f3:31:e1:
                   66:36:d0:8e:56:12:44:ba:75:eb:e8:1c:9c:5b:66:
                   70:33:52:14:c9:ec:4f:91:51:70:39:de:53:85:17:
                   16:94:6e:ee:f4:d5:6f:d5:ca:b3:47:5e:1b:0c:7b:
                   c5:cc:2b:6b:c1:90:c3:16:31:0d:bf:7a:c7:47:77:
                   8f:a0:21:c7:4c:d0:16:65:00:c1:0f:d7:b8:80:e3:
                   d2:75:6b:c1:ea:9e:5c:5c:ea:7d:c1:a1:10:bc:b8:
                   e8:35:1c:9e:27:52:7e:41:8f
               Exponent: 65537 (0x10001)
   Signature Algorithm: md5WithRSAEncryption
       93:5f:8f:5f:c5:af:bf:0a:ab:a5:6d:fb:24:5f:b6:59:5d:9d:
       92:2e:4a:1b:8b:ac:7d:99:17:5d:cd:19:f6:ad:ef:63:2f:92:
       ab:2f:4b:cf:0a:13:90:ee:2c:0e:43:03:be:f6:ea:8e:9c:67:
       d0:a2:40:03:f7:ef:6a:15:09:79:a9:46:ed:b7:16:1b:41:72:
       0d:19:aa:ad:dd:9a:df:ab:97:50:65:f5:5e:85:a6:ef:19:d1:
       5a:de:9d:ea:63:cd:cb:cc:6d:5d:01:85:b5:6d:c8:f3:d9:f7:
       8f:0e:fc:ba:1f:34:e9:96:6e:6c:cf:f2:ef:9b:bf:de:b5:22:
       68:9f
```

Abbildung 2.1: ein Beispiel eines X.509 Zertifikat

2.2.3 Gültigkeitsmodelle

Ein digitale Signatur soll den Zweck erfüllen u.a. die elektronischen Unterlagen zu beglaubigen. Dabei wird bei der Verifizierung immer auf das dazu zugehörige Zertifikat zurückgegriffen. Da jedes Zertifikat ein Gültigkeitsdatum hat, oder zu irgend einem Zeitpunkt T revoziert werden kann, entsteht ein Problem, wenn das Zertifikat zum Zeitpunkt der Signaturprüfung ungültig ist. Ein Bsp.: Bob signiert ein Dokument A im Zeitpunkt t1 mit seinem privaten Schlüssel, Alice will das Dokument A im Zeitpunkt t2 auf seine Gültigkeit überprüfen, wobei t1 ist kleiner als t2. Zum Zeitpunkt t2 war Bobs Zertifikat bereits abgelaufen. Wie kann sie feststellen, ob zum Zeitpunkt t1 der öffentliche Schlüssel vom Bob gültig war oder nicht.
Es existieren verschiedene Gültigkeitsmodelle, die das Problem beheben, die eine Zertifikatsprüfung zugrunde legen. An dieser Stelle werden die zwei grundlegende Modelle, das Schalenmodell (shell Model) und das Kettenmodell (chain Model) kurz vorgestellt.

2.2.3.1 Schalenmodell

Der Ansatz dieses Modells besteht darin, dass zum Zeitpunkt der Signaturprüfung alle Zertifikate der Zertifikatskette gültig sein müssen. Bei der Ausstellung eines Zertifikates ist darauf zu achten, dass der Gültigkeitszeitraum des ausgestellten Zertifikats den Gültigkeitszeitraum des ausstellenden Zertifikats nicht übertrifft. Zum Beispiel die Kompromittierung eines CA-Zertifikats führt zum sofortigen Widerruf aller untergeordneten Zertifikate. Der Vorteil des Schalenmodells ist die einfache Verifikation. Der Nachteil ist die Notwendigkeit von sehr langen Gültigkeitsperioden von Root und CA Zertifikaten. In Abbildung 2.2 wird der Ansatz graphisch dargestellt.

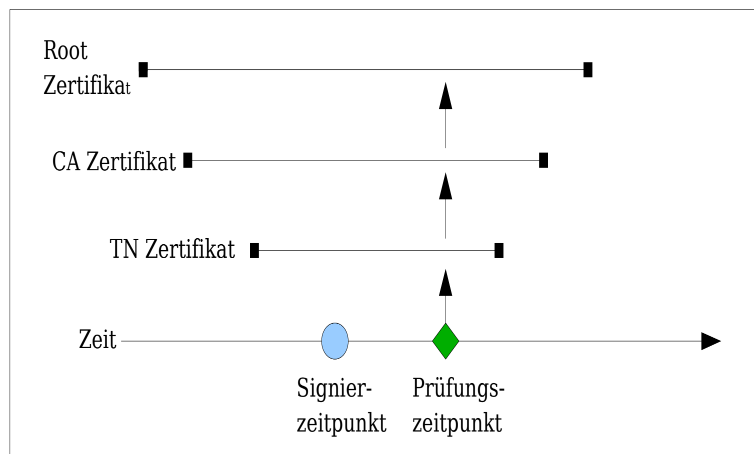

Abbildung 2.2: die Verifikation im Schalenmodell

2.2.3.2 Kettenmodell

Der Ansatz besteht darin, dass bei der Verifizierung geprüft wird, ob zum Zeitpunkt der Signaturerstellung, das Teilnehmerzertifikat des Signierers gültig war oder nicht. Für jedes Zertifikat in der Zertifikatskette muss zum Zeitpunkt der Zertifikatserstellung, das Zertifikat des Issuer gültig sein. Bei der Ausstellung eines Zertifikats darf das ausgestellte Zertifikat eine Gültigkeitsdauer besitzen, die über die Gültigkeitsdauer des ausstellenden Zertifikates hinausgeht. Die Zeiträume kann

man sich aneinandergekettet vorstellen. Widerrufsinformationen sollten auch nach Ende der Lebensdauer eines Zertifikates abrufbar sein, um bei einer Signaturprüfung den Erstellungszeitpunkt mit einem Widerrufszeitpunkt vergleichen zu können. In diesem Modell ist die Prüfung auf die Gültigkeit schwieriger als beim Schalenmodell.

Zur Veranschaulichung wird im Abbildung 2.3 der Ansatz graphisch dargestellt.

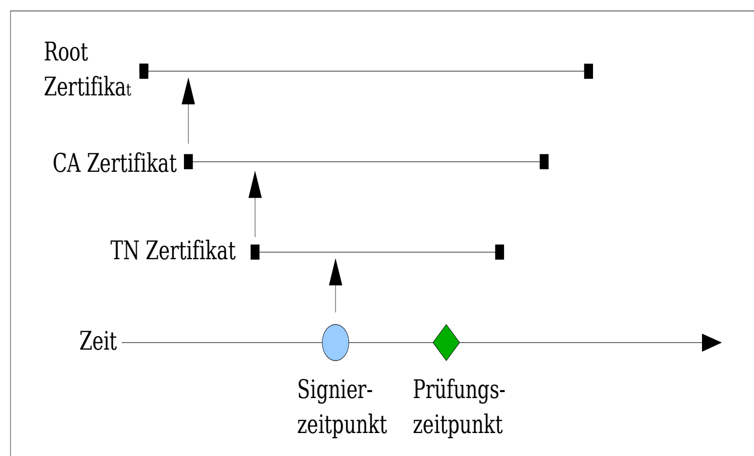

Abbildung 2.3: die Verifikation im Kettenmodell

2.2.4 Certificate Revocation List

Die X509-Zertifikate, die durch die Certification Authority (CA) revoziert worden sind, werden in einer Widerrufsliste auch als Certificate Revocation List [rfc3280] (CRL) gespeichert. Die CRL kann mit einer schwarzen Liste verglichen werden, welche die Zertifikate enthält, die nicht mehr gültig sind.
Im allgemeinen wird eine CRL von der CA ausgestellt und mit dem privaten Schlüssel der CA signiert.
Die CAs veröffentlichen die CRLs, um Status Angaben über die Zertifikate zur Verfügung zu stellen, die sie ausgestellt haben. Jedoch kann eine CA diese Aufgabe zu einer anderen vertraulichen Autorität beauftragen. Wenn der CRL-Aussteller nicht die CA ist, die die Zertifikate ausgestellt hat, wird die CRL als eine *indirekte CRL* gekennzeichnet.
Eine *vollständige CRL* listet alle nicht abgelaufenen Zertifikate die widerrufen worden sind wegen einen der Revozierungsgründe, die in der CRL enthalten werden können, auf. Ein Zertifikat wird revoziert wenn der private Schlüssel kompromittiert, gestohlen oder verloren wird. Weitere Gründe sind, der Mißbrauch des Zertifikats, die nicht Einhaltung der Zertifizierungspolicy, veraltete Informationen im Zertifikat oder wenn die Entität nicht mehr existiert.
Ein Zertifikat kann gesperrt werden, in diesem Fall kann das Zertifikat seine Gültigkeit zurückerhalten, bei der Revozierung jedoch nicht.
Die vollständige CRLs werden in regelmäßigen Abständen herausgebracht. Der CRL-Aussteller kann auch *DELTA CRLs* erzeugen. Eine DELTA CRL verzeichnet nur jene Zertifikate, deren Revozierungsstatus sich seit der Ausgabe der letzten vollständigen CRL geändert hat. Die bezogene

vollständige CRL wird als eine *Base CRL* gekennzeichnet. Eine DELTA CRL gehört immer zu einer Base CRL. Mit Hilfe der DELTA CRLs, wird versucht der PKI-Benutzer möglichst auf dem aktuellsten Stand zu halten und die Lücke zwischen zwei ausgestellten kompletten CRLs klein zu halten.

Die Felder eines CRLs

Eine Certificate Revocation List besteht nach der von der ITU standardisierte RFC 3280 [rfc3280] aus einer Sequenz von drei Feldern (Abbildung 2.4):

- tbsCertList
- signatureAlgorithm
- signatureValue

```
CertificateList ::= SEQUENCE {

    tbsCertList TBSCertList,
    signatureAlgorithm AlgorithmIdentifier,
    signatureValue BIT STRING }

TBSCertList ::= SEQUENCE {

    version Version OPTIONAL, -- if present, MUST be v2
    signature AlgorithmIdentifier,
    issuer Name,
    thisUpdate Time,
    nextUpdate Time OPTIONAL,
    revokedCertificates SEQUENCE OF SEQUENCE {

    userCertificate CertificateSerialNumber,
    revocationDate Time,
    crlEntryExtensions Extensions OPTIONAL -- if present, MUST be v2 } OPTIONAL,

    crlExtensions [0] EXPLICIT Extensions OPTIONAL -- if present, MUST be v2 }

-- Version, Time, CertificateSerialNumber, and Extensions --

-- AlgorithmIdentifier
```

Abbildung 2.4: die X.509 V2 CRL Syntax

tbsCertList ist das erste Feld in der Sequenz und besteht selbst aus einer Sequenz von mehreren Feldern (Abbildung 2.4). Das Feld *version* beinhaltet die Versionsnummer der CRL. Sind die weiteren, als „optional" markierten Felder vorhanden, muss die Versionsnummer *v2* sein. Im Feld *signature* wird der verwendete Algorithmus für die CRL-Signatur identifiziert. Die Felder

issuer, *thisUpdate* und *nextUpdate*, beschreiben jeweils der Name des CRL-Ausstellers, das Ausgabedatum der CRL und das Ausgabedatum der nächsten CRL. Im Feld *revokedCertificates* sind schließlich die Seriennummer der revozierten Zertifikate(*userCertificate*), das Revokationsdatum (*revocationDate*) und zusätzlichen Vermerke (*crlEntryExtensions*) gespeichert. Im crlEntryExtensions können u.a. Informationen über den Sperrungsgrund abgelegt sein. Im letzten Feld *crlExtensions* können verschiedene Informationen abgelegt werden, beispielsweise zusätzliche Daten über den CRL-Herausgeber, der CRL-Seriennummer und weitere Informationen über den Verteilungspunkt, von dem die Sperrliste bezogen wurde.

signatureAlgorithm enthält die Algorithmusbezeichnung für den von den CRL-Aussteller verwendete Algorithmus, um das CertificateList zu signieren. In [rfc3279] sind alle unterstützte Algorithmen für diese Spezifikation aufgelistet, andere signier Algorithmen können auch unterstützt werden.
Dieses Feld muss die gleiche Algorithmusbezeichnung enthalten, wie in dem signature Feld in der tbsCertList Sequenz.

CRLs müssen vom Client selbst heruntergeladen und verwaltet werden. Je mehr Zertifikate eine CA verwaltet, desto umfangreicher und größer wird die CRL. Da die CRL periodisch aktualisiert wird, bedeutet das mehr Aufwand für den Client. Die Anforderung an eine bessere Lösung muss den Punkten:

- Wie kann man den Datenvolumen der zu übertragenden Informationen geringer halten
- wie können die Informationen über die Widerrufstatus zeitlich genauer sein

gerecht sein werden
Für diese Anforderungen gibt es mehrere Lösungsansätze. Davon haben wir die Erweiterung des CRL-Ansatzes (DELTA-CRL) gesehen. Es gibt jedoch bessere Ansätze wie der Onlinedienst für Widerufsstatusinformationen. Im Kapitel 3 wird der Online Certificate Status Protocol, der ein solchen Dienst realisiert, vorgestellt und erläutert.

2.3 Cache

Ein Cache oder auch schneller Pufferspeicher genannt, ist ein begrenzter Speicherbereich für die temporäre Speicherung der Daten oder Rechenergebnisse, die wahrscheinlich wieder benutzt werden. Im Cache werden Kopien von den Daten die im Hauptspeicher sind, gespeichert. Mit dieser Eigenschaft wird versucht die Zugriffszeit zu verringern und die Performanz zu optimieren. Ein anderer Effekt beim Einsatz von Caches ist die verringerte Bandbreitenanforderung an die Hauptspeicherebene. Dadurch, dass oft die Anfragen vom Cache beantwortet werden können, sinkt die Anzahl der Zugriffe und damit die Bandbreitenanforderung an den Hauptspeicher. Die Anwendung des Caches bleibt jedoch für den Benutzer völlig transparent.
Die Verwendung des Caches wird sehr oft im Zusammenhang mit der CPUs gebracht. Um die Arbeit des Prozessors zu beschleunigen, wird ein Cache eingebaut. Es handelt sich dabei um eine Hardware Implementierung des Caches.
Der Cache kommt auch in Software Anwendungen zum Einsatz, beispielsweise bei Webbrowsern, Suchmaschinen oder diversen Software Applikationen.

2.3.1 Organisation und Aufbau eines Caches

Der Cache besteht aus mehreren Blöcken oder Einträge. Jeder Block hat ein Datum, das eine Kopie der im Hauptspeicher befindliche Information ist. Dazu hat jeder Block einen Tag-Anteil, der auf die Identität der Information im Hauptspeicher referenziert. Zu jedem Eintrag gehören auch mehrere Status-Bits die verschiedene Wirkungen haben.
Die Cache-Blöcke können je nach Anwendung entweder direkt abgebildet (direct mapped), vollassoziativ (fully associative) oder satzassoziativ (auch „Mengenassoziativ“ (set associative) genannt) sein.
Beim direkt abgebildeten Cache hat jeder Cache-Block eine genaue Adresse. Wird ein neuer Eintrag gespeichert oder gesucht , so wird mit Hilfe von Hash Funkionen und Modulo Rechnung die genaue Adresse des dazu gehörigen Block ermittelt (Abbildung von Adresse auf genau einen Cacheblock).
In einem vollassoziativen Cache können sich die Daten in jedem beliebigen Block im Cache befinden. Bei der Speicherung wird einfach der nächste freie Block gewählt. Bei der Suche werden alle Einträge des Caches überprüft.
Der mengenassoziative Cache ist eine Zusammensetzung der direkt abgebildeten und vollassoziativen Cache. Wobei der Cache aus mehreren adressierten Blöcken (direkt abgebildet) besteht. Jeder Block wiederum besteht aus mehreren Einträge die zu einander vollassoziativ sind.
Nach dem jetzigen technischen Möglichkeiten, es ist kaum möglich einen sowohl großen als auch schnellen Cache zu bauen. Daher werden bei manchen CPUs oder Applikationen mehrere Caches verwendet. In diesem Fall spricht man von einer Cache-Hierarchie. Die einzelnen Caches werden in verschiedene Levels (Ll, L2 bis Ln) geteilt (Moderne CPUs haben meist zwei bis drei Cache-Levels).
Der L1 bezeichnet hierbei den Cache mit der kleinsten Zugriffszeit, welcher als erstes durchsucht wird. Wurden die Daten im L1 nicht gefunden, wird in L2, der meist etwas langsamer und größer ist, gesucht usw. Wurde in diesem Cache-Hierarchie kein Treffer erzielt, werden die Informationen vom Hauptspeicher geholt.

2.3.2 Ersetzungsstrategie

Der Cache verfügt über eine begrenzte Größe, d.h. es können nicht alle Informationen im Cache gespeichert werden. Wurde ein Eintrag nach einer Anfrage im Cache nicht gefunden, so spricht man von einem „Cache-Miss“ und die Informationen werden dann vom Hauptspeicher geholt. Wurde der Eintrag im Cache gefunden so spricht man von einem „Cache-Hit“. In diesem Zusammenhang sind die zwei Begriff „Miss-Rate“ und „Hit-Rate“ von wichtige Bedeutung.
Die Hit-Rate beschreibt die Anzahl der Anfragen, bei denen ein Cache-Hit auftrat, geteilt durch die Anzahl der insgesamt an diesen Cache gestellten Anfragen. Die Miss-Rate lässt sich analog zur Hit-Rate berechnen oder ergibt sich aus der Formel: $MissRate = 1 - HitRate$.
Eine hohe Hit-Rate bedeutet, dass der Cache effizient arbeitet, daher es ist sinnvoll bei der Verwaltung des Caches, möglichst die häufig zugegriffenen Informationen im Cache zu behalten. Zu diesem Zweck existieren verschiedene Ersetzungsstrategien, diese bestimmen welche Eintrag im Cache ersetzt werden soll, im Falle, das alle Cache Blöcke besetzt sind. Von den verschiedenen Strategien werden an dieser Stelle ein paar genannt:

- First in First out (FIFO): Der älteste Eintrag wird ohne weiteres ersetzt

- Last in First out (LIFO): Der neuste Eintrag wird ohne weiteres ersetzt

- Clock: Der Eintrag mit dem ältesten Datum wird ersetzt, wobei bei jedem Zugriff (auch im Falle einer Cache-Hit) wird das Datum neu eingelesen und aktualisiert
- Random: in zufälliger Eintrag wird ersetzt
- Least Frequently Used (LFU): Der am wenigsten gelesene Eintrag wird ersetzt
- Least Recently Used (LRU): Der Eintrag, auf den am längsten nicht zugegriffen wurde, wird ersetzt

Eine der häufig benutzten und implementierten Ersetzungsstrategie ist die LRU. Es bleibt jedoch die Entscheidung des Entwicklers welche Strategie er für seine Anwendung für gut hält.

Kapitel 3

Online Zertifikat Revozierung

Zu den Hauptaufgaben einer Public Key Infrastructure, zählen neben der Verwaltung erstellte Zertifikate, den Kunden, über die nicht abgelaufene jedoch revozierte Zertifikate zu benachrichtigen, um ihn davon zu bewahren, diese ungültige Zertifikate zu akzeptieren. In einer PKI gehört diese Aufgabe zur Ceritification Authority (CA). Die CA veröffentlicht die Certificate Revocation List (CRL Abschnitt 2.2.4), die die revozierte Zertifikate enthalten.

Für viele Anwendungen sind jedoch die Sperrlisten nicht ausreichend oder in der Handhabung zu umständlich. Es ist oft nicht genug sich in regelmäßigen Abständen mit einer Sperrliste zu versorgen, da eine CRL z.B. bei einer Signaturprüfung nicht mehr aktuell sein kann. Außerdem können die CRLs im umfangreichen PKIs schnell mehrere Megabytes groß werden und dadurch unhandlich für den Client sein. Viele PKI Betreiber haben sich daher entschieden sich nicht nur auf CRLs zu verlassen. Als Alternative bieten sich die Online-Anfragen, die den Client das Herunterladen von CRLs sparen, und nur den Abruf von Sperrinformationen zu einem bestimmten Zertifikat zum Zweck haben. Das Internet-Standardisierungsgremium IETF hat zu diesen Zweck den Online Certificate Status Protocol(OCSP) standardisiert.

In diesem Kapitel wird der OCSP detailliert vorgestellt und erläutert. Dazu wird der Lightweight-OCSP vorgestellt, welcher das Ziel hat, den OCSP Standard zu performieren. Zuletzt wird der Simple Certificate Validition Protocol (SCVP) vorgestellt.

3.1 Online Certificate Status Protocol

Der Online Certificate Status Protocol (OCSP) ist ein Protokoll, welches die online Statusabfrage eines Zertifikates ermöglicht. Das OCSP-Protokoll wurde durch die PKIX Arbeitsgruppe IETF als Standard (RFC 2560 [rfc2560]) definiert. Es wurde im April 1999 von der PKIX Arbeitsgruppe der IETF als RFC 2560 spezifiziert und standardisiert. Es ist ein einfaches Request-Response-Protokoll, das den Revokationsstatus eines Zertifikats abfragen und übermitteln kann. Der OCSP besitzt kein eigenes Transportprotokoll. Es setzt auf beliebige Transportprotokolle wie z.B. http auf. Im Unterschied zum CRL Mechanismus, wird im OCSP die Statusüberprüfung ein oder mehrere Zertifikate in Echtzeit vorgenommen. Der Client schickt bei der Statusüberprüfung eines Zertifikats eine Anfrage an den OCSP-Server auch OCSP-Responder genannt. Der OCSP-Responder muss bei der CA autorisiert sein, um die Informationen über den aktuellen Zertifikatsstatus von der CA abfragen zu können und an den Client zu liefern. Die Antwort wird vom OCSP-Server digital signiert um die Authentizität sicherzustellen.

Je größer und umfangreicher die PKI ist, desto belastet die periodische Verteilung von CRLs an den Clients die Infrastruktur. Dies ist bei vielen Anwendungen nicht zu verkraften und kann die Sicherheit enorm beeinflussen. Im OCSP Verfahren, richtet der Client dagegen seine spezifische Anfrage an den OCSP-Responder, um den Status der entsprechenden Zertifikate zu prüfen. Der Client muss die CRLs nicht mehr herunterladen. Die OCSP-Response(Abschnitt 3.1.2) auf eine Client-Anfrage kann den Status „good“, „revoked“ oder „unknown“ enthalten. Der OCSP-Server, kann seine Informationen über eine Datenbank, LDAP oder CRL beziehen.

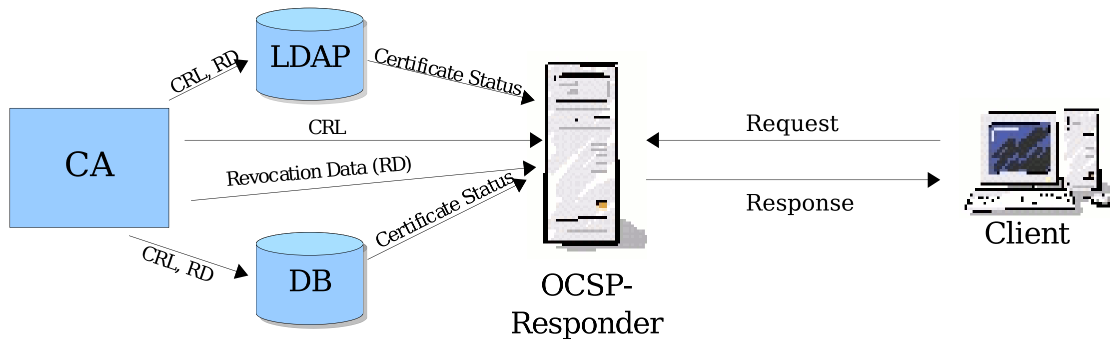

Abbildung 3.1: OCSP Infromationsbeschafung

Die gewählte Zugriffsmethode kann die Dienstgüte des OCSP-Responders erheblich beeinflussen. Außerdem muss die Authentizität des Datenaustausches gewährleistet werden.

3.1.1 Aufbau eines OCSP-Server

Kann der OCSP-Responder zu jedem Zeitpunkt die aktuellen Sperrinformationen zur Verfügung stellen, so spricht man von einem reinen Online-Statusüberprüfungsprotokoll. Dies kann der Fall sein, wenn der OCSP-Responder direkt in der CA integriert ist oder wenn der OCSP-Responder ein direkten Zugriff auf die CA-Datenbank hat. Die Integration ist dann sinnvoll, nur wenn der OCSP-Responder und die CA vom selben Hersteller stammen. Diese beiden Varianten haben jedoch der Nachteil, dass die CA ständig verfügbar sein muss, damit der OCSP-Server, der auf jeden Fall immer verfügbar sein muss, die Statusinformationen bekommen kann. Die Hochverfügbarkeit der CA ist jedoch wegen hohen Sicherheitsmaßnahmen und Kostengründen nicht oder nur sehr schwer realisierbar. Auf diesem Grund basieren die meisten OCSP-Responders auf die CRLs. Auf diese Weise sind zwar die Statusinformationen nicht auf dem aktuellsten Stand, dafür ist die Information um einiges günstiger. Außerdem muss die CA nicht ständig verfügbar sein wie der OCSP-Server. Somit werden die CRLs nur von den OCSP-Server und nicht von dem Client heruntergeladen, was die Kosten sinkt und die Infrastruktur entlastet.

Ein OCSP-Server kann ein Produkt von ein zur CA unabhängigen Hersteller sein und lediglich eine Schnittstelle zur CA Datenbank aufweisen. Falls die CA ein öffentliches Directory verwendet, kann z.B der Lightweight Directory Access Protocol (LDAP) zum Einsatz kommen. Bei den Protokollen, die die erforderliche Sicherheitsmaßnahmen nicht unterstützen, muss die Authentizität

besonders gewährleistet werden können. Zusätzlich kann die Übertragung durch die Sicherheitsprotokolle wie SSL/TLS oder IPSec [EckItSec] geschützt werden. Diese OCSP-Architektur entspricht dem eigentlichen Sinn von OCSP. Die Zertifikatsdaten sind auf der CA-Datenbank gespeichert, sie werden vom OCSP-Responder auf Anfrage geholt und das Zertifikatsstatus wird an den Client übermittelt. So gesehen, sind die Sperrinformationen in jedem Zeitpunkt aktuell.
Basiert der OCSP-Server jedoch ausschließlich auf die CRLs, so muss ständig überprüft werden, ob die CA eine neue CRL ausgestellt hat. Dabei basiert die Aktualität der Informationen auf die der CRL. Darunter leidet jedoch die Aktualität der Daten. Der Vorteil des CRL-basierten OCSP-Responders gegenüber der Nutzung von CRLs ist, die Entlastung des Clients von dem Herunterladen und der Verwaltung von CRLs sowie die Entlastung der Netzinfrastruktur und die damit geringere Kosten für alle Beteiligten. Diese OCSP Variante entspricht zwar nicht ganz dem ursprünglichen OCSP-Konzept, jedoch wird sie meist implementiert und verwendet.

3.1.2 Einzelheiten des OCSP-Protokolls

Die OCSP-Daten sind in der Abstrakt Syntax-Notation 1 (ASN.1) definiert. Für die Signatur werden die zu signierende Daten mit der ASN.1 distinguished encoding rules (DER) kodiert.

OCSP Request

Der Client schickt zu den OCSP-Server eine Anfrage, mit dem zu prüfenden Zertifikat. Eine Anfrage kann auch mehrere Zertifikate enthalten und sie kann vom Client signiert werden. In diesem Abschnitt ist der Request nach der ASN.1 Notation spezifiziert. Die tatsächliche Formatierung der Anfrage kann von den benutzten Transportprotokoll (HTTP, smtp, LDAP, etc) abhängen.

Der Request besteht im Allgemeinen aus zwei Sequenzen, *tbsRequest* und *optionalSignature*. Diese wiederum bestehen aus mehrere Sequenzen (siehe Abb. 3.2). Im folgenden werden die Request-Felder erläutert:

- *tbsRequest*: enthält eine Versionsnummer, aktuell gibt es nur die Version 1. Falls der OCSP-Request signiert wurde, muss der Name des Clients in *requestorName* angegeben werden. Dies kann z.B der Relative Distinguished Name, die URL, die IP-Adresse, etc. sein. *requestList* enthält eine Folge von Einzelanfragen. Der Request kann durch optionale *requestExtensions* ergänzt werden

- *requestList*: enthält die Anfrage selbst. Die Liste der Einzelanfragen kann mehrere unterschiedliche Zertifikatsreferenzen enthalten. Diese Zertifikate müssen nicht alle von der gleichen CA ausgestellt sein

- *signature*: besteht aus *signatureAlgorithm* , *signature* und *certs*. Die Signatur wird im Feld signatureAlgorithm angegeben. Der Bitstring signature enthält den Binärwert der digitalen Signatur. Im Feld certs kann eine Menge von Zertifikaten, die der OCSP-Responder bei der Signaturverifikation verwenden kann, übergeben werden. Die Signatur wird über die gesamte tbsRequest-Struktur berechnet. Falls eine OCSP-Anfrage signiert wird, so muss der Anfragende seinen Namen im requestorName-Feld der tbsRequest-Struktur angeben

- *request*: In diesem Feld werden die einzelnen Zertifikate bezeichnet (ein Zertifikat aus der requestList) deren Status abzufragen ist. Der Request besteht aus dem Zertifikatsbezeichner *certID*. Zusätzlich können eine oder mehrere Anfrageerweiterungen *singleRequestExtensions*, die sich auf dieses Zertifikat beziehen, angefügt werden

- *reqCert*: Ein Zertifikat wird mit *CertID* vom Feld *reqCert* eindeutig identifiziert. Das Feld *issuerNameHash* enthält das Hashwert von den DER-codierten Namen des Zertifikats des Ausstellers. Das Feld *issuerKeyHash* bezeichnet das Hashwert des öffentlichen Schlüssels des Ausstellers. Der wird über den subject Public Key (ausschließlich den Tag- und die Längewert) des Zertifikat des Ausstellers berechnet. Der benutzte Algorithmus für die Berechnung der beiden Hashwerte wird im Feld *hashAlgorithm* spezifiziert. Das Feld *serialNumber* enthält die Seriennummer des Zertifikates, das abgefragt werden soll

```
OCSPRequest     ::=     SEQUENCE {
       tbsRequest                   TBSRequest,
       optionalSignature   [0]      EXPLICIT Signature OPTIONAL }

   TBSRequest      ::=     SEQUENCE {

       version             [0]      EXPLICIT Version DEFAULT v1,
       requestorName       [1]      EXPLICIT GeneralName OPTIONAL,
       requestList                  SEQUENCE OF Request,
       requestExtensions   [2]      EXPLICIT Extensions OPTIONAL }

   Signature       ::=     SEQUENCE {
       signatureAlgorithm      AlgorithmIdentifier,
       signature               BIT STRING,
       certs               [0] EXPLICIT SEQUENCE OF Certificate
   OPTIONAL}

   Version         ::=             INTEGER  {  v1(0) }

   Request         ::=     SEQUENCE {
       reqCert                      CertID,
       singleRequestExtensions      [0] EXPLICIT Extensions OPTIONAL }

   CertID          ::=     SEQUENCE {
       hashAlgorithm       AlgorithmIdentifier,
       issuerNameHash      OCTET STRING, -- Hash of Issuer's DN
       issuerKeyHash       OCTET STRING, -- Hash of Issuers public key
       serialNumber        CertificateSerialNumber }
```

Abbildung 3.2: Aufbau eines Requests nach der ASN.1 Syntax [rfc2560]

Die Unterstützung von spezifischen Erweiterungen (extension) ist optional. Im Abschnitt 3.1.2 werden ein paar wichtige Erweiterungen vorgestellt.

In dem Fall, dass der Client sein OCSP-Anfrage signieren will, muss die Signature auf die tbsRequest Struktur berechnet werden. Dazu soll der Client sein Name im Feld *requestorName* eintragen. Der Client kann auch die Zertifikate im Feld *certs* eintragen, die den OCSP-Responder dabei helfen können, die Signatur des Clients zu prüfen.

OCSP Response

Der OCSP-Server beantwortet die Client-Anfrage mit einer Response-Nachricht. Die Response können von den verschiedenen Typen sein. Es gibt eine „Basic-Typ“ Response, die von allen OCSP-Responder unterstützt werden muss. Für einen Basic OCSP-Responder, wird das Feld *responseType* mit *id-pkix-ocsp-basic* belegt (Abb. 3.3). Diese Abschnitt beschreibt die Basic Response eines OCSP-Responders.
Der Schlüssel womit die Response signiert ist, muss:

- entweder die CA, die das angefragte Zertifikat ausgestellt hat, zugehören
- oder von einem vertraulichen Responder, der der Client seinen öffentlichen Schlüssel vertraut
- oder von einem CA-Autorisierten-Responder, der eine spezielle Zertifikat von der CA ausgestellt bekommen hat, die den Responder erlaubt OCSP-Responses für diese CA auszustellen

Eine OCSP-Response besteht im Allgemeinen aus einem *OCSPResponseStatus* Feld und *responseBytes* Sequenz wie in Abb. 3.3 dargestellt ist.

```
OCSPResponse ::= SEQUENCE {
      responseStatus         OCSPResponseStatus,
      responseBytes          [0] EXPLICIT ResponseBytes OPTIONAL }

OCSPResponseStatus ::= ENUMERATED {
       successful            (0),  --Response has valid confirmations
       malformedRequest      (1),  --Illegal confirmation request
       internalError         (2),  --Internal error in issuer
       tryLater              (3),  --Try again later
                                   --(4) is not used
       sigRequired           (5),  --Must sign the request
       unauthorized          (6)   --Request unauthorized
   }

ResponseBytes ::=       SEQUENCE {
       responseType   OBJECT IDENTIFIER,
       response       OCTET STRING }
```

Abbildung 3.3: OCSPResponse in ASN.1 Kodierung [rfc2560]

Tritt kein Fehler auf, so beinhaltet das Feld *responseStatus* den Wert successful. Wenn der OCSP-Responder jedoch den Request nicht richtig bearbeiten kann, so liefert er eine nicht signierte

Fehlermeldung, die die Fehlerursache beschreibt. Es können die folgenden Fehlermeldungen auftreten (siehe Abb. 3.3):

malformedRequest bedeutet, dass der erhaltene Request, der OCSP-Syntax nicht entspricht

internalError zeigt, dass der OCSP eine inkonsistente interne Zustand erreicht hat. Der Request soll wiederholt werden, eventuell zu einem anderen Responder

tryLater besagt, dass der OCSP-Server verfügbar ist, aber vorübergehend nicht antworten kann

sigRequired wird zurückgeliefert, wenn der OCSP-Server den Client explizit fordert, die Anfrage zu signieren

unauthorized wird angegeben, wenn der Client nicht berechtigt ist diese Anfrage zu stellen

```
id-pkix-ocsp           OBJECT IDENTIFIER ::= { id-ad-ocsp }
id-pkix-ocsp-basic     OBJECT IDENTIFIER ::= { id-pkix-ocsp 1 }
```

Abbildung 3.4: Inhalt des *responseType* Feldes [rfc2560]

Das Feld *ResponseBytes* setzt sich aus das Feld *responseType*, das den Typ der Response beschreibt (Abb. 3.4) und die Sequenz *response*. Die Sequenz *response* in der Sequenz ResponseBytes soll die DER Kodierung der BasicOCSPResponse (siehe Abb. 3.5) entsprechen. Sie besteht im Allgemeinen aus:

- Version der Response Syntax
- Name des OCSP-Responders
- Response für jedes angefragte Zertifikat in dem Request
- Optionale Erweiterungen
- Signatur Algorithmus OID
- die berechnete Signatur über den Hashwert der Response

Die Response, für jedes einzelne Zertifikat, besteht aus aus folgenden Komponenten: einer eindeutigen Kennzeichnung des Zertifikats, der Status des Zertifikats, das Gültigkeitsintervall der Response und optionale Extensions.

Response Status

Der OCSP-Responder liefert den Status des angefragten Zertifikats in einer signierten Response falls es keine Fehler bzw. Ausnahmen eingetreten sind. Die Signatur soll über den Hashwert, des in DER Kodierte ResponseData (Abb.3.5) berechnet werden.

Eine Response kann einen der drei Status enthalten:

good dieser Zustand gibt an, dass die Anfrage einen positiven Status hat. Das bedeutet lediglich, dass diese Zertifikat nicht revoziert ist, aber nicht notwendigerweise, dass das Zertifikat je ausgestellt wurde oder dass es nicht abgelaufen ist. Jedoch können die Response Erweiterungen mehr Informationen über das Zertifikat enthalten, wie z.B. Gültigkeitszeitraum, Angaben über die Ausstellung des Zertifikats etc.

revoked dieser Zustand zeigt an , dass das Zertifikat dem OCSP-Server bekannt ist und temporär oder endgültig gesperrt ist. Im Falle der Revokation, wird der Sperrzeitpunkt im Feld *revocationTime* und optional der Sperrgrund im Feld *revocationReason* eingetragen

unknown dieser Zustand bedeutet, dass das angefragte Zertifikat dem OCSP-Responder nicht bekannt ist. Entweder ist er von der entsprechenden CA nicht für die Beantwortung von Statusabfragen autorisiert, oder es können keine Revokationsinformationen zu dem Zertifikat gefunden werden

Zeiten einer OCSP-Response

Die OCSP-Response liefert neben dem Status auch verschiedene Zeiten, die den Client Informationen über die Response, Zertifikat und vieles mehr bieten. Die Zeiten werden in vier verschiedenen Felder erfasst (Abb. 3.5):

1. ***producedAt***: liefert der Erstellungs- und Signierzeitpunkt einer OCSP-Response. Der Zeitpunkt darf die Systemzeit nicht überschreiten, sonst ist die Response unglaubwürdig

2. ***thisUpdate***: enthält den Zeitpunkt, für den die gemachte Aussage gültig ist. Diese Zeitpunkt entspricht das thisUpdate von der zugehörigen CRL. OCSP-Response, deren thisUpdate Zeitpunkt nach der lokalen Systemzeit liegt, sollten als ungültig betrachtet werden. Es ist die Entscheidung des Client, wie weit dieser Zeitpunkt zurückliegen darf

3. ***nextUpdate***: beschreibt die Zeit, wann die neue Informationen über das angefragte Zertifikat verfügbar sein wird. Dieser Zeitpunkt entspricht den von der entsprechenden CRL. Liegt diese Zeit vor der Systemzeit, soll diese Response als unzuverlässig angenommen werden. Liegt keine nextUpdate vor, ist die Response äquivalent zu einer CRL die kein nextUpdate enthält

4. ***revocationTime***: gibt den Sperrzeitpunkt eins Zertifikats an, zu dem die Sperrung erfolgte

Wenn es sich um einen reinen OCSP-Server handelt, der nicht auf CRLs basiert, sondern der seine Daten direkt von der CA-Datenbank erhält, so wären die Zeiten thisUpdate und producedAt gleich. Auch nextUpdate wäre in diesem Falle überflüssig, da zu jedem Zeitpunkt die aktuellen Informationen zur Verfügung ständen.

```
BasicOCSPResponse       ::= SEQUENCE {
    tbsResponseData      ResponseData,
    signatureAlgorithm   AlgorithmIdentifier,
    signature            BIT STRING,
    certs                [0] EXPLICIT SEQUENCE OF Certificate OPTIONAL }

   ResponseData ::= SEQUENCE {
    version              [0] EXPLICIT Version DEFAULT v1,
    responderID              ResponderID,
    producedAt               GeneralizedTime,
    responses                SEQUENCE OF SingleResponse,
    responseExtensions   [1] EXPLICIT Extensions OPTIONAL }

   ResponderID ::= CHOICE {
    byName               [1] Name,
    byKey                [2] KeyHash }

   KeyHash ::= OCTET STRING -- SHA-1 hash of responder's public key
   (excluding the tag and length fields)

   SingleResponse ::= SEQUENCE {
      certID                       CertID,
      certStatus                   CertStatus,
      thisUpdate                   GeneralizedTime,
      nextUpdate         [0]       EXPLICIT GeneralizedTime OPTIONAL,
      singleExtensions   [1]       EXPLICIT Extensions OPTIONAL }

   CertStatus ::= CHOICE {
       good        [0]     IMPLICIT NULL,
       revoked     [1]     IMPLICIT RevokedInfo,
       unknown     [2]     IMPLICIT UnknownInfo }

   RevokedInfo ::= SEQUENCE {
       revocationTime              GeneralizedTime,
       revocationReason    [0]     EXPLICIT CRLReason OPTIONAL }

   UnknownInfo ::= NULL -- this can be replaced with an enumeration
```

Abbildung 3.5: DER Kodierung der BasicOCSPResponse [rfc2560]

Extensions

Die Erweiterungen ermöglichen ein anpassungsfähiges OCSP-Protokoll, das je nach individuellen Bedarf angepasst werden kann. Die im OCSP verwendeten Extensions, entsprechen dem Format von den in X.509-Extension der X.509v3-Zertifikate. Im OCSP werden zwei Möglichkeiten angeboten um die Erweiterungen zu nutzen. Es gibt Extensions, die der Request bzw. die Response betreffen und die die angefragten Zertifikaten einzeln betreffen [rfc2560] und [rfc2459].
Die Unterstützung von Erweiterungen ist optional. Die Erweiterungen sollten als nicht-kritisch markiert werden. Werden sie jedoch als kritisch markiert (das Flag critical wurde gesetzt), können die nicht anerkannte Extensions nicht mehr ignoriert werden. Dies hat zur Folge, dass der Request nicht beantwortet werden kann. Im [rfc2560] und [rfc2459] werden die wichtigsten Extensions erläutert, hier wird lediglich auf die zwei Erweiterungen, *Nonce* und *retrieveIfAllowed* eingegangen.

- ***Nonce*** Extension dient dazu eine Replay-Attacke zu unterbinden. Sie bindet eine Response zu genau einem Request. Die Nonce kann in Request- bzw. ResponseExtensions enthalten sein. In beiden Fällen wird die Nonce mit der Objekt *identifier id-pkix-ocsp-nonce* identifiziert.
  id-pkix-ocsp-nonce OBJECT IDENTIFIER ::= id-pkix-ocsp 2
  Die Nonce-Extension kann z.B. eine Zufallszahl sein. Die Nonce wird vom OCSP-Responder in die Antwort miteinbezogen um die Signatur der Response zu berechnen.

- ***retrieveIfAllowed*** Extension ist eine zertifikatsspezifische Erweiterung, die besagt, ob das Zertifikat mit der Response publiziert werden darf oder nicht. Ist ein Zertifikat nicht „public“ und wird es trotzdem vom Client im Request verlangt, so wird das Zertifikat vom OCSP-Server nicht mit dem Status geliefert.

3.1.3 Performanz- und Sicherheitsüberlegungen

Wie bei jedem anderen Internetprotokoll stellen sich die Fragen wie robust und sicher das Protokoll ist, welche Angriffe möglich sind und wie man sie vermeiden kann.
Ein OCSP-Server muss ständig verfügbar sein. Es ist jedoch möglich die Verfügbarkeit des OCSP-Responders zu beeinträchtigen, entweder gezielt und bewusst mit einer Denial-of-Service-Attacke, oder unbewusst, wenn die PKI-Umgebung sehr umfangreich ist und der OCSP-Responder nicht alle Anfragen schnell genug beantworten kann. Durch eine Flut von Anfragen kann der OCSP-Server der Belastung nicht mehr Stand halten. Die Auslastung wird vor allem durch die Signatur jeder Antwort auf jede Anfrage erzeugt. Auch unsignierte Antworten können zu einer Denial-of-Service führen, falls der Angreifer gezielt falsche Responses vom Server fordert. Es ist möglich diese Art von Angriffe abzuwehren in dem der Client identifiziert werden muss, bevor die Anfrage bearbeitet werden kann und falls der Verdacht vorliegt, dass es sich um einen Angreifer handelt, diesen Client zu blockieren. Des weiteren sind die Pre-Produced Responses zu nutzen, d.h. die signierten Antworten können gespeichert werden und für andere Clients verwendet werden.

Durch signierte Responses kann nicht nur die Wahrscheinlichkeit einer Denial-of-Service-Attacke verringert werden, sondern auch die Performanz und Skalierbarkeit verbessert werden. Jedoch es verbirgt sich dahinter auch die Gefahr einer Replay-Attacke zum Opfer zu fallen. Ein Angreifer kann nämlich die alte gute signierte Response, die vor dem Verfallsdatum revoziert wurde, noch einmal benutzen. Dies verhindert die Erhaltung von aktuellen Revokationsinformationen und stattdessen bekommt der Client falsche Informationen. Wir haben im Abschnitt 3.1.2 gesehen das ein

solcher Angriff durch die Verwendung einer Nonce vorgebeugt werden kann. Dafür erlaubt die Nonce nicht die Verwendung von bereits signierten Responses. Falls jedoch die Pre-Produced Responses gewünscht sind, werden die Nonce nicht verwendet. Ein Möglichkeit besteht darin ein Zeitstempeldienst zu nutzen, der die Aufgabe der Nonce ersetzen kann, und die Wiederverwendung der signierten Antworten erlaubt.

Ein weiterer wichtiger Aspekt ist die Client-Server-Zeitsynchronisation, aus diesem Grund müssen die Toleranzzeiten dem entsprechend genau bzw. kurz gewählt werden.

Ein Man-in-the-Middle-Angriff ist auch nicht ganz auszuschließen, wenn die Authentikation beide Seiten nicht erfolgt oder falls das Authentikationsverfahren nicht mehr sicher ist. Ein Angreifer kann dann z.B. die signierte Response vom Server erhalten und nicht signierte Fehlermeldung an den Client weiterleiten. Eine gegenseitige Authentikation beugt zwar dieser Gefahr vor, jedoch öffnet sie die Möglichkeit für ein Denial-of-Service-Angriff.
Der private Schlüssel des OCSP-Responders muss besonders geschützt werden. Schafft es ein Angreifer in Besitz des privaten Schlüssels zu kommen, so kann er beliebige Responses erstellen und signieren und irreführende Informationen über den Revokationsstatus eines Zertifikats erteilen und dadurch die Legitimation einer Signatur entziehen.
Bei sensiblen Anwendung zählt jede kleine Verzögerung der Response, aus diesem Grund muss die Performanz des OCSP-Servers maximal optimiert werden um die Aufgabe des Online Service nicht zu verfehlen. Es werden viele Vorschläge gemacht wie man ein OCSP-Responder je nach Anwendungsumgebung anzupassen hat um eine bessere Performanz und Skalierbarkeit zu erzielen.
Es wurde an ein OCSPv2 gedacht und dafür wurden mehrere Drafts von der ITEF herausgegeben, jedoch der OCSPv2 konnte sich nicht durchsetzen und wurde nicht weiter entwickelt.
Im nächsten Abschnitt werden die folgenden Verfahren: Der Lightweight-OCSP und der Service Validition Certificate Protocol (SVCP), der eine andere Funktionsweise als OCSP hat,vorgestellt.

3.2 Lightweight-OCSP

Die Forderungen nach einem skalierbaren und kosteneffektiven Zertifikat Status Verfahren werden immer höher, mit den ständig wachsenden PKIs sowie deren Einsatz in diversen Umgebungen. Der OCSP, wie er derzeit definiert und eingesetzt wird, genügt nicht den Anforderungen von sehr umfangreichen PKIs oder PKIs-Umgebungen die eine minimale Kommunikationsbandbreite beanspruchen wie z.B. bei mobilen oder drahtlosen Umgebungen.
Der *Lightweight Online Certificate Status Protocol* (L-OCSP) wurde von der PKIX Working Group in einem Internet Draft [L-OCSP] beschrieben. Der L-OCSP zielt darauf, die Skalierbarkeit des OCSPs in einer umfangreichen PKI-Umgebung mit sehr großen Datenvolumen oder in einer PKI-Umgebung die eine minimale Kommunikationsbandbreite beansprucht.

Im L-OCSP werden die Nachricht sowie der OCSP-Client und der OCSP-Responder so definiert, dass sie folgendes erlauben:

1. OCSP-Response Pre-Produktion und Verbreitung

2. maximale Reduzierung der OCSP-Nachricht-Größe

3.2.1 Das Verhalten in einem Lightweight-OCSP

Die Lightweight OCSP-Request und OCSP-Response basieren auf den im Abschnitt 3.1 vorgestellten OCSP-Request und OCSP-Response. In diesem Abschnitt werden lediglich die Änderungen bzw. die Anforderungen, die der OCSP erfüllen muss, erläutert, um die Skalierbarkeit eines L-OCSPs zu erreichen.

Struktur der Lightweight OCSP-Request und OCSP-Response

Der OCSP-Request muss verschiedene Bedingungen erfüllen, um den Request eines Lightweight-OCSPs [rfc2560] zu entsprechen:

- In der *OCSPRequest.RequestList* Struktur darf nur ein Request[1] enthalten
- Die Clients müssen der SHA1 Algorithmus für die Berechnung der Werte CertID.issuerNameHash und CertID.issuerKeyHash benutzen
- Die Clients dürfen weder die *singleRequestExtensions* noch die *requestExtensions* Struktur im Request einfügen
- Die Clients sollen keine signierte OCSP-Requests senden. Die Responders können signierte Requests ignorieren und müssen eingestellt werden, unsignierte OCSP-Requests die ein *requestorName* Feld enthalten zu bearbeiten. Der *requestorName* reicht jedoch nicht für die Authentizität aus

Der Responder soll keine responseExtensions in die Response einfügen. Wie in [rfc2560] spezifiziert ist, müssen die Clients unerkannte, nicht-kritische responseExtensions in der Response ignorieren. Im Falle, dass ein OCSP-Responder nicht die Fähigkeit hat, einen Request, der eine Nonce enthält zu beantworten, oder falls er nur Pre-produzierte Antworten benutzen kann, soll der Responder eine Response senden, die keine Nonce enthält. Der Client soll die Response abarbeiten können, selbst wenn diese keine Nonce beinhaltet.
Die Responders, die mit einer Nonce versehener Request nicht abarbeiten können, können dieser Request zu einem anderen Responder weiterleiten der in der Lage ist solche Requests zu beantworten.

Um zu gewährleisten, dass die Datenbank der Revokationsinformationen mit der Zeit nicht unbegrenzt wächst, soll der Responder der Status der abgelaufenen Zertifikate löschen. Der Responder soll dem zufolge mit dem unsiginerten Response *unauthorized* auf die Requests die nach einem abgelaufenen Zertifikat fragen, antworten. Dazu kann ein OCSP-Responder, der im voraus die Responses berechnet nicht alle mit dem Status „unknown“ vorbereiten und signieren, da er nicht in der Lage ist alle „unknown“ certIDs im Voraus zu kennen.
In einer pre-produzierten OCSP-Response entsprechen die *thisUpdate*, *nextUpdate* und *producedAt* Zeiten, die einer normalen OCSP-Response (siehe Abschnitt 3.1.2). Die Zeit *nextUpdate* muss immer in einer Response enthalten sein um die Caching Option zu erleichtern.

[1] es darf nur nach einer Zertifikat pro Request angefragt. d.h. die Länge des RequestLists gleich eins sein muss

Clients Verhalten

Um unnötigen Netzwerkverkehr zu vermeiden, müssen andere Applikationen die Signatur der signierten Daten prüfen, bevor den OCSP-Client eine Anfrage an den OCSP-Server schickt um den Status des benutzen Zertifikats für die Verifikation des Datens zu prüfen. Falls die Signatur ungültig ist, oder die Applikation, sie nicht verifizieren kann, darf ein OCSP-Anfrage nicht stattfinden.
Ebenso, muss eine Applikation die Signatur eines Zertifikats in einer Validierungskette validieren bevor ein OCSP-Anfrage durchgeführt wird. Die OCSP-Clients sollen keine Anfragen für abgelaufene Zertifikate an das OCSP-Server senden.

Ein Client muss sicherstellen, dass die vom OCSP erhaltene Antwort aktuell ist. Eine Antwort ist nicht aktuell, wenn der in der OCSP-Response enthaltene Status *good* ist, jedoch der OCSP in der Lage ist, eine Antwort zu erstellen die den Status *revoked* für das gleiche Zertifikat enthält.
Im Allgemeinen sind zwei Mechanismen vorhanden, die die Aktualität der Response gewährleisten. Der erste Mechanismus nutzt eine Nonce und der zweite basiert auf die Zeit. Die zweite Lösung verlangt, dass der OCSP-Client und der OCSP-Responder beide über eine präzise Zeitquelle verfügen.

Da im Lightweight-OCSP der Client eine Nonce-Extension nicht einfügen soll, um die pre-Produzierte Antworten nutzen zu können, muss der Client in der Lage sein über die Aktualität der Response anhand des Zeitstempels zu entscheiden. Die Clients, die sich trotzdem für eine Nonce entscheiden, sollen die Responses die ausschließlich auf die Zeit basieren, nicht ablehnen. Sie sollen auf die zeitbasierte Antworten zurückgreifen können um den Status des angefragten Zertifikats zu ermitteln. Die Clients die keine Nonce im Request einfügen, müssen jegliche Nonce ignorieren, die in der Response auftauchen können.
Der Client muss die Response auf die Existenz des *nextUpdate* Felds prüfen und muss auch sicher sein, dass die momentane Zeit zwischen die *thisUpdate* und *nextUpdate* Zeiten liegt. Ist eine der beiden Bedingungen nicht erfüllt, darf der Client der Antwort nicht vertrauen. Der Client kann ein Abweichungsinterevall festlegen um eine geringe Zeitdifferenz zwischen ihm und dem Responder zu behandeln. Der Zeitintervall ist von der Synchronisierung der Zeit zwischen Server und Client abhängig.

Cache Empfehlungen

Die Fähigkeit die OCSP-Response überall im Netzwerk cachen zu können, ist ein sehr bedeutender Aspekt in den großen PKI Umgebungen.
Um die Netzlast zu minimieren, müssen die Clients die authorisierte Response lokal speichern. Die authorisierte Response sind OCSP-Responses, die eine „successful“ Status haben. Um der Flut von Anfragen, die unmittelbar nach der *nextUpdate* Zeit von Clients geschickt werden, die Ihren Cache aktualisieren wollen, muss der OCSP-Responder in der OCSP-Response angeben, wann der Client die neue OCSP-Response abrufen kann.
Der OCSP-Responder soll den HTTP Header von der OCSP-Response derart setzen, sodass das Caching in den HTTP Proxys erlaubt ist.
Die Responses können auch im OCSP-Server gespeichert werden und nach Anfrage wieder verwendet werden.

Die Cache-Option bietet viele Vorteile davon sind zu erwähnen:

- Entlastung des OCSP-Servers, da der Server nicht jede Anfrage neu bearbeiten und signieren muss
- der Client kann auch im offline Modus die Zertifikatsvalidierung durchführen und muss nicht eine neue HTTP Session etablieren[2]

3.2.2 Sicherheitsüberlegungen

Die Tatsache, dass die Benutzung der Nonce optional ist, birgt die Gefahr, dass die Wahrscheinlichkeit einem *Replay* Angriff größer wird als sonst. Beim Replay Angriff kann ein Angreifer eine abgelaufene Response verwenden und der Client denkt, dass diese Response *good* ist, obwohl der Zertifikat-Status sich geändert hat und das Zertifikat revoziert ist. Ein solchen Angriff kann vermieden werden, wenn die Response mit einem Zeitstempel versehen wird und der Client Zugriff auf eine präzise und korrekte Zeitquelle hat, die die Aktualität der Response garantieren kann. Es bleibt zu erwähnen, dass die Toleranz zwischen den Server- und der Clientclock die Korrektheit der Response erheblich beeinflussen kann. Zum Beispiel ein Client mit einer Uhr die vorgeht als die Uhr des Servers, kann eine gültige Response als nicht gültig betrachten, ebenso kann ein Client, eine ungültige Response als gültig betrachten wenn seine Uhr nachgeht, als die Uhr des Servers.

Der *Man-in-the-middle* Angriff kann dadurch verhindert werden, in dem der Server die Responses signiert und der Client muss die Signatur der Response regelrecht validieren.
Da die *HTTP headers* nicht geschützt sind, da sie im Klartext transportiert werden, birgt das HTTP Cachen von der Response die Gefahr, dass ein Angreifer den HTTP Header manipulieren kann. Der Client soll nur die signierte Response annehmen und darf die gecachte Response, die die *nextUpdate* Zeit überschreitet, nicht vertrauen.
Der Lightweight-OCSP erlaubt die Verwendung von unsiginierten Requests, diese Maßnahme ermöglicht es jeder, der eine Request schicken kann, Zugriff auf den Responder. Somit steigt die Wahrscheinlichkeit, dass der Responder einen Denial of Service Angriff zugesetzt werden kann, in dem eine Flut von Request wahrscheinlicher wird. Ein Responder kann einen solchen Angriff vermindern, indem er die Anzahl der Requests, die von einer IP-Adresse kommen beschränkt, falls ein verdächtiges Verhalten entdeckt wird.

Der Lightweight-OCSP versucht eine Lösung für große und umfangreiche PKIs, sowie für die Umgebungen die einen minimalen Datenaustausch benötigen, um die Performanz in diesen Umgebungen zu verbessern. Jedoch kann die Sicherheit darunter leiden, wenn die nicht richtigen Maßnahmen getroffen werden. Es bleibt die Entscheidung des Entwicklers und PKI-Betreibers wie sie einen Trade-off zwischen der Sicherheit und der Performanz in einem OCSP-Responder finden und realisieren wollen.

[2] nur wenn das Caching auf der Client Seite implementiert ist

3.3 Simple Certificate Validition Protocol

In diesem Abschnitt wird der Simple Certificate Validition Protocol (SCVP) vorgestellt. Es werden die Zusatzfunktionen des SCVP im Gegensatz zum OCSP aufgezeigt und miteinander verglichen. Anschliessend wird der Verlauf des Protokolls erläutert. Dieses Kapitel gibt lediglich einen Überblick über den SCVP. Für ausführlichere Details wird auf die Literatur ([DraftSCVP] und [rfc3379]) verwiesen.

3.3.1 Einführung in das SCVP Protokoll

Das OCSP Protokoll bietet die Möglichkeit, einen Zertifikatstatus online zu prüfen, und somit entlastet es die Infrastruktur und der Client braucht die CRLs nicht herunterzuladen. Jedoch muss im OCSP der Client selbst die Validierung der Zertifikate und die Konstruktion der Zertifikatskette ausführen. Dazu prüft der OCSP das Zertifikat nur auf den Widerruf und nicht auf die zeitliche Gültigkeit und den korrekten Verwendungszweck des Zertifikates. So gesehen, bietet der OCSP keine Verringerung der Validierungskomplexität auf der Clientseite, da die Konstruktion und die Validierung der Zertifikatskette vom Client durchzuführen ist. Um den Client diese Aufgabe zu ersparen, wurde ein weiteres Protokoll vorgeschlagen und von den IETF in mehreren Drafts (im Januar 2007 wurde das draft-ietf-pkix-scvp-31.txt herausgegeben) immer weiter entwickelt. Dieses Protokoll ist der *Simple Certificate Validation Protocol (SCVP)*, der eine ähnliche Aufgabe wie der OCSP erfüllen soll, jedoch zusätzliche Funktionen bietet. Die Aufgabe des SCVPs ist, die zentralisierte Implementation der Zertifikatsvalidierung und die Minimierung des Implementationsaufwand auf die Seite des Clients. Der SCVP ist wie der OCSP, ein einfaches Request-Response-Protokoll, das kein eigenes Transportprotokoll besitzt sondern mit anderen Transportprotokolle arbeiten kann, wie z.B. HTTP.

Das SCVP-Protokoll soll den Client eine partielle bis vollständige Auslagerung der Zertifikats-Validierung ermöglichen. Der Server kann dabei den Client mit zusätzliche Informationen neben dem Revokationssatus eines Zertifikates versorgen. Dazu zählt z.B. ob das Zertifikat oder der Zertifikatspfad bis zu einer bestimmten Root-CA gültig ist.

Das SCVP-Protokoll wurde mit dem Fokus auf zwei Klassen von Benutzern entwickelt:

1. Benutzer, die zwar die Validierung einer Zertifikatskette selbst durchführen können, diese jedoch nicht selbst konstruieren können. Es gibt viele Gründe, die es den Client die Konstruktion nicht ermöglichen, wie z.B. wenn der Client die nötigen Protokolle nicht implementiert hat oder wenn der Aufwand der Konstruktion die Möglichkeiten des Clients überschreitet.

2. Benutzer, die nur am Ergebnis der Validierung eines Zertifikates für eine bestimmte Anwendung interessiert sind. Da sie beispielsweise nicht über die nötigen Ressourcen verfügen, um selbst die Validierung zu tätigen, oder die Sicherheitsrichtlinien durch den SCVP-Server eingehalten werden können.

Der SCVP bietet somit die Möglichkeit die protokollseitige Abwicklung von Requests zur:

Delegated Path Discovery [rfc3379] hier wird die Konstruktion einer Zertifikatskette durchgeführt. Der Server ermittelt für den Client die Vertrauenspfade und leitet alle Informationen für die Prüfung an den Client weiter

Delegated Path Validation [rfc3379] zielt auf die Validierung einer Zertifikatkette. Der Server ermittelt für den Client die Vertrauenspfade und prüft sie anschließend mittels verschiedener Verfahren wie CRLs oder OCSP. Das Ergebnis wird anschließend an den Client verschickt

3.3.2 Protokollablauf

Der SCVP ist wie der OCSP, ein einfaches Request-Response-Modell. Der SCVP-Client erstellt eine Anfrage (Request) und schickt sie an den SCVP-Server. Der Server erzeugt daraufhin eine Antwort (Response) und sendet sie zurück an den Client. Der SCVP kann verschiedene Transportsprotokolle wie z.B. HTTP verwenden.

Der SCVP spezifiziert zwei Request-Response Abläufe:

1. Der Client fragt den Server nach unterstützten Validierungsrichtlinien (validation policy [rfc3379]). Die Validierungsrichtlinie ist anwendungsspezifisch (S/MIME, IPsec, TLS) und gibt an, wie die Validierung erfolgen soll (Konfiguration des SCVP-Servers). Der Server spezifiziert in der Antwort die unterstützten Richtlinien durch die stellvertretende Objekt-IDs.

2. Der Client beauftragt den Server mit der (teilweisen) Validierung von Zertifikaten. Der Client sendet die Zertifikat-IDs, die durchzuführende Aktionen und die Kontext-Information (soweit sie nicht durch die Sicherheitsrichtlinie vorgegeben sein) und spezifiziert, welche Informationen der Server liefern soll. Der Client spezifiziert auch den Zeitpunkt, somit ist eine Prüfung eines Zertifikates in einem in der Vergangenheit liegenden Zeitpunkt möglich. Der Server antwortet mit den geforderten, signierten Daten oder mit einer nicht signierten Fehlermeldung.

Beispiele der Request-Response Szenarien können veranschaulicht werden wie im Abb. 3.6 dargestellt ist. Auf die Details der Request und Response Aufbau und Syntax wird an dieser Stelle nicht eingegangen, es sei auf den Internet Draft von IETF [DraftSCVP] verwiesen.

Aufgrund der hohen Komplexität des SCVP Verfahrens, konnte sich das SCVP-Prtokoll im Gegensatz zum OCSP-Protokoll nicht durchsetzen und hat keine große Verbreitung wie der OCSP. Es existieren jedoch verschiedene Implementierungen des SCVP.

```
SCVP Request {
here is a target certificate: <certificate>
Following checks should be performed: build
certificate path
I want back: certificate path
}

SCVP Response {
here is a target certificate: <certificate>
Reply checks: built a path or couldn't build a path
Reply want backs: certificate path
}

SCVP Request {
here is a target certificate: <certificate>
Following checks should be performed: build a
validated certificate path
I want back: certificate path
}

SCVP Response {
here is a target certificate: <certificate>
Reply checks: Valid / Not Valid
Reply want backs: certificate path
}

SCVP Request {
here is a target certificate: <certificate>
Following checks should be performed: Perform full
validation including revocation checking
I want back: Status Indicator and Public Key Value
}

SCVP Response {
here is a target certificate: <certificate>
Reply checks: Good,Revoked,Unknown,unavailable
Reply want backs: Status Indicator: valid / not valid
Public key value field obtained from a valid target
certificate
}
```

Abbildung 3.6: Verschiedene Request-Response Szenarien eines SCVP-Protokolls

Kapitel 4

Design und Entwurf

Dieses Kapitel gibt einen Überblick über die vorhandene Software [flexsec] und die Erweiterungen die zur Steigerung der Performanz vorgenommen wurden. Die grundlegenden Design-Entscheidungen zur Optimierung der Software werden erläutert und diskutiert.
Die Implementierung, des hier vorgestellten Entwurfes sowie die Klassen und Methoden, die aus der Entwicklersicht wichtig sind, werden im Kapitel 5 beschrieben. Die Erläuterung ist von großer Bedeutung um die interne Struktur der Software genauer zu verstehen und ggf. weiter zu entwickeln.

4.1 Allgemeine Architektur

Der OCSP-Responder ist als J2EE-Anwendung umgesetzt worden. Er setzt sich aus den dafür typischen Bausteinen zusammen: Servlets, Enterprise Java Beans (EJBs) und MBeans, wie in Abbildung 4.1 dargestellt ist.
Ein Servlet übernimmt die Kommunikation zwischen den OCSP-Client und OCSP-Responder. Die Daten werden dann an die BusinessLogikSchicht (BLS) zur Analyse weitergereicht. Dort übernehmen die EJBs (der RequestProcessor und das CertificateManagement) die weitere Verarbeitung. Dazu benötigen sie Zugriff auf diverse Ressourcen (Dateisystem, Kartenleser, LDAP-Server). Gemäß dem JMX-Standard (Java Management Extensions) werden diese Anwendungsressourcen in Form von MBeans verwaltet.

Prinzipiell greift in dieser Schichtenarchitektur immer die obere Schicht auf die untere Schicht zu: Das Servlet benutzt die EJBs, diese greifen wiederum auf die MBeans zu. Es gibt jedoch im OCSP-Responder eine Ausnahme: Das CryptoProcessorEJB enthält u.a. eine Reihe von Methoden zum Verifizieren von Zertifikaten und CRLs, die auch von den MBeans genutzt werden.
Des Weiteren gibt es eine Reihe von Hilfsklassen, die als normale Java-Objekte („POJOs“) umgesetzt sind. In ihnen sind Funktionen wie das Logging, der Zugriff auf die Konfigurationsdatei und die diversen Caches für die Zertifikate, die CRLs, die Issuers und die bereits beantwortete Responses untergebracht. Diese Hilfsklassen werden aus allen drei Schichten heraus benutzt. Ebenso werden vereinzelt Funktionen aus CryptoProcessorEJB und den MBeans aus den Utility-Klassen benutzt (ServiceHelper ist eine Wrapper-Klasse für den Zugriff auf diverse Mbeans).

Der InternalTrigger, der dafür sorgt, dass der CRLcache bei Veränderungen im LDAP aktualisiert wird, ist ebenfalls als MBean umgesetzt worden. Gesteuert wird das Scannen nach Updates

vom SchedulerService von JBoss. Durch die Umsetzung als MBean lässt es sich allerdings auch manuell ausführen.

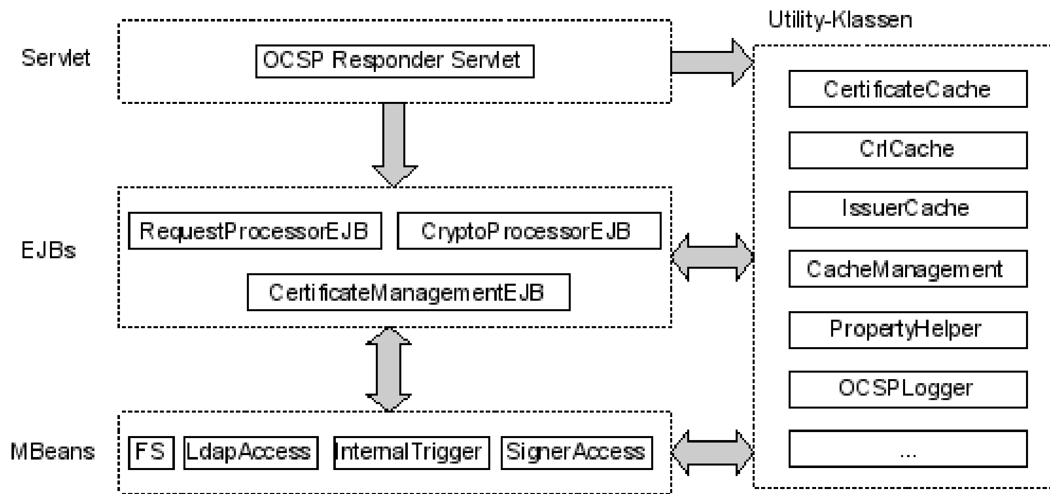

Abbildung 4.1: Architektur des OCSP-Responders

4.2 Interne Datenstrukturen

Der OCSP-Responder unterhält für die Bearbeitung der Anfragen vier Hauptcaches (Abbildung 4.2).

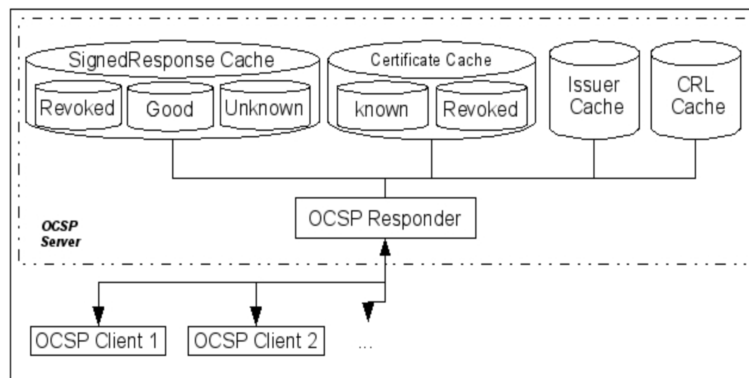

Abbildung 4.2: Interne Datenstruktur

Der CRLcache und der Issuercache werden beim Starten des OCSP-Servers vom LDAP aufgefüllt. Sie enthalten jeweils die Revokationslisten (CRLs) und die sämtliche Issuer-Zertifikate. Der SignedResponsecache besteht aus drei unabhängige Caches, je nach Status (good, revoked oder unknown) werden die signierte Antworten im entsprechenden Cache gespeichert. Die Aufteilung in drei verschiedenen Caches zielt erstens darauf die Vereinfachung und Beschleunigung der Aktualisierung des Caches und zweitens die Möglichkeit eine falsche Antwort mit einem falschen Status zu unterdrücken. Der vierte Cache ist der Zertifikatscache und unterteilt sich in zwei unabhängigen Caches, ein mit den revozierten Zertifikaten (Revoked) und der andere mit den produzierten Zertifikaten,

die bis zum Zeitpunkt der Anfrage, den OCSP bekannt und noch nicht revozierte worden sind (known). Die zwei Caches, Zertifikatscache und SignedResponsecache sind im Gegensatz zu den anderen beiden Caches CRLcache und Issuercache zum Beginn leer, sie werden mit der Zeit je nach Abfragen aufgefüllt.

4.2.1 CRL Cache

Im CRLcache werden Revokationsinformationen zu den Zertifikaten gehalten. Diese Informationen werden aus CRLs generiert. Dabei werden zwei Datenstrukturen angelegt: Die erste enthält Informationen über die CRL (clientID, Issuer, thisUpdate usw.), die zweite enthält Informationen zu den CRL-Einträgen (s. Abbildung 4.3).
Die Informationen über die CRLs werden in CRLInfo-Objekten abgelegt. Der Zugriff darauf erfolgt über eine Hashtable mit den clientIDs als Schlüssel.
Für jeden Eintrag in den CRLs wird ein CRLCacheObject mit dem Revokationsdatum und einer Referenz zum dazugehörigen CRLInfo-Objekt angelegt. Der Zugriff auf die Objekte erfolgt über zwei Hashtables, von denen die eine Hash-Algorithmen auf die entsprechende zweite Hashtable mappt, die wiederum Zertifikatsseriennummer, issuerNameHash und issuerKey-Hash als Schlüssel verwendet, um zu den CRLCache-Objekten zu gelangen. Die CRLs werden beim Starten des OCSP-Servers im CRLcache abgelegt, der CRLcache wird ständig mit Hilfe des Triggers (4.3.2) aktualisiert.
Da der CRLcache immer eine neue CRL beinhaltet, wurde auf einen Cache, mit einer vordefinerten Größe und einer Ersetzungsstrategie verzichtet. Der Cache mit einer vordefinerten Größe und einer Ersetzungsstrategie bringt keine sichtbare Performanzverbesserung in diesem Fall, weil immer die gleiche Zeit und Aufwand beansprucht wird und die CRL immer als ganze im Cache geladen wird.

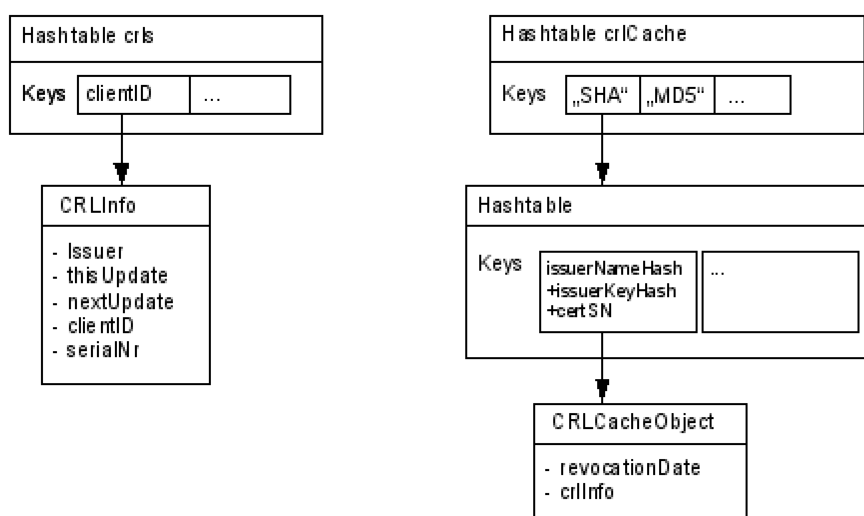

Abbildung 4.3: Datenstruktur des CRLcaches

4.2.2 Issuer Cache

Der Issuercache enthält Informationen zu den Mandanten und zu deren Issuern. Die Issuer-Zertifikate werden vom Dateisystem eingelesen. Unter den Begriff „Issuer“ fallen hier Root- und CA-Zertifikate sowie CRL- und OCSP-Signer inkl. deren ausstellenden Zertifikaten. Issuer-Zertifikate können auch von externen CAs produziert worden sein.
Der Issuercache umfasst deutlich mehr einzelne Datenstrukturen als die SignedResponse-, Zertifikats- und CRLcache. Das liegt daran, dass von sehr vielen verschiedenen Stellen des OCSP-Responders auf den Issuercache zugegriffen wird. Je nach Stelle werden unterschiedliche Informationen benötigt und liegen unterschiedliche Informationen vor. So kann zum Beispiel anhand der Daten aus dem Request (certID) ein Issuer-Zertifikat eindeutig identifiziert werden, auch wenn zwei Issuer-Zertifikate den selben SubjectDN besitzen, da in certID auch noch der Hashwert des Public Keys vorhanden ist. Wenn dagegen der CRLcache aufgebaut wird, ist nur der SubjectDN des Issuer-Zertifikats vorhanden. An einer dritten Stelle wird gar kein Issuer benötigt, sondern nur eine Liste aller clientIDs. Um all diese unterschiedlichen Anfragen beantworten zu können, verwaltet der Issuercache die folgenden Datenstrukturen:

- Einen Vector mit allen Issuern
- Einen Vector mit allen clientIDs
- Eine Hashtable, die zu einer clientID alle dazugehörigen Informationen aus der Konfigurationsdatei liefert
- Zwei Hashtables, die - ähnlich wie in CRLcache - zu issuerNameHash und issuerKeyHash (für verschiedene Hash-Algorithmen) den passenden Issuer finden. Damit wird in den Klassen CertificateManagementEJB und RequestProcessorEJB, in denen certID vom Request vorliegt, nach dem passenden Issuer zum angefragten Zertifikat gesucht
- Fünf Hashtables, die Distinguished Names auf Issuer-Objekte mappen. Jede dieser Hashtables entspricht einem Verzeichnis mit Issuer-Zertifikaten im Dateisystem. Diese Hashtables werden in den Fällen benutzt, in denen der Public Key des Issuer-Zertifikats nicht vorhanden ist und deshalb ausschließlich der DN für die Suche nach dem Issuer-Zertifikat verwendet werden kann.
  Die Auftrennung in fünf Hashtables ist sinnvoll, da sie erstens der Verzeichnisstruktur entspricht und zweitens unterschiedliche Zwecke mit den einzelnen Zertifikatstypen verbunden sind. So dienen manche der Zertifikate der Zuordnung zu Mandanten (clientIDs), während andere für die Verifizierung von Signaturen verwendet werden. Je nachdem, um welchen Mandanten es sich handelt, kann ein Zertifikat unterschiedliche Funktionen übernehmen. Zum Beispiel kann Zertifikat A das CA-Zertifikat von Mandant 1 sein, wenn es aber ein OCSP-Signer-Zertifikat für Mandant 2 ausgestellt hat, dann hat es für Mandant 2 die Funktion der Verifizierung von OCSP-Signer-Zertifikaten

  Die einzelnen Hashtables im Überblick sind:

  - dnToCrtIssuerCert: Enthält CA-Zertifikate und wird in CRL- und Zertifikatscache für die Suche nach dem Issuer der jeweiligen Zertifikate (bzw. CRL-Einträge) verwendet. Die Issuer-Objekte aus dieser Hashtable werden in CRL- und Zertifikatscache referenziert,

was u. a. bedeutet, dass es zu falschen Zuordnungen zwischen Zertifikat und Issuer-Objekt (und damit auch dem Mandanten) kommen kann, falls mehrere Zertifikate mit gleichem DN in den „crtCa“-Verzeichnissen liegen

- dnToCrlSignerCert: Enthält Aussteller-Zertifikate von CRLs. Wird vom CRLcache verwendet, um die CRL zu verifizieren (falls hier kein passender CRL-Signer gefunden wird, wird auch noch bei dnToCrtIssuerCert gesucht) und um den issuerKeyHash für die CRL-Einträge zu ermitteln. Der InternalTrigger verwendet sie, um die clientID zu einer CRL zu finden
- dnToCrlSignerIssuerCert: Enthält Zertifikate, die CRL-Signer-Zertifikate signiert haben. Wird vom OCSP nicht verwendet
- dnToOcspSignerCert: Enthält OCSP-Signer-Zertifikate. Wird im SignerAccess-Service für die Zuordnung von Zertifikaten zu Mandanten (clientIDs) verwendet
- dnToOcspSignerIssuerCert: Enthält Issuer-Zertifikate von OCSP-Signern und wird im Signer Access Service für die Verifizierung der OCSP-Signer-Zertifikate verwendet

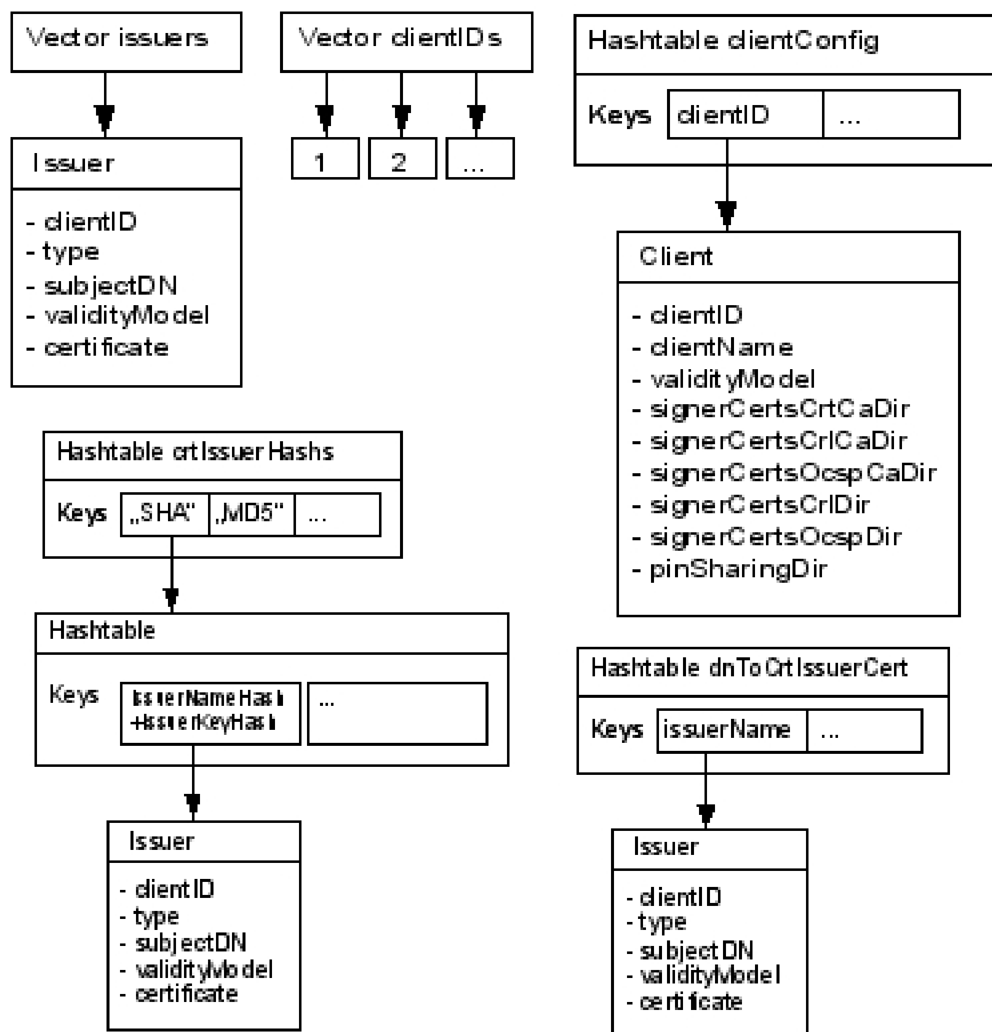

Abbildung 4.4: Datenstruktur des Issuercaches

Sämtliche oben aufgeführten Datenstrukturen des Issuercaches sind in den Abbildung 4.4 graphisch dargestellt. Von den fünf Hashtables, die einen DN auf Issuer-Objekte mappen, ist nur dnToCrtIssuerCert in Abbildung 4.4 dargestellt, die anderen vier Hashtables sind identisch aufgebaut. Analog zum CRLcache wird auch der Issuercache beim Initialisierung des OCSP-Servers aufgefüllt.

4.2.3 Certificate Cache

Um die Performanz des OCSP-Responders zu erhöhen, wurde an dieser Stelle ein Cache entworfen, der die „Pre-Produced“ Responses beinhaltet. Der Cache bietet die Möglichkeit auf die Zeit, die eine SignerCard benötigt, um eine Response zu signieren, zu verzichten, da die Signature die meiste Zeit in Anspruch nimmt. So gesehen kann der OCSP-Responder viel mehr Anfragen in einen bestimmten Zeitintervall beantworten als vorher. Dadurch wird auch die Gefahr eines Denial-OF-Service Attack [Attacks] vermindert. Will man einen solchen Cache zur Verfügung stellen, der die Pre-Produced response beinhaltet, stosst man auf das folgende Problem:
Ein Client-Request kann mehrere Extensions enthalten [rfc2560]. Eine davon ist die *Nonce-Extension*, die eine wichtige Rolle spielt, sie zielt auf die Verhinderung des Replay Attacke [Attacks]. Beinhaltet ein Request eine Nonce, so wird die Response dem entsprechend konstruiert, sodass diese Nonce in der Signatur miteinbezogen wird. Dies hat zur Folge, dass jede Response einen Client zugeordnet ist, der sie angefragt hat und somit für jeden Client individuell zusammengebaut werden muss.
Um das Problem zu lösen, wurden zwei Caches erzeugt die verschiedene Informationen verwalten. In diesem Abschnitt wird der Zertifikatscache der die unsignierte Daten beinhaltet, beschrieben.

Der Zertifikatscache hält alle Informationen zu den angefragten Zertifikaten aus dem Master LDAP. Falls sie öffentlich („public“) sind, befinden sich die Zertifikate selbst im Cache, ansonsten (d. h. bei nur nachprüfbaren Zertifikaten) werden nur die Informationen gespeichert, die für die Antwort notwendig sind. Dies umfasst neben der Seriennummer, die implizit als Teil des Keys für die Hashtable gespeichert wird, noch den Hash-Wert des DER-kodierten Zertifikats für die CertHash-Extension.

Die Zertifikatsdaten werden in „CertificateCacheObjects“ abgelegt. Der Wert von CertHash wird beim Aufruf des Konstruktors aus dem übergebenen Zertifikat erzeugt. Des Weiteren wird dem Konstruktor ein Flag übergeben, das angibt, ob es sich um ein abrufbares Zertifikat handelt, wenn es der Fall ist, so wird eine Referenz auf das Zertifikatsobjekt behalten.

Für den Zugriff auf die Zertifikatsdaten über „certID“,[1] sollte der OCSP einerseits eine frei konfigurierbare Auswahl an Hash-Algorithmen unterstützen und andererseits ein möglichst schneller Zugriff auf die Inhalte des Zertifikatscaches gewährleistet sein. Die Daten aus certID bilden den Key um ein Zertifikat eindeutig zu identifizieren. Die Tatsache, dass der OCSP alle erlaubten

[1] CertID enthält den issuerNameHash, issuerKeyHash und Seriennummer des Zertifikats sowie den Algorithmus, mit dem die Hashes erzeugt worden sind.

Hash-Algorithmen[2] unterstützen soll, wird nicht beeinträchtigt da der Schlüssel aus dem gehashten Werten besteht.

Wenn ein Zertifikat im Cache gesucht werden soll, dann wird direkt der Schlüssel aus den Informationen des Requests gebildet und es wird der passende Eintrag aus der LRUMap Tabelle geholt, falls er vorhanden ist, ansonsten wird in der Master LDAP gesucht.

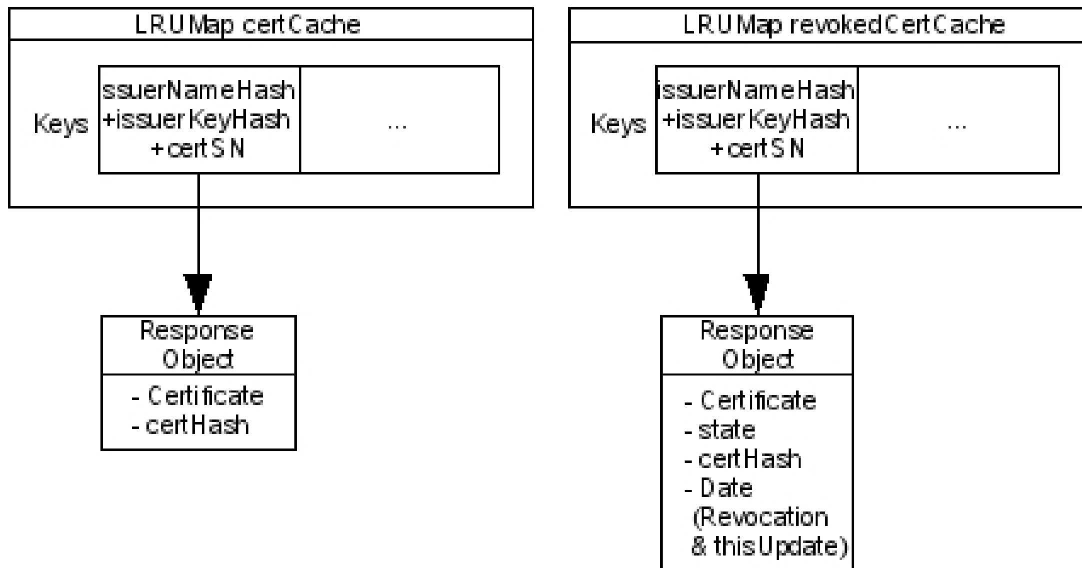

Abbildung 4.5: Datenstruktur des Certificatecaches

Der Zertifikatscache besteht aus Performanzgründen aus zwei LRUMaps Datenstrukturen die zu einander unabhängig sind (siehe Abbildung 4.5).

- certCache: hält die Zertifikate die in Zeitpunkt $x < t$ angefragt worden sind und zu diesem Zeitpunkt noch nicht revoziert sind.

- revokedCertCache: hält nicht signierte Response von den revozierten Zertifikate.

Im Gegensatz zu den CRLcache und Issuercache, die beim Starten und Initialisieren des OCSPs vollständig ausgefüllt werden, ist der Certificatecache zu dieser Zeit noch leer und wird dann im Verlauf der Client-Request-Beantwortung nachgefüllt. Dazu wird den Certificatecache eine einheitliche Größe zugewiesen, die man in dem Property File festlegen kann, dieser definiert dann die Anzahl der Einträge des Certificatecache. Die Einschränkung der Größe betrifft die restlichen Caches nicht.
Die Auftrennung in zwei LRUMaps ist sinnvoll, da sie die Zugriffszeit verkürzen und eine erhebliche Vereinfachung der Aktualisierungsaufwand bieten. Zudem wird hier auf den Triggers verzichtet, da die Daten, die nicht im Cache vorhanden sind, direkt aus dem LDAP geholt werden.
Wird ein Zertifikat angefragt, so wird zuerst im revokedCertCache gesucht, ist ein Treffer erzielt

[2] Gemäß ISIS-MTT muss ein OCSP-Responder den Hash-Algorithmus SHA-1 und sollte die Hash-Algorithmen RIPEMD160 und MD5 unterstützen. Der FlexiTrust OCSP-Responder kann durch Konfigurationsänderung um weitere Algorithmen erweitert werden.

worden, so wird die Response signiert und versendet. Anderenfalls wird im certCache gesucht, wurde hier auch nichts gefunden dann wird im LDAP gesucht. Wurde das Zertifikat im certCache oder im LDAP gefunden so wird der Status überprüft, wenn der Status *revoked* ist, wird die Response im revokedCertCache gespeichert und vom certCache gelöscht, ist der Status jedoch *good* wird das Zertifikat im certCache aufgenommen, falls es nicht bereits vorhanden war.

Obwohl mit dem Zertifikatscache das Ziel, die SignerCard Prozedur zu umgehen, nicht erreicht wird, wird im Falle der revokedCertCache die Zeit für die Sammlung der nötigen Informationen für eine komplette Response vor der Signature um das dreifache verkürzt. Es wird vor allem durch dem Zertifikatscache die Zeit für das Starten des OCSP-Servers erheblich verringert, da die Zertifikate die im LDAP sind, nicht mehr im OCSP-Cache gespeichert werden müssen. Der Speicherplatzverbrauch des OCSPs für die Zertifikate wird vom Betreiber selbst verwaltet.
Jedoch, wenn die Zertifikate nicht im Zertifikatscache gespeichert sind, werden sie vom Master LDAP geholt und die Beantwortungszeit eines Request kann sich erhöhen.

4.2.4 SignedResponse Cache

Der SignedResponsecache verwaltet die Informationen, die eine komplette OCSP-Response darstellen (siehe Abbildungen 4.6). Diese Informationen sind die Responses für die Anfragen, die keine Nonce-Extension beinhalten und somit schon vom OCSP-Responder signiert sind und für die Absendung zum Client bereit sind. Dieser Cache bringt größere Vorteile mit gegenüber, den mit den nicht signierten Responses (Zertifikatscache) was die Performanz anbelangt, da die Signercards nicht zum Einsatz kommen, falls die Response schon im Cache vorhanden ist. Die einzige Bedingung neben dem Verzichten auf eine Nonce-Extension auf der Clientsseite oder dem Ignorieren auf der Responderseite, ist die Beschränkung des RequestLists[3] auf genau ein Zertifikat pro Request (requestList =1).

Der SignedResponsecache umfasst mehr Datenstrukturen als der nicht signierte Cache (Zertifikatscache). Das liegt daran, dass:

1. Es werden alle Reponse gespeichert unabhängig vom Status
2. Um die Effizienz der Suche im Cache und deren Aktualisierung zu optimieren. Wie beim Zertifikatscache wird auf das Triggern verzichtet
3. Im Cache werden wie bereits erwähnt die signierte OCSP-Responses gespeichert, d.h. sie werden direkt zum Client verschickt, weil sie bereit vom OCSP-Responder signiert sind. Da der Status eines Zertifikates sich jede Zeit ändern kann[4], muss jede Response noch einmal in dem Zeitpunkt der neuen Anfrage auf die Richtigkeit dessen Status geprüft werden, mehr dazu im Abschnitt 4.3. Somit ist die Wahrscheinlichkeit, dass eine OCSP-Response mit dem falschen Status zum Client verschickt werden könnte, ausgeschlossen

Hier sind nochmal die drei Datenstrukturen des *SignedResponsecache*:

- **sigGoodRespCache**: verwaltet Informationen zu den signierten OCSP-Responses mit dem Status *good* in einem LRUmap. Für den Zugriff auf die OCSP-Response wird ein Schlüssel

[3] repräsentiert die Anzahl der Zertifikaten deren Status nachgefragt wird in einem Client-Request
[4] ein Zertifikat kann jede Zeit revoziert werden bzw. ein unbekanntes Zertifikat kann auch jede Zeit ihren Status ändern, jedoch ein Zertifikat mit dem Status revoked bleibt für immer revoziert bis sein Gültigkeitsdatum erlischt

benötigt, der setzt sich zusammen aus: einem *String*, der *Zertifikatsseriennummer*, dem *issuerNameHash* und dem *issuerKey-Hash*. Der String im Schlüssel wird auf *public* gesetzt, falls das Zertifikat veröffentlicht werden darf und der Client es in seinem Request verlangt hat, ansonsten ist der String gleich *null*, d.h der Schlüssel besteht nur aus den drei restlichen Komponenten (Zertifikatsseriennummer, issuerNameHash und issuerKey-Hash)

- **sigRevokedRespCache**: dieses LRUMap ist genau so aufgebaut wie der sigGoodRespCache, der einzige Unterschied ist der Status der OCSP-Response der in diesem Fall *revoked* ist. Der Zugriff auf eine OCSP-Response wird durch einen Schlüssel realisiert, der gleich gebildet ist wie der Schlüssel von sigGoodRespCache

- **sigUnknownRespCache**: ist auch als LRUMap realisiert worden und beinhaltet Informationen zur OCSP-Response mit dem Status *unknown*. Wenn der Zertifikatsstatus unbekannt ist bedeutet dies, dass das Zertifikat im Certificate-Cache und im Master LDAP nicht vorhanden ist. Der Schlüssel für diese Tabelle sieht genauso aus wie beim sigRevokedRespCache und sigGoodRespCache

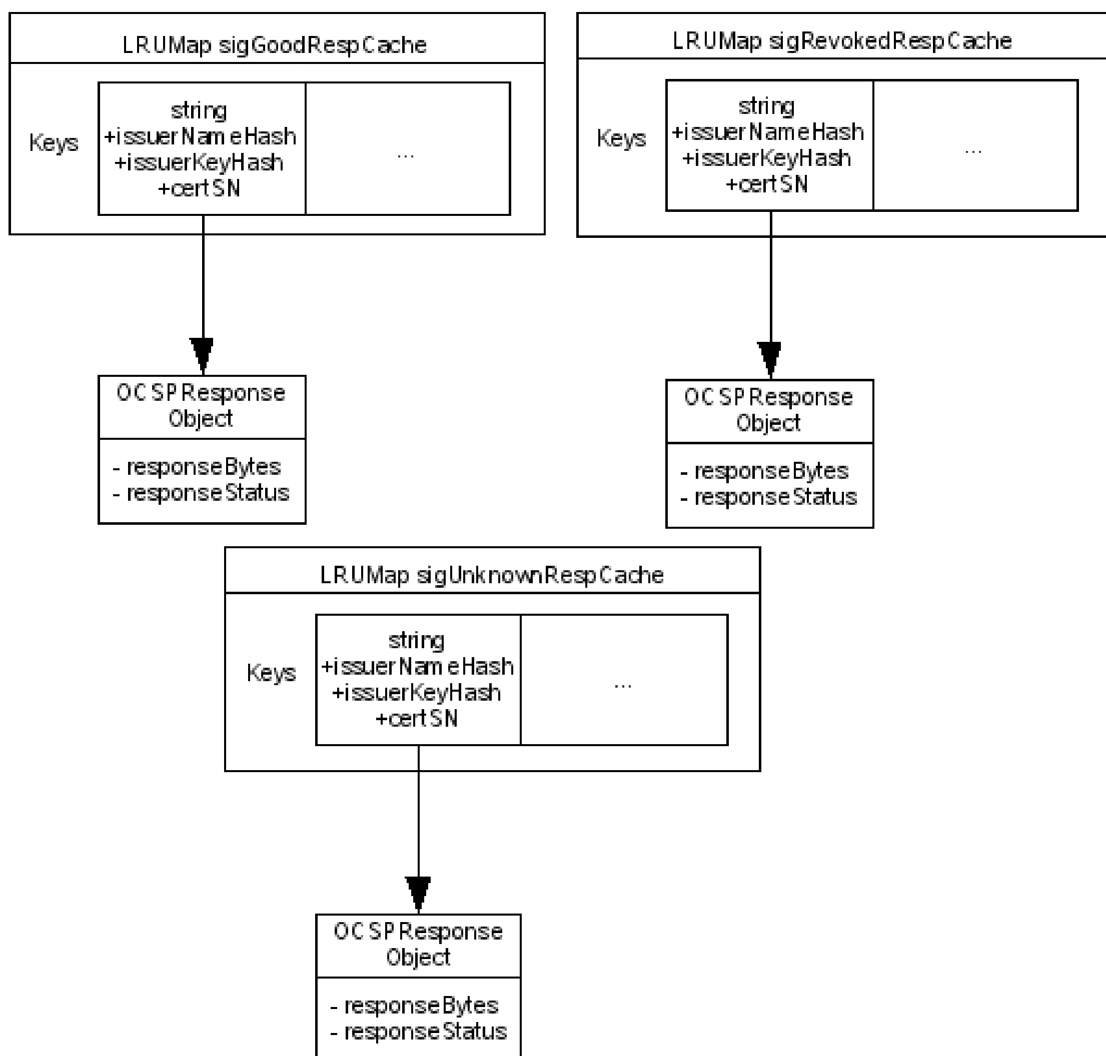

Abbildung 4.6: Datenstruktur des SignedResponsecaches

Analog zu dem Zertifikatscache, ist der SignedResponsecache am Anfang leer und wird dann im Verlauf der Client-Request-Beantwortung aufgefüllt. Es wird im *Property file* eine einheitliche Größe zugewiesen. Die Größe des SignedResponsecache definiert dann die Anzahl der Einträge. Die restlichen Caches sind davon nicht betroffen.

Der SignedResponsecache trägt zur Verbesserung der Performanz erheblich bei. Da die Signierzeit bei einem Hit Treffer im Cache gleich Null ist, wird der Request viel schneller bearbeitet und somit wird die Auslastung des OCSP-Responders verringert.

4.2.5 Ersetzungsstrategie

Die Anzahl der Einträge des SignedResponse- und Zertifikatcache kann im Property-File des OCSP-Responders je nach Entscheidung des Entwicklers beliebig festgelegt werden. Der gewünschte Wert ist im entsprechenden Feld *ResponseCacheSize* in der *ocsp.properties* Datei (siehe Kapitel 5.3.1) einzutragen. Wird die Anzahl der Cache-Einträge erreicht, ist der ResponseCache voll und somit die Speicherung weiterer Einträge nicht möglich. Um die Aufnahme immer neue OCSP-Response Einträge im Cache zu ermöglichen, ohne dass der Cache weiter wächst, wurde eine Cache-Ersetzungsstrategie ausgewählt und implementiert.
Die Wahl der Strategie spielt eine sehr wichtige Rolle in der Effizienz der Cache-Anwendung. Aus diesem Grunde wurde ein *Interface* entworfen, das dem Entwickler erlaubt immer neue Strategien zu implementieren und im OCSP-Responder einzufügen, ohne große Aufwand und ohne sich mit dem gesamten Code zu beschäftigen. Es reicht die Funktionsweise des Strategie-Interfaces des OCSP-Response-Cache zu verstehen. Die genaue Beschreibung des Strategie-Interface wird im Kapitel 5 erläutert, dort wird auch erklärt, wie man das Interface implementieren kann und was dabei zu beachten ist.
Zusätzlich zur Interface-Klasse wurde eine Faktory-Klasse implementiert, die angibt welche Ersetzungsstrategie den OCSP-Responder verwenden soll, falls mehrere Strategien vorhanden sind. Die gewählte Strategie wird von dem Benutzer in der Property-File des OCSP-Responders angegeben, der gewünschte Wert wird dann dem Eintrag *ReplacementStrategy* in der *ocsp.properties* Datei zugewiesen. Als Default-Wert ist die *Least Recently Used Algorithm* (LRU), die im OCSP-Responder implementiert wurde, gesetzt.
Diese Konfigurierbarkeit erlaubt die einfache Implementierung einer neuen Ersetzungsstrategie und das leichte Wechseln der vorhandenen Strategien.

Die Entscheidung für die LRU-Strategie war nicht schwer zu treffen, da im Package commons-collections-3.1.jar diese Ersetzungsstrategie schon im Form von LRUMaps implementiert und bereit für die Anwendung ist. Allerdings hat diese Entscheidung auch ein kleinerer Nachteil im Bezug auf das Design des OCSP-Responders, der wie folgt erklärt wird: die Cache-Struktur wurde der Strategie entsprechend als LRUMaps realisiert, die Maps müssen in der Klasse wo die LRU-Strategie implementiert wird initialisiert. Das hat zur Folge, dass bei der Implementierunge neue Strategien müssen dementsprechend die Datenstrukturen des Caches (in diesem Fall LRUMaps) neu deklarieren. Anders gesehen ist das auch ein Vorteil, da der Entwickler immer die freie Wahl hat, die passende Struktur zu seiner Strategie zu bestimmen. Dazu bleiben die anderen vorhandenen

Ersetzungsstrategien im OCSP-Responder ohne jegliche Änderungen im Code funktionsbereit.
Alle Details dazu, wie diese Konzepte implementiert wurden, werden im Kapitel 5 ausführlich erklärt.

4.3 Interne Abläufe

Zu den internen Abläufen gehören zum einen, die Schritte bei der Verarbeitung eines OCSP-Requests, die entweder von einem externen Client oder von der IS (Infrastructure Services Komponente) in Form von Trigger-Requests kommt, und zum anderen, das regelmäßige Durchsuchen des Master LDAPs nach Veränderungen inkl. der automatischen Aktualisierung des Caches.

4.3.1 Verarbeitung eines OCSP-Requests

In diesem Abschnitt werden alle Abläufe bei der Verarbeitung eines OCSP-Requests, von der Annahme des Requests bis zum Verschicken der Response an den Client vorgestellt.
In den Abbildungen 4.7 und 4.8 sind die Schritte bei einem Standardrequest (d. h. kein Triggerrequest der IS und kein privilegierter Request eines Testclients) als Flußdiagramm dargestellt.
Die Abläufe bei einem Triggerrequest werden in Abschnitt 4.3.2 näher erläutert.

Privilegierte Requests unterscheiden sich nur an zwei Stellen von den normalen Requests:

1. Im OCSP-Responder-Servlet wird anhand der IP-Adresse festgestellt, dass es sich um einen privilegierten Request handelt und die Zugriffszeit gespeichert. Falls der letzte Request von dieser Adresse innerhalb eines konfigurierbaren Intervalls gestellt worden ist, wird er abgelehnt, um keinen Denial-of-Service-Angriff mit gefälschten IP-Adressen zu ermöglichen. Falls ein ausreichender Abstand zwischen den Requests war, wird ein Flag für die weitere Verarbeitung gesetzt.

2. Im SignerAccess-Service wird für den Fall, dass alle Signer des Mandanten belegt sind, das Flag ausgewertet und bei privilegierten Requests wird für den aktuellen Thread die Priorität auf den maximalen Wert gesetzt. Dadurch wird versucht zu erreichen, dass der Scheduler diesen Thread auswählt, sobald wieder ein Signer verfügbar wird. Es gibt allerdings keine Garantie dafür, da es erstens noch einen weiteren Thread mit der maximalen Priorität geben könnte und zweitens die Implementierung des Thread-Schedulings von Java Plattformabhängig ist. So wird die Thread-Priorität unter Linux komplett ignoriert [5], unter Solaris und Windows dagegen funktioniert die Priorisierung.

Bei den Darstellungen in den Abbildungen 4.7 und 4.8 ist an der linken Seite vermerkt, in welcher Klasse, der Schritt durchgeführt wird. Weggelassen wurden dabei Zwischenschritte in Klassen, die im Wesentlichen nur dazu dienen, die Parameter durchzureichen. So geht der Aufruf von getSignerCertificate() in der Klasse RequestProcessorEJB (letzter Schritt in Abbildung 4.7) zum Beispiel den folgenden Weg: CryptoProcessor –> CryptoProcessorEJB –> ServiceHelper –> SignerAccess.

[5]Getestet unter einem 2.6.17er Linux Kernel mit NPTL 2.3.6 und Java 1.4.2_12 im Server-Mode. Siehe hierzu auch: http://$bugs.sun.com/bugdatabase/view_bug.do?bug_id = 4813310$

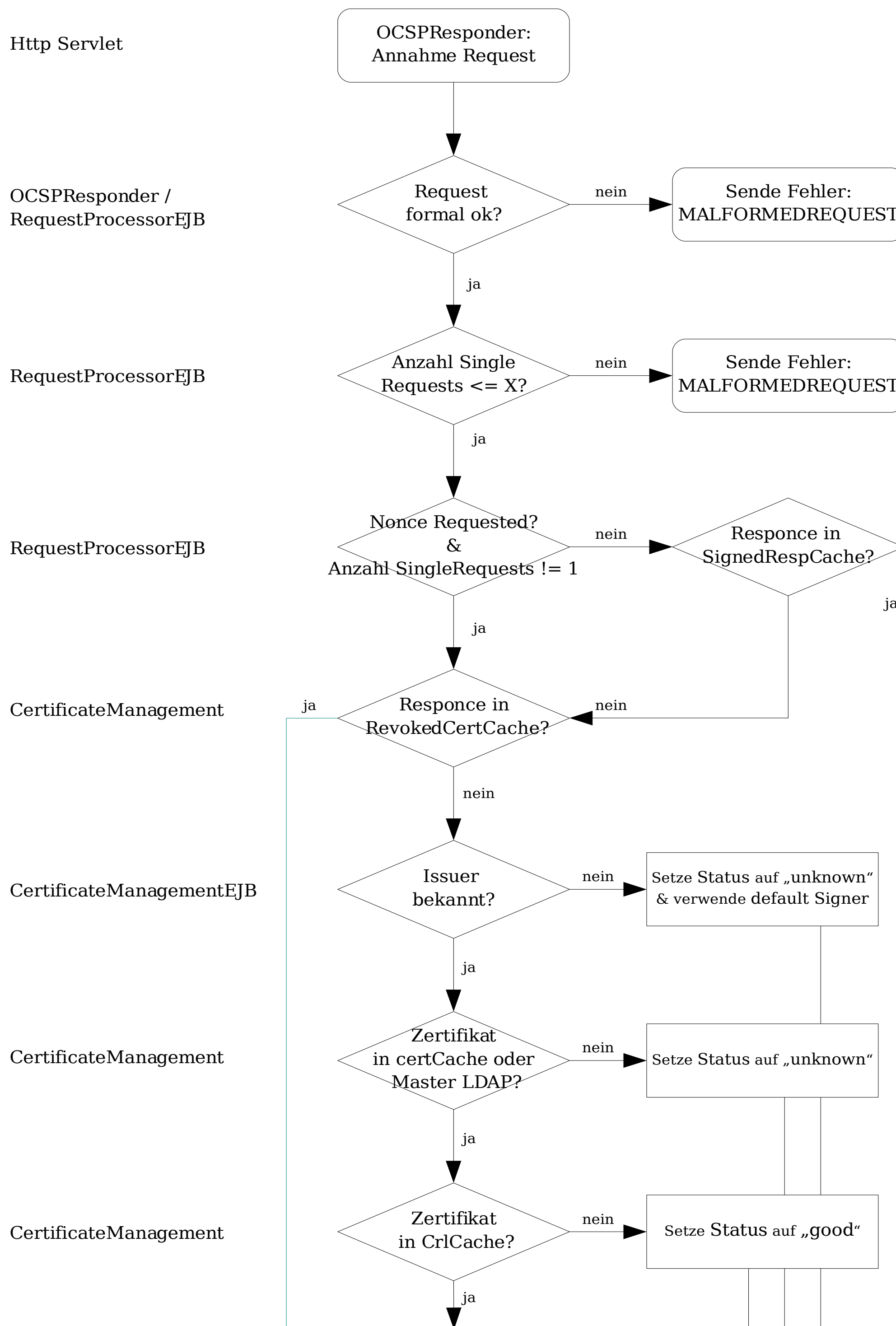

Abbildung 4.7: Ablauf bei einem Standard-Request -Teil1-

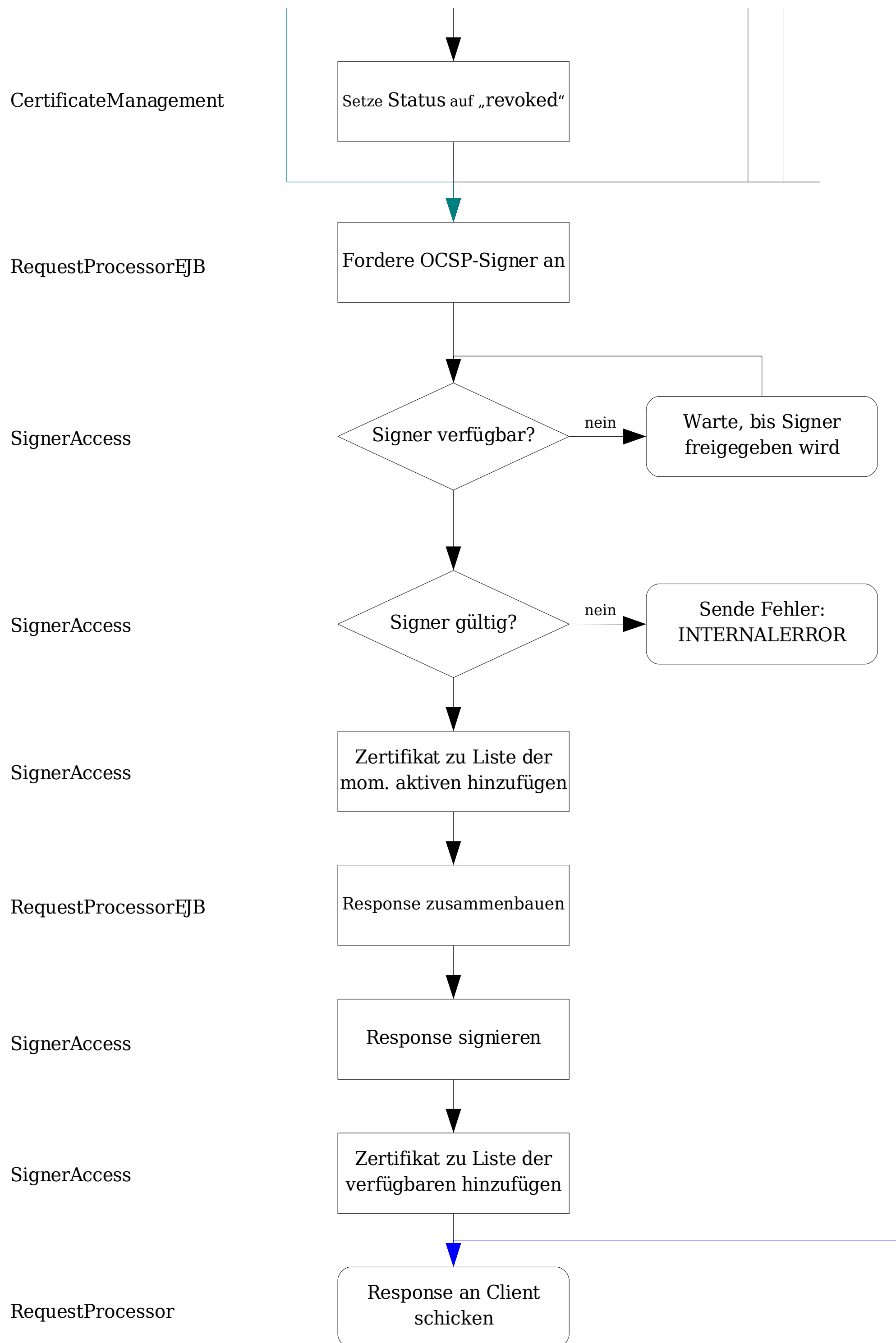

Abbildung 4.8: Ablauf bei einem Standard-Request -Teil 2-

Die Beantwortung eines OCSP-Requests beginnt im OCSP-Responder-Servlet, in dem der HTTP-Request entgegengenommen wird. Um die restliche Verarbeitung besser zu verstehen, wird sie in fünf umfassenden Schritten erklärt:

Schritt 1: Überprüfung der syntaktische Korrektheit und der maxRequests:

- ist der Request formal korrekt, und die maxRequests[6] nicht überschritten –> Schritt 2
- ist der Request formal nicht korrekt oder die maxRequests überschritten, wird eine nicht signierte Response mit der Meldung „MALFORMEDREQUEST“ an den Client geschickt

Schritt 2: Überprüfung der Nonce-Extension und der maxRequests:

- sind die beiden Bedingungen [7] erfüllt, so wird der Schlüssel des angefragten Zertifikates berechnet (siehe Abb. 4.6) und anschließend im SignedResponsecache gesucht
- wurde jedoch ein Treffer im Cache gefunden dann wird es an den Client ohne weiteres geschickt
- tritt eine der beiden Bedingungen nicht ein oder wurde keine Response im SignedResponsecache gefunden –> Schritt 3

Schritt 3: Berechnung der Schlüssel der angefragten Zertifikate (siehe Abb. 4.5), für die Suche im Zertifikatscache. Zuerst wird der revokedCertCache danach der certCache durchsucht:

- wurde das Zertifikat im Cache gefunden so wird die Response aufgebaut und zum signieren bereitgestellt –> Schritt 5
- wurde jedoch kein Zertifikat im Zertifikatscache gefunden –> Schritt 4

Schritt 4: Das suchen des Zertifikates im LDAP, die Ermittlung der Status und die Speicherung der Objekte:

- zuerst wird das Zertifikat im Master LDAP gesucht
- der Status des Zertifikates ermitteln
- Aufbau der Response und die Bereitstellung zur Signatur
- ist der Status *good*, wird das Zertifikat im *certCache* gespeichert
- ist der Status *revoked*, wird das Response-Objekt im revokedCertCache gespeichert
- ist der Status *unknown*, wird nichts gespeichert

Schritt 5 Response an den Client verschicken und ggf. im Cache speichern:

- ist ein Signer verfügbar und gültig wird die Response signiert und an den Client verschickt. sind die entsprechenden Bedingungen erfüllt, erfolgt die Speicherung im SignedResponsecache
- ist der Signer ungültig, so bekommt der Client die nicht signierte Fehlermeldung „INTERNALERROR“ als Response und es wird nichts im Cache gespeichert.

[6] Maximale Anzahl von SingleRequests in einem OCSP-Request.
[7] die Bedingungen sind zwei: Nonce-Extension gleich null und maxRequests gleich eins.

Die Aktualisierung der CRLcache geschieht durch das Triggern. Die beiden Caches Zertifikats- und SignedResponsecache werden im Laufe der Bearbeitung der Anfrage aktualisiert. Der SignedResponsecache wird wie folgt auf den neuesten Stand gebracht: immer wenn eine OCSP-Response vom Cache geholt wird, wird der aktuelle Status des Zertifikats abgerufen bevor es zum Client geschickt wird. Hat sich der Status nicht geändert wird die signierte Response an den Client ohne weiteres verschickt, ansonsten wird der Eintrag vom Cache gelöscht. Eine neue OCSP-Response wird erstellt und an den Client verschickt, gleichzeitig wird die neue Response im Cache gespeichert. Auf diese Weise wird der CRLcache mit Hilfe des Triggers immer auf den aktuellsten Stand gebracht, und der Aufwand des Cache-Updates bleibt gering. Hier ist gut zu sehen wie wichtig die Aufteilung der SignedResponsecache in drei verschiedenen unabhängigen Caches ist.
Das updaten des Zertifikatscaches basiert auf die Abfrage des aktuellen Status des Zertifikats. Analog zum sigRevokedRespCache, wenn sich ein Zertifikat im revokedCertCache befindet, gibt es keine Möglichkeit mehr dieses vom Cache zu löschen, außer das Zertifikat ist abgelaufen, oder es wird durch die Ersetzungsstrategie gelöscht. Beim certCache wird immer der Status abgeprüft, da ein Zertifikat jede Zeit revoziert werden kann. Hat sich der Status geändert, d.h. das Zertifikat ist revoziert worden, so wird es vom certCache gelöscht und das dazugehörige Response-Objekt im revokedCertCache gespeichert.

4.3.2 Triggern

Mit Triggern wird das Anstoßen einer Aktualisierung des CRLcaches des OCSPs bezeichnet. Die Änderungen werden vom Master LDAP eingelesen. Das Triggern kann entweder extern durch die IS in Form von speziellen OCSP-Requests erfolgen oder durch einen Dienst des OCSPs, der aktiv nach Änderungen im Master LDAP scannt.

Externes Triggern durch die IS

Beim externen Triggern sendet die IS spezielle Requests an den OCSP, um zu überprüfen, ob die Aktivierung erfolgreich war und um den OCSP eine Änderung des Master LDAPs mitzuteilen. Diese Trigger-Requests können von der Produktions- oder der Revokations-IS kommen. Unterschieden werden die beiden anhand der IP-Adresse.
Damit der CRL-Cache aktualisiert wird, müssen folgende Überprüfungen erfolgreich durchgeführt worden sein:

- IP-Adresse muss von Revokations-IS sein
- Als Content-Type muss „application/is-ocsp-request“ gesetzt sein
- Der Request darf nur eine bestimmte maximale Länge haben

Sind diese Bedingungen erfüllt, wird die CRL des entsprechenden Mandanten vollständig neu eingelesen (da es sich um eine Hashtable handelt, werden dadurch keine Daten doppelt geschrieben). Die Abläufe bei Trigger-Requests sind sonst identisch zu denen in Abschnitt 4.3.1.

Internes Triggern

Das interne Triggern ermöglicht eine Benutzung des OCSP-Responders ohne die IS. Es wird dabei in regelmäßigen Abständen (Intervall konfigurierbar) nach Veränderungen auf dem Master LDAP gesucht. Der „InternalTrigger-Service“ ist als MBean realisiert und wird vom Scheduler-Service von JBoss gesteuert.
Veränderte Knoten werden anhand der operationalen Attribute modifyTimestamp und createTimestamp erkannt. Alle Knoten, bei denen einer dieser beiden Timestamps größer ist, als der Timestamp der letzten Überprüfung, werden eingelesen und verarbeitet.
Befindet sich in dem Knoten eine CRL, wird deren Issuer und dadurch auch die clientID ermittelt. Damit lässt sich das Aktualisieren des CRL-Caches genau wie beim externen Triggern anstoßen.
Nachdem alle Änderungen abgearbeitet wurden, merkt sich der InternalTrigger den größten gefundenen Timestamp und wartet dann auf seinen nächsten Aufruf.

Kapitel 5

Implementierung und Evaluierung

In diesem Abschnitt wird die Implementierung, das im Kapitel 4 vorgestellten Design und Entwurf erläutert. Es werden ausgewählte Klassen und deren Methoden vorgestellt, die die Implementierung des Caches sowie die interne Abläufe im Bezug auf die Beantwortung des Requests beschrieben. Zudem wird der OCSP evaluiert und bewertet und es werden Testfälle sowie dessen Ergebnisse beschrieben. Aus diesen wird ein Vergleich zwischen den beiden OCSPs (mit und ohne Cache) durchgeführt. Zuletzt werden die wichtigen Hinweise zur Installation und Konfiguration der Software ausgeführt.

5.1 Implementierung

5.1.1 Realisierung des Cache-Konzepts

Der Cache wurde so konzipiert, dass zwei unabhängige Klassen geschrieben wurden, die jeweils der *CertificateCache* und *ResponseCache* implementieren. Zudem wurde eine Klasse entworfen, die beide Caches verwaltet.

CacheManagement

Die Klasse *CacheManagement* wie in Abb. 5.1 dargestellt, ist als Singleton Pattern implementiert und verwaltet den CertificateCache und ResponseCache.
Die Methode *getCertObject(in,in,in)* sucht nach den entsprechenden Zertifikate im CertificateCache und liefert es zurück, falls es vorhanden ist, ist es nicht vorhanden geht die Suche eine Stufe tiefer zum Master LDAP.
Die Methode *getRevokedRespObject(in)* sucht nach revozierte Zertifikat im CertificateCache und liefert es zurück falls es vorhanden ist, andernfalls liefert sie *null* zurück.
Die Methode *getSignedRespObject(in,in)* sucht nach signierten Responses im ResponseCache.
Analog zur Suche, fügen die Methoden *getCertObject(in,in,in)*, *getRevokedRespObject(in)* und *getSignedRespObject(in,in)* die entsprechende Einträge im jeweiligen Cache ein.

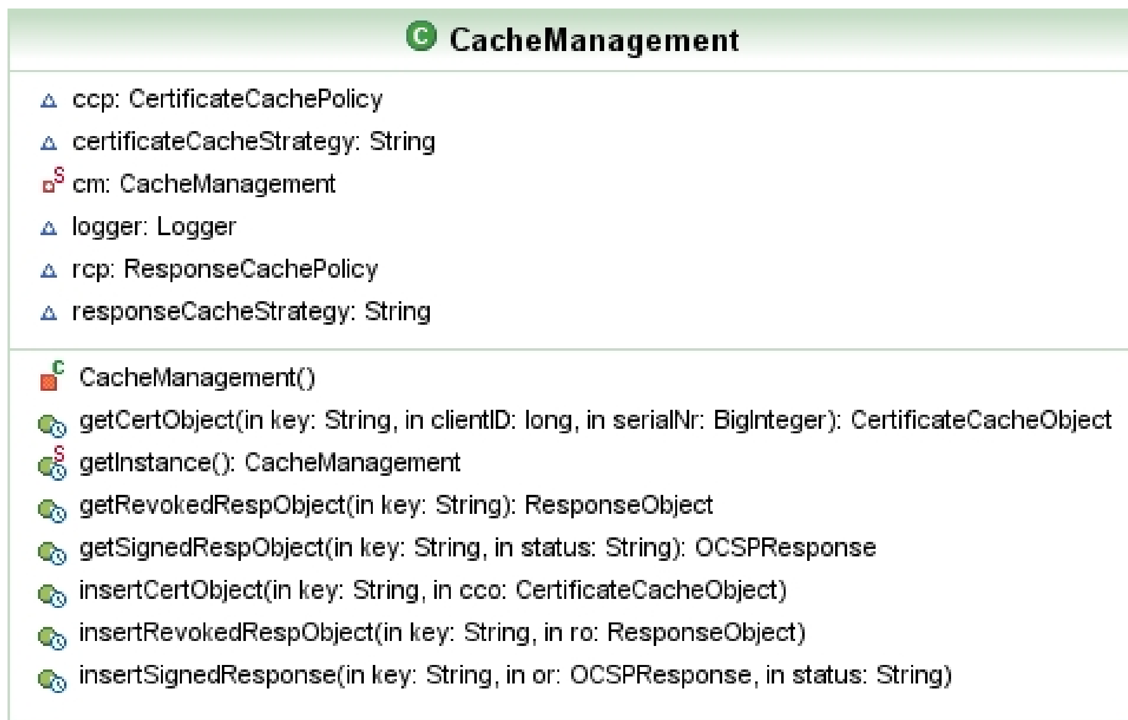

Abbildung 5.1: Die Klasse CacheManagement mit ihren Instanzen und Methoden

CertificateCache

Für die Implementierung des CertificateCaches wurde ein Interface entworfen, das *CertificateCachePolicy*. Dieses Interface wird von der Klasse *CertificateCache* implementiert wie in Abb. 5.2 dargestellt ist. Die Klasse CertificateCache realisiert die zwei Hauptteilen des Caches, der Cache selbst und die Ersetzungsstrategie. Aus diesem Grund wurde das Interface so entworfen, dass der Entwickler die Möglichkeit hat, immer neue Strategien zu implementieren und die dafür notwendigen Datenstrukturen zu wählen und zu verwenden.

Es werden in der Klasse CertificateCache zwei Caches im Form von *LRUmaps* definiert: *certCache* und *revokedCertCache*. Die LRUmaps Struktur wurde verwendet aufgrund der Wahl der LRU-Strategie.
Die Methoden:

- *getCertificateCacheObejct(in)* und *getRevokedRespCacheObejct(in)* liefern die entsprechende Objekte zurück, falls sie im Cache gespeichert sind andernfalls liefern sie *null* zurück

- *insertCertificateCacheObject(in,in)* und *insertRevokedRespObeject(in,in)* speichern die Einträge, in den dafür geeigneten Caches (*certCache* oder *revokedCertCache*).
  Bei jedem Eintrag der vom certCache geholt wird, wird der Status anhand der vorhandene CRL geprüft und zur Weiterbearbeitung freigegeben. Der im revokedCertCache gespeicherte Eintrag, kann seinen Status nicht mehr ändern. Daher ist die Aktualisierung des CertificateCaches trivial: immer wenn ein neuer Eintrag im revokedCertCache gespeichert wird, wird er vom certCache gelöscht, falls er dort gespeichert war

Abbildung 5.2: Klassendiagramm CertificateCachePolicy und CertificateCache

ResponseCache

Der ResponseCache, steht für die Speicherung und Wiederverwendung, die vom OCSP signierte Responses. Er besteht aus den drei Caches im Form von LRUmaps: *sigGoodRespCache*, *sigRevokedRespCache* und *sigUnknownRespCache*. Die Einträge werden je nach Status im entsprechenden Cache gespeichert. Analog zum CertificateCache wurden die LRUmaps aufgrund der Ersetzungsstrategie verwendet.
Um dem Entwickler die Möglichkeit zu bieten, eine andere Ersetzungsstrategie zu implementieren oder andere Datenstruktur für den Cache zu verwenden, ohne viele Änderungen in den Code vorzunehmen, wurde eine abstrakte Klasse *ResponseCachePolicy* entworfen. Diese Klasse implementiert die *LRUStrategie*, wie in Abb. 5.3 dargestellt ist. Die Klasse LRUStrategie realisiert die beiden Bestandteile eines Cache-Konzepts und zwar der Cache selbst als *maps* und die LRU Ersetzungsstrategie.

Die Methode *insertRespCacheObject(in,in,in)* speichert die OCSP-Response Objekte in dem entsprechenden Cache.
Die Methode *getRespCacheObject(in,in,in)* liefert das abgefragte Objekt falls es im Cache vorhanden ist zurück oder sie liefert *null* zurück.

Wenn der Status einer Response good oder revoked ist und diese Response sich im sigUnknownRespCache befindet, so wird das ResponseObjekt vom sigUnknownRespCache gelöscht. Analog zum sigUnknownRespCache werden die ungültige Einträge vom sigGoodRespCache gelöscht, wenn sich das ResponseObject im sigRevokedRespCache befindet. Für den sigRevokedRespCache ist kein Eingriff notwendig, da er nur aus Responses für revozierte Zertifikate besteht und deren Status sich nicht mehr ändert. Mit diesen Maßnahmen wird versucht den Responsecache zu optimieren, indem die Redundanz beseitigt wird.

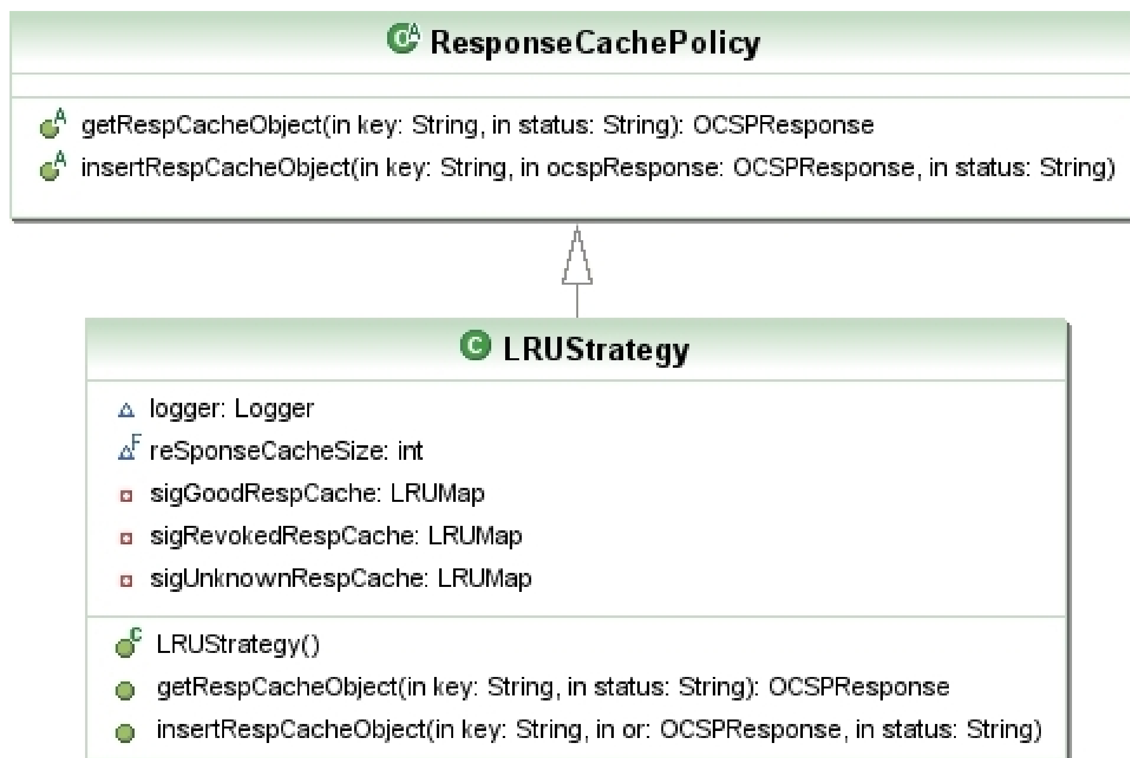

Abbildung 5.3: Klassendiagramm für die ResponseCachePolicy Klasse und deren LRU-Implementierung

Wahl der Cache Strategie

Zu jeder Cache Implementierung gehört eine Ersetzungsstrategie, die es erlaubt, die Einträge auszutauschen sobald der Cache voll ist. Es wurde wie im Kapitel 4 erläutert wurde, sich für die LRUStrategie entschieden. Jedoch wurde das Konzept so entwickelt, dass für den Entwickler, jede Zeit die Möglichkeit besteht, eine andere Strategie zu implementieren (mehr dazu, in den Abschnitten 5.1.1 und 5.1.1).

Falls der Entwickler mehrere Ersetzungsstrategien implementiert hat, so besteht für den OCSP Betreiber die Möglichkeit[1], eine beliebige [2] Ersetzungsstrategie auszuwählen.
Da der OCSP zwei Hauptcaches beinhaltet, wurden zwei Methoden in der FactoryCachePolicy Klasse verwendet wie in Abb. 5.4 beschrieben ist. Somit ist es möglich für jeden Cache eine andere

[1]bei jedem neuen Start des OCSP-Servers
[2]die Cache Strategie muss vorher implementiert sein, dazu ist sie in der *ocsp.properties*(s. Abschnitt 5.3.1) einzugeben, wird keine Strategie eingegeben, so wird mit der LRUStrategie gearbeitet

Ersetzungsstrategie zu verwenden, falls mehrere implementiert sind. Als default wird die LRU Strategie für die beiden Caches, CertificateCache und ResponseCache benutzt, falls es in der Grundeinstellung nicht anders vorgegeben ist.

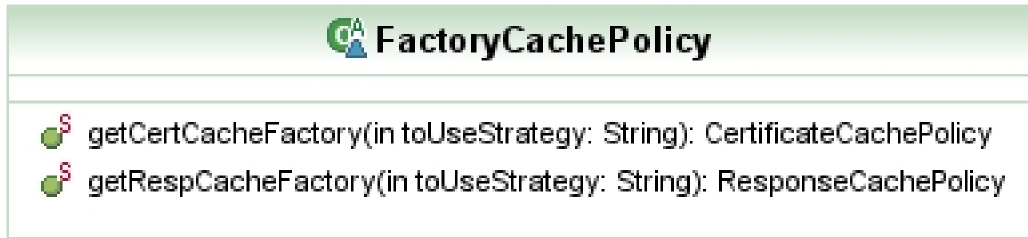

Abbildung 5.4: Die Klasse FactoryCachePolicy mit ihre Instanzen und Methoden

5.1.2 Bearbeitung eines Requests

Ein Request wird von der *Http servlet* des OCSPs entgegen genommen und zum *RequestProcessorEJB* weitergeleitet, für die weitere Bearbeitung. Die Klasse RequestProcessorEJB wie in Abbildung 5.5 dargestellt ist, wurde als EJB implementiert und besteht aus mehreren Methoden die verschiedene Aufgaben erledigen.

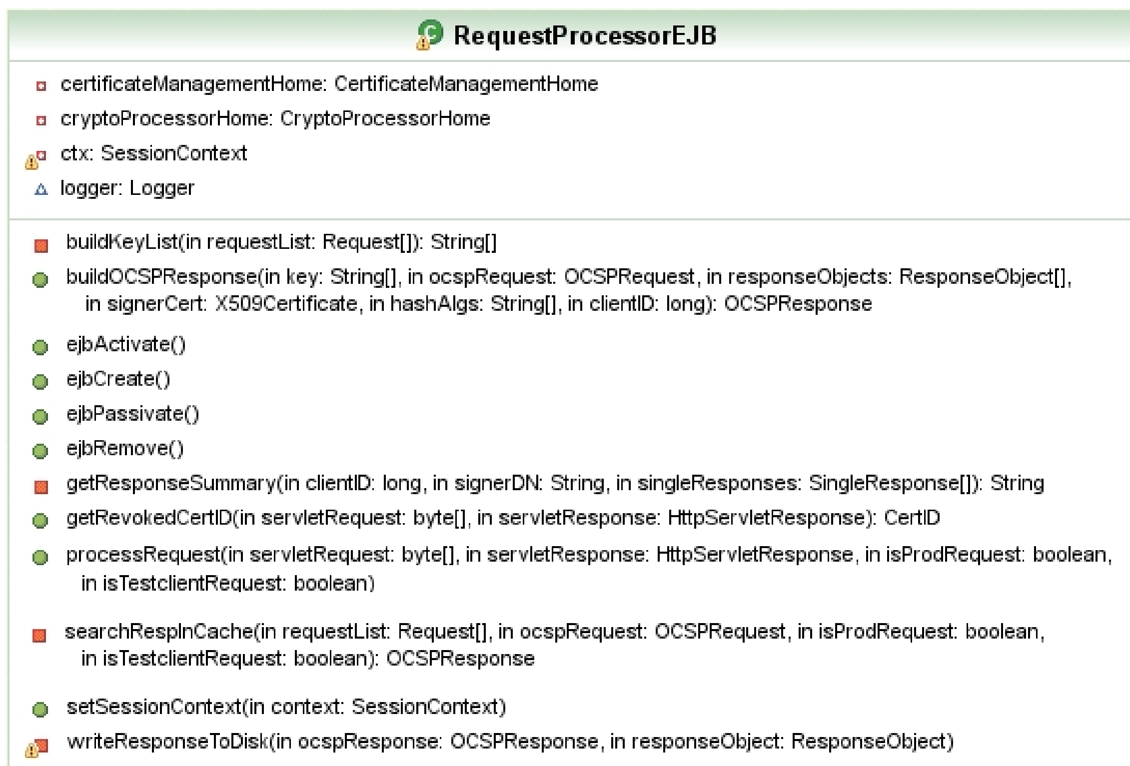

Abbildung 5.5: Die Klasse RequestProcessorEJB mit ihre Instanzen und Methoden

Die Methode `processRequest(in, in, in, in)` prüft zuerst, ob der Request formal korrekt ist und ob die Anzahl der SingleRequests nicht die die vorgegeben sind überschreitet. Sind die beiden Bedingungen erfüllt so wird die Anfrage zur `searchRespInCache(in, in, in, in)` Methode weitergeleitet, sonst wird eine unsignierte *MALFORMEDREQUEST* Antwort an den Client geschickt.

Die Methode `searchRespInCache(in, in, in, in)` prüft ob keine Nonce-Extension vorhanden ist und ob die Anzahl des SingleRequests im OCSP-Request nicht größer als eins ist. Sind die zwei Voraussetzungen erfüllt, so wird die Anfrage zum *CacheManagement* in der *util Package*, mit dem von der Methode `buildKeyList(in)` berechneten Schlüssel weitergeleitet und es erfolgt eine Suche in der ResponseCache. Wenn die Suche einen Erfolg erzielt hat, wird die signierte OCSP-Response an der `processRequest(in, in, in, in)` Methode zurückgeliefert und anschließend zum Client geschickt, andernfalls wird *null* zurückgeliefert und eine Anfrage wird dann an den *CertificateManagement*(Abb. 5.6) für die weitere Bearbeitung weitergeleit.

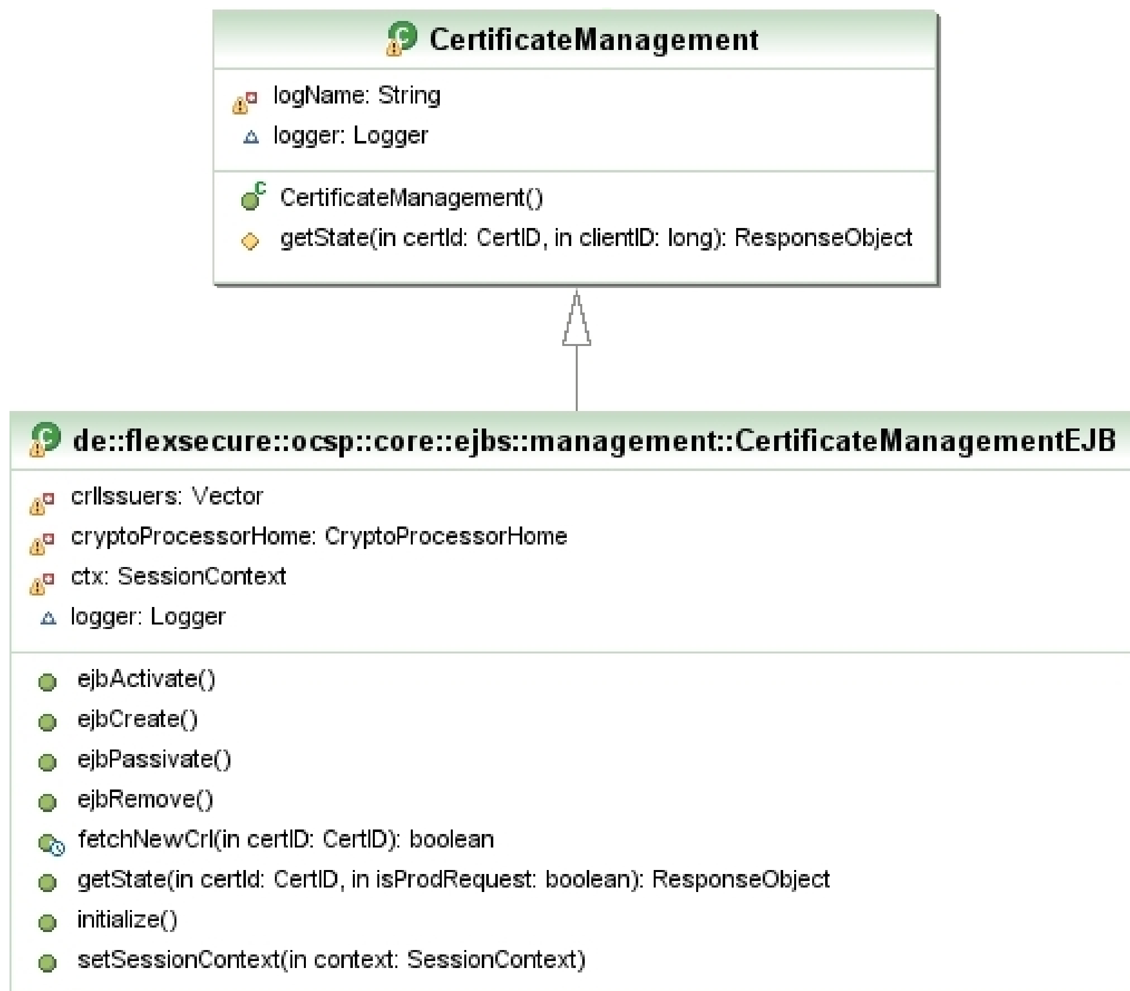

Abbildung 5.6: Die Klasse CertificateManagement mit ihre Instanzen und Methoden

Die Methode `getState(in, in)` in der Klasse CertificateManagement stellt eine Anfrage an den CacheManagement, ob das Zertifikat im *revokedCertCache* vom CertificateCache vorhanden ist oder nicht. Im Fall, dass das Zertifikat im revokedCertCache gespeichert ist, wird es zur Klasse RequestProcessorEJB zurückgeliefert, dort wird das OCSP-Response aufgebaut und signiert. Dazu wird es im ResponseCache gespeichert und anschließend zum Client verschickt. Wurde das Zertifikat im revokedCertCache nicht gefunden, so geht die Suche ein Stufe tiefer, es wird dann im *certCache* vom CertificateCache gesucht. Falls das Zertifikat dort gespeichert ist, so wird dessen Status ermittelt. Dementsprechend wird ein neuer Eintrag im revokedCertCache gespeichert, wenn

der Status revoked ist und das ResponseObject an der RequestProcessorEJB weitergeleitet. Hat die Suche jedoch auch im certCache kein Ergebnis gebracht, wird abschließend vom CacheManagement eine Suche im *Master LDAP* geleitet. Wird das Zertifikat im Master LDAP gefunden, so wird es im certCache gespeichert und die weitere Bearbeitung verläuft analog wie bereits beschrieben wurde.

Die Abbildung 5.7 beschreibt und veranschaulicht die Relationen zwischen ein Teil der Klassen in der Util-Package.

Abbildung 5.7: Ausschnitt aus der Klassen-Beziehungen im Package *de.flexsecure.ocsp.util*

5.2 Evaluierung

In diesem Abschnitt werden die auf den OCSP durchgeführten Tests beschrieben und bewertet. Der OCSP wurde unter dem Betriebsystem Linux openSuse 10.2 mit dem Kernel 2.6.18.8-0.3 und einem 1,73Ghz Intel centrino Prozessor mit 512 MB Arbeitspeicher getestet. Zum Vergleich wurden der OCSP [3] und der erweiterte OCSP[4], unter den gleichen Bedingungen getestet.

5.2.1 Testsdurchführung

Für die Testsdurchführung wurde ein java TestClient geschrieben, der die Anfragen als *openssl* zum Server schickt. Die Responses werden dann anschließend vom Client bewertet.
Die beiden OCSPs verfügen in den durchgeführten Tests nur über *Softtokens* um die Responses zum signieren. Jeder OCSP-Responder unterhält 4 Mandanten.
Der TestClient wurde wie folgt eingestellt: pro RequestList wird eine Anfrage gestellt. Den OCSP-Responder werden maximal 150 Anfragen gleichzeitig gestellt und die Verzögerung zwischen zwei nacheinander gestellten Anfragen beträgt 100msec. Dabei wurde ein Response als nicht erhalten bewertet, wenn die Zeit seit der Absendung des Requests 30 Sekunden überschreitet ohne eine Antwort zu erhalten. Der TestClient erfasst die Zeit zwischen dem Absenden eines Requests und dem Empfangen einer Response, es werden die minimale und maximale Zeiten gespeichert sowie einen Mittelwert aus allen Zeiten ermittelt.
Neben der Zeit, die der Responder braucht um eine Anfrage zu beantworten, sind der Speicherverbrauch und die Zeit, die der OCSP zum Starten braucht ist in dieser Arbeit von großer Bedeutung. Die Response- und Zerttifikatscache wurden in diesen Tests mit dem in dieser Arbeit implementierten Ersetzungsstrategie LRU verwaltet.
In jedem Testfall wird angegeben wie viele Zertifikate angefragt wurden und die Größe des Zertifikat- und des Responsecaches. In allen Testfällen beträgt die Anzahl der Zertifikate im Master LDAP 15066.

Die Testfälle werden in drei Kategorien unterteilt:

Kategorie 1

Das Verhalten bezüglich der Startzeit und der Speicherverbrauch des OCSP-Responders.

Testfall 1

Beschreibung Es werden 15066 Zertifikate und vier CRLs (für jeden Manadant eine CRL) im Master LDAP geladen, danach werden die beiden OCSPs nacheinander gestartet. Dieser Test wurde 5 mal durchgeführt um die Schwankungen der Testrechner zu berücksichtigen.

Ziel: Bestimmung der Dauer des Startvorganges, sowie der Speicherverbrauch des OCSPs.

[3] Der OCSP, der von der Firma Flexsecure [flexsec] bereitgestellt wurde, der als Basis für den in dieser Diplomarbeit erweiterten OCSP gilt.
[4] der in dieser Arbeit entworfen und implementiert wurde

Erwartung: Der optimierte OCSP soll viel schneller betriebsbereit sein als der Basis-OCSP. Dazu soll der optimierte OCSP-Responder weniger Speicher verbrauchen als der Basis-OCSP-Responder

Ergebnis

- Basis-OCSP: für den Start wurden in der besten Messung 3min:14.978sec und im langsamsten Start wurden 3min:29.488sec benötigt. Beim Speicherverbrauch haben die Messungen die folgende Werte geliefert: das Minimum war 222mb und das Maximum war 265mb, als Mittelwert wurde 248mb ermittelt.
- Optimierte OCSP: bei Betrachtung des Startvorganges, wurden in den fünf Fällen verschiede Werte gemessen, der minimal Wert war 18.226sec und der maximal Wert war 34.623sec. Als Mittelwert wurde der Wert 24.919sec ermittelt. Der Speicherverbrauch lag zwischen 93mb und 96mb beobachtet.

An den Zahlen kann man beobachten wie die Erwartungen der OCSP Optmierung erfüllt worden sind.

Kategorie 2

Das Verhalten des Zertifikatscaches des optimierten OCSP-Responders im Bezug auf die Antwortszeit. Es werden 3 Testfälle erzeugt und durchgeführt.

Testfall 2

Beschreibung Es wird von 5000 Zertifikaten jeweils der Status abgefragt. 4000 Zertifikate haben entweder den Status *good* oder *revoked* und die restlichen 1000 sind *unknown*. Die Größe der Zerttifikatscache des optimierten Responders beträgt 5000 Einträge. Es wurde einen Durchlauf für den Basis-OCSP gemacht und zwei für den optimierten OCSP. Beim optimierten OCSP wurde der erste Durchlauf mit einem leeren Zerttifikatscache durchgeführt und der zweite Durchlauf mit einem vollen Zerttifikatscache, der jedoch keines der angefragten Zertifikate enthält.

Ziel: Der „worst case“ des optimierten OCSP-Responders darzustellen.

Erwartung: Der optimierte OCSP soll die Requests langsamer beantworten als der Basis-OCSP.

Ergebnis

- Basis-OCSP: die minimale und maximale Bearbeitungszeit waren jeweils 127msec und 18581msec, der Mittelwert lag bei 814msec. Die gesamte Bearbeitungsdauer betrug 776626msec
- Optimierte OCSP:
  - Durchlauf 1 Cache ist leer: die minimale und maximale Bearbeitungszeit waren jeweils 144msec und 15865msec, der Mittelwert lag bei 934msec. Die gesamte Bearbeitungsdauer betrug 871330msec

  - Durchlauf 2 Cache ist voll: die minimale und maximale Bearbeitungszeit waren jeweils 141msec und 19469msec, der Mittelwert lag bei 998msec. Die gesamte Bearbeitungsdauer betrug 862827msec

Die erzielten Ergebnisse zeigen, dass im worst case der Zeitunterschied zwischen dem optimierten und dem Basis-OCSP schwanken zwischen 14 msec und maximal 200msec. Diese Verzögerung im Vergleich zum Basis-OCSP ist unmitellbar nach dem Starten des optimierten OCSP zu beobachten, aber sie wird durch dem schnelleren Start und den wesentlichen geringeren Speicherverbrauch kompensiert. Die Verzögerung kann auch auftreten, wenn die Zerttifikatscachegröße nicht optimal ausgewählt wird.

Testfall 3

Beschreibung Es wird von 5000 Zertifikaten jeweils der Status abgefragt. 4000 Zertifikate haben entweder den Status *good* oder *revoked* und die restlichen 1000 sind *unknown*. Die Größe der Zerttifikatscache des optimierten Responders beträgt 5000 Einträge. Es wurde der selbe Durchlauf für den Basis-OCSP zum Vergleich genommen wie im Testfall 2 und ein weiterer Durchlauf für den optimierten OCSP durchgeführt. Im weiteren Durchlauf befinden sich 50% der angefragten Zertifikate im Zerttifikatscache, 30% sind im Master LDAP und 20% sind unbekannt.

Ziel: Der „average case“ des optimierten OCSP-Responders mit dem Basis-Responder zu vergleichen.

Erwartung: Der optimierte OCSP soll die Requests langsamer beantworten als der Basis-OCSP.

Ergebnis

- Basis-OCSP: gleiche Ergebnisse wie im Testfall 2
- Optimierte OCSP: die minimale und maximale Bearbeitungszeit waren jeweils 139msec und 15641msec, der Mittelwert lag bei 712msec. Die gesamte Bearbeitungsdauer betrug 797731msec

Mit einer 50% Hit-Rate kann man eine bemerkbare Verbesserung im Vergleich zum worst case beobachten. Es wurde ein besserer Mittelwert berechnet als beim Basis-OCSP, jedoch bleibt die gesamte Bearbeitungsdauer ein wenig länger.

Testfall 4

Beschreibung Es wird von 4000 bekannte Zertifikaten jeweils der Status abgefragt. Die Größe der Zerttifikatscache des optimierten Responders beträgt 4000 Einträge. Alle abgefragten Zertifikate befinden sich im Zertifikatscache.

Ziel: Der „best case“ des optimierten OCSP-Responders mit dem Basis-Responder zu vergleichen.

Erwartung: Der optimierte OCSP soll ungefähr genau so schnell die Anfragen beantworten als der Basis-OCSP.

Ergebnis

- Basis-OCSP: die minimale und maximale Bearbeitungszeit waren jeweils 126msec und 10690msec, der Mittelwert lag bei 652msec. Die gesamte Bearbeitungsdauer betrug 630629msec
- Optimierte OCSP: die minimale und maximale Bearbeitungszeit waren jeweils 139msec und 9621msec, der Mittelwert lag bei 683msec. Die gesamte Bearbeitungsdauer betrug 651106msec

Mit einer 100% Hit-Rate gleicht der Zerttifikatscache des optimierten OCSPs, der des Basis-OCSP. Wenn genug Speicher zur Verfügung steht und der Zerttifikatscache im optimierten Responder groß genug definiert wird, ist der Zeitunterschied zum Basis-OCSP kaum bemerkbar.

Kategorie 3

In diesem Abschnitt wird der Einsatz von ResponseCache im optimierten OCSP getestet. Der *worst case* wird in diesem Abschnitt nicht behandelt, da er mit den Ergebnissen der Testfälle von Kategorie 2 identisch ist.

Testfall 5

Beschreibung Es wird von 10000 Zertifikaten jeweils der Status abgefragt. 9000 Zertifikate haben entweder den Status *good* oder *revoked* und die restlichen 1000 sind *unknown*. Die Größe der Zerttifikatscache und der Responsecache des optimierten Responders beträgt 10000 Einträge.

Es wurde ein Durchlauf für den Basis-OCSP gemacht und zwei für den optimierten OCSP. Im ersten Durchlauf des optimierten OCSPs sind 33% der abgefragten Zertifikate im Responsecache, 33% im Zerttifikatscache, 24% sind im Master LDAP und 20% sind *unknown*. Im zweiten Durchlauf befinden sich 50% der abgefragten Zertifikate im Zerttifikatscache und die restlichen 50% im Responsecache.

Ziel: der „average case“ des optimierten OCSP-Responders bei der Einsetzung vom Responsecache.

Erwartung: Der optimierte OCSP soll die Requests schneller beantworten als der Basis-OCSP.

Ergebnis

- Basis-OCSP: die minimale und maximale Bearbeitungszeit waren jeweils 126msec und 17552msec, der Mittelwert lag bei 686msec. Die gesamte Bearbeitungsdauer betrug 1511765msec

- Optimierte OCSP:
  - Durchlauf 1: die minimale und maximale Bearbeitungszeit waren jeweils 14msec und 21769msec, der Mittelwert lag bei 793msec. Die gesamte Bearbeitungsdauer betrug 1499639msec
  - Durchlauf 2: die minimale und maximale Bearbeitungszeit waren jeweils 14msec und 191175msec, der Mittelwert lag bei 605msec. Die gesamte Bearbeitungsdauer betrug 1400131msec

Durch den Zerttifikatscache wird der Speicherverbrauch des OCSPs kontrolliert sowie eine sehr schnelle Startzeit erreicht. Der Responsecache gleicht den Zeitverlust gegenüber dem Zerttifikatscache aus, wenn die Daten aus dem Master LDAP geholt werden. Bei einer guten Hit-Rate liefert der optimierte OCSP bessere Werte als der Basis-OCSP.

Testfall 6

Beschreibung Es wird von 10000 Zertifikaten jeweils der Status abgefragt. 9000 Zertifikate haben entweder den Status *good* oder *revoked* und die restlichen 1000 sind *unknown*. Die Responses für alle abgefragten Zertifikate befinden sich im ResponseCache des optimierten OCSPs.

Ziel: Der „best case“ des optimierten OCSP-Responders bei der Verwendung vom Responsecache.

Erwartung: Der optimierte OCSP soll ungefähr genau so schnell die Anfragen beantworten als der Basis-OCSP.

Ergebnis

- Basis-OCSP: es wurden die gleiche Ergebnisse des Testfalles 5 zum Vergleich genommen
- Optimierte OCSP: die minimale und maximale Bearbeitungszeit waren jeweils 14msec und 652msec, der Mittelwert lag bei 24msec. Die gesamte Bearbeitungsdauer betrug 1050598msec

Je höher die Hit-Rate im Responsecache ist, desto kürzer werden die Antwortzeiten des OCSP-Responders. Der Responder kann somit eine größere Anzahl von Anfragen abarbeiten.

5.2.2 Analysierung der Testergebnisse

Die Ergebnisse die im vorigen Abschnitt erzielt wurden, zeigen die Vorteile, die die Optmierung des OCSP-Servers bringen kann. Anhand der durchgeführten Testfälle, war es zu sehen, wie der erweiterter OCSP sich verhält in bezug auf die vorgegebene Situationen. Eine erhebliche Steigerung der OCSP-Performanz, war im Fall der optimalen Benutzung des Responsecaches deutlich zu sehen, die Beantwortungszeit wurde bis zum fünffache reduziert.
Der Zerttifikatscache hat den weiteren Effekt, neben der Verwaltung des Speicherverbrauches im OCSP, wird die Startzeit erheblich verkürzt, je höher der Anzahl der Zertifikate ist die sich im Master LDAP befinden.
Im Allgemeinen haben die durchgeführten Tests gezeigt, dass der erweiterte OCSP weitaus bessere Performanz hat als die vom Basis-OCSP, wenn die Bedingungen für die Optimale Nutzung erfüllt

werden: möglichst auf die Nonce-Extension zu verzichten, und die RequestList auf eine Anfrage pro Requet zu beschränken. Werden diese Bedingungen nicht eingehalten, entstehen keine Nachteile gegenüber dem Basis-OCSP.

5.3 Installation und Konfiguration

Die Konfiguration des OCSP-Responders besteht aus einer Properties-Datei („ocsp.properties“) sowie diversen XML-Dateien von JBoss.
Von den diversen JBoss-Konfigurationsdateien sind hier nur die beschrieben, die speziell für den OCSP-Responder angepaßt wurden.

5.3.1 ocsp.properties

Die einzelnen Properties sind in der folgenden Tabelle dargestellt:

Property	***Beschreibung***
ocspLogger.logLevel	Log-Level (TRACE, DEBUG, INFO, ...)
ocspLogger.logRotationSize	Max. Größe der Log-Dateien
ocspLogger.logRotationCount	Max. Anzahl der Log-Dateien
ocspLogger.logFile	Dateiname und Pfad der Log-Datei
ocspLogger.logToConsole	Zusätzlich auf Konsole loggen?
ocspLogger.logPasswords	Falls auf true, wird versucht, alle Passwörter unkenntlich zu machen
responderServlet.isProdHost-Adresses	IP-Adresse der Produktions-IS für externes Triggern
responderServlet.isRevocHost-Adresses	IP-Adresse der Revokations-IS für externes Triggern
responderServlet.testclientAdresses	IP-Adresse des Testclients für privilegierte Requests
responderServlet.testclient-MinReqInterval	Min. Wartezeit zwischen zwei privilegierten Anfragen
responderServlet.maxRequestLength	Max. Request-Länge
responderServlet.sendTry-LaterWhenUpdatingCaches	Damit kann die Annahme von neuen Requests verhindert werden, während die Caches aktualisiert werden. Der OCSP antwortet dann mit `TRY_LATER`
maxRequests	Max. Anzahl von SingleRequests in einem OCSP-Request. Bei Überschreitung antwortet der OCSP mit MALFORMEDREQUEST
responseCacheStrategy	Die zu benutzende Strategie für den SignedResponse Cache (siehe Abschnitt 4.2.4) im Fall, dass mehrere Ersetzungsstrategien implementiert sind. Als default wird vom OCSP-Responder die LRU Strategie verwendet
ResponseCacheSize	Max. Anzahl der Maps-Einträge des SignedResponse-Caches

certificateCacheStrategy	Die zu benutzende Strategie für den Certifcate Cache (siehe Abschnitt 4.2.3) im Fall, dass mehrere Ersetzungsstrategien implementiert sind. Als default wird vom OCSP-Responder die LRU Strategie verwendet
certificateCacheSize	Max. Anzahl der Maps-Einträge des Certificate-Caches
sigHashAlgs	Liste von möglichen Hash-Algorithmen für die Signatur der Response. Ausgewählt wird der Algorithmus aus dieser Liste anhand des im Request verwendeten Hash-Algorithmus
responseCertHash	Hash-Algorithmus für die CertHash-Extension
client.X.clientName	Symbolischer Name des Mandanten
certCacheHashAlgs	Liste der Hash-Algorithmen, die für Caches verwendet werden sollen. Damit wird indirekt festgelegt, welche Hash-Algorithmen in den Requests erlaubt sind. Durch die Konfigurierbarkeit lässt sich der Algorithmus sehr einfach wechseln, falls die bisher verwendeten nicht mehr sicher sein sollten. Als Default-Werte sind gemäß ISIS-MTT SHA-1, RIPEMD160 und MD5 gesetzt
client.X.clientID	ClientID des Mandanten; das „X“ ist eine aufsteigende Nummer, beginnend bei 1 für den ersten Mandanten in der Properties-Datei
client.X.validityModel	Gültigkeitsmodell (`CHAIN oder SHELL`); wird zum Beispiel für die Überprüfung der Gültigkeit des OCSP-Signers verwendet
client.X.pinsharingDir	Verzeichnis mit den PIN-Shares des Mandanten X; wenn die Zeile auskommentiert ist, wird kein PIN-Sharing verwendet
client.X.signerCerts.crtCa	Verzeichnis mit CA-Zertifikaten für den Issuer-Cache (s. Abschnitt 4.2.2)
client.X.signerCerts.crlCa	Verzeichnis mit CRL-Signer-Issuer-Zertifikaten für den Issuer-Cache (s.Abschnitt 4.2.2)
client.X.signerCerts.ocspCa	Verzeichnis mit OCSP-Signer-Issuer-Zertifikaten für den Issuer-Cache (s.Abschnitt 4.2.2)
client.X.signerCerts.crl	Verzeichnis mit CRL-Signer-Zertifikaten für den Issuer-Cache (s. Abschnitt 4.2.2)
client.X.signerCerts.ocsp	Verzeichnis mit OCSP-Signer-Zertifikaten für den Issuer-Cache (s. Abschnitt 4.2.2)
ldapAccess.ldapHost	IP-Adresse oder Domain-Name des Master LDAP-Servers
ldapAccess.ldapPort	Port des Master LDAP-Servers
ldapAccess.baseDN	Root-DN des Master LDAP-Servers
ldapAccess.keystore	Keystore zur Verifizierung des SSL-Zertifikats des Master LDAP
ldapAccess.addresses	Adresse für JNDI-Lookups

fsService.addresses	Adresse für JNDI-Lookups
signerAccessService.addresses	Adresse für JNDI-Lookups
cardManager.drivers	Treiber-String für Cardman-Bibliothek
cardManager.hotplug	Hotplugging für Signaturkarten ein-/ausschalten
cardManager.pin	PIN, falls kein PIN-Sharing verwendet wird
cardManager.smKey	SM-Key, falls kein PIN-Sharing verwendet wird
security.providers	Java-Security-Provider (Komma-separiert)
jnpProviderUrlEJB	JNDI Host and Port für Lookups von EJBs
jnpPortMBeans	JNDI Host and Port für Lookups von MBeans

Tabelle 5.2: Beschreibung der ocsp.properties

5.3.2 scheduler-service.xml

In der Konfigurationsdatei des Scheduler-Services von JBoss wird u. a. festgelegt, wie oft der InternalTrigger- Service ausgeführt werden soll.

In der folgenden Tabelle sind die einzelnen Parameter der scheduler-service.xml näher erläutert:

Attribut	***Beschreibung***
StartAtStartup	Falls hier false eingetragen wird, muß explizit startSchedule aufgerufen werden, damit der Scheduler mit der Arbeit beginnt
SchedulableMBean	Auszuführendes MBean (voll qualifizierter JMX-Objektname)
SchedulableMBeanMethod	Methode des MBeans, die ausgeführt werden soll
InitialStartDate	Zeitpunkt, für den die erste Ausführung vorgemerkt wird
SchedulePeriod	Intervall in Millisekunden zwischen zwei Aufrufen
InitialRepetitions	Anzahl der Aufrufe des auszuführenden MBeans; „-1“ bedeutet, bis zur Beendigung des Servers weitergemacht werden soll

Tabelle 5.3: Beschreibung der scheduler-service.xml

Da es sich bei dem Scheduler-Service auch um ein MBean handelt, lassen sich einzelne Parameter wie z. B. das Intervall zwischen zwei Aufrufen über JMX verändern, ohne den ganzen JBoss neu starten zu müssen:

```
./bin/twiddle.sh set jboss:service=Scheduler SchedulePeriod 60000
./bin/twiddle.sh invoke jboss:service=Scheduler restartSchedule
```

Ein Beispiel für die scheduler-service.xml Datei sieht folgendermaßen aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE server>
<server>
     <mbean code="org.jboss.varia.scheduler.Scheduler" name="jboss:service=Scheduler">
      <attribute name="StartAtStartup">true</attribute>
      <attribute name="SchedulableMBean">jboss:service=InternalTriggerService</attribute>
      <attribute name="SchedulableMBeanMethod">scanForLdapChanges(DATE, REPETITIONS)</attribute>
      <attribute name="InitialStartDate">NOW</attribute>
      <attribute name="SchedulePeriod">10000</attribute>
      <attribute name="InitialRepetitions">-1</attribute>
      <depends>jboss:service=InternalTriggerService</depends>
      <depends>
    <mbean code="javax.management.timer.Timer" name="jboss:service=Timer"/>
      </depends>
     </mbean>
</server>
```

Abbildung 5.8: scheduler-service.xml

5.3.3 server.xml

Die server.xml von Tomcat entspricht weitgehend dem Original. Angepaßt wurden die Verbindungseinstellungen.

In der folgenden Tabelle sind die Parameter für die Verbindungen aufgelistet:

Attribut	***Beschreibung***
maxThreads	Max. Anzahl an gleichzeitigen Requests; erhöht von 200 auf 300
minSpareThreads	Min. Anzahl an Threads, die immer für neue Verbindungen bereit gehalten werden; von 4 auf 25 erhöht
maxSpareThreads	Die max. erlaubte Anzahl an unbenutzten Threads; wenn es mehr werden, wird damit begonnen, Threads zu stoppen; auf 50 gelassen
acceptCount	Max. Queue-Länge für hereinkommende Requests (wenn maxThreads erreicht ist) von 10 auf 50 erhöht
connectionTimeout	Millisekunden, die der Connector nach dem Verbindungsaufbau darauf wartet, bis eine URL angefordert wird; von 60 Sek. auf 30 verringert

Tabelle 5.4: Beschreibung der server.xml

Der Abschnitt aus der server.xml sieht dementsprechend wie folgt aus:

```
<!-- A HTTP/1.1 Connector on port 8080 -->
<Connector port="${FT_OCSP_PORT}" address="${jboss.bind.address}"
        maxThreads="300" minSpareThreads="25" maxSpareThreads="50"
        enableLookups="false" redirectPort="8443" acceptCount="50"
        connectionTimeout="30000" disableUploadTimeout="true"/>
```

Abbildung 5.9: Abschnitt aus der server.xml

5.3.4 standardjboss.xml

Die Datei standardjboss.xml enthält die Default-Konfigurationen für die einzelnen Bean-Typen. Da der OCSP-Responder nur Stateless Session Beans verwendet, ist nur der entsprechende Abschnitt der Datei modifiziert worden.
Die Änderungen betreffen ausschließlich den Thread-Pool. Sie sind in der folgenden Tabelle aufgelistet:

Attribut	***Beschreibung***
MinimumSize	Min. Anzahl der EJB-Instanzen im Pool, wenn wenig oder keine Last auf dem Server ist; auf 40 gesetzt
MaximumSize	Max. Anzahl an EJB-Instanzen, die im Pool verfügbar gehalten werden; auf 300 gesetzt
strictMaximumSize	Falls gesetzt, wird damit verhindert, dass bei Bedarf mehr Instanzen als MaximumSize erzeugt werden. Weitere Requests werden dann so lange geblockt, bis eine Instanz freigegeben wird.

Tabelle 5.5: Beschreibung der standardjboss.xml

Der veränderte Abschnitt in der standarjboss.xml sieht somit folgendermaßen aus:

```
<container-configuration>
<container-name>Standard Stateless SessionBean</container-name>
...
<persistence-manager></persistence-manager>
    <container-pool-conf>
        <MinimumSize>40</MinimumSize>
        <MaximumSize>300</MaximumSize>
        <strictMaximumSize/>
    </container-pool-conf>
</container-configuration>
```

Abbildung 5.10: Abschnitt aus der standarjboss.xml

5.3.5 Installation der Testumgebung

Vor dier ersten Ausführung des OCSP-Servers müssen bestimmte Benutzer (secadmin, flexitru und ldap) und Gruppen (flexi, tomcat und ldap) in das System angelegt werden. Diese Aufgabe übernehmen die beiden Skripte *instUsers.sh und FlexEnv.sh*. Die FlexEnv.sh ist nach dem Verzeichnis */etc* zulegen. Danach soll `"./instUsers.sh YES"` aufgerufen werden.
Nachdem die Benutzer angelegt sind, muß der OCSP nur noch ins Verzeichnis */usr/local* entpacket werden.

Gestartet wird der OCSP folgendermaßen:

```
cd /usr/local/ft0prog.ocsp
. setFLEX_ENV.sh
. FlexEnv.sh
./bin/mldap0 start
./bin/controlOCSP.sh startOCSP
```

Mit `"./bin/controlFT.sh status"` kann überprüft werden, ob der OCSP läuft. Um den OCSP zu stoppen geht dann entsprechend mit `"stopOCSP"` bzw. `"stop"`.

Für das Einfügen von weiteren Zertifikaten in den LDAP liegt im Verzeichnis *„LdapFiller“* ein Skript bereit. Die neuen Zertifikate müssen dann in das entsprechende Verzeichnis des Mandanten abgelegt werden: `LdapFiller/OCSPTestCerts/Mandant000X/LdapFill/certs/`
Der Import der Zertifikate in den LDAP funktioniert dann folgendermaßen:

```
cd /usr/local/ft0prog.ocsp/LdapFiller
./LdapFiller.sh -c 2 -d OCSPTestCerts/Mandant000X -f LdapFiller.properties
```

Wobei anstelle von X, ist die Mandanten Nummer einzugeben. *-c* ist die ClientID und *-d* ist das Verzeichnis des Mandanten.
Die bereits importierten Zertifikate befinden sich noch in den oben genannten Verzeichnissen, damit hat man eine Übersicht, welche Zertifikate welchem Mandanten zugeordnet sind. Die CRL läßt sich am einfachsten mit dem ldapbrowser austauschen.

Kapitel 6

Ausblick

Der Online Certificate Revocation Protokol hat sich als Lösung für die online Statusabfrage eines Zertifikates durchgesetzt. Es besteht jedoch in umfangreichere PKIs einen Verbesserungsbedarf was die Performanz und Skalierbarkeit anbelangt.

Durch den Zertifikatscache wurde die Möglichkeit angeboten der Speicherverbrauch des OCSP-Responder selbst zu verwalten, dazu wurde der Startvorgang deutlich verkürzt im Vergleich zu dem Basis-OCSP, der alle im Master LDAP gespeicherten Zertifikate zu seinem Cache holt.

Die Pre-Produced Responses stellt ein guter Ansatz dar und eignet sich hervoragend in sehr großen PKI-Umgebungen. Allerdings ist dieser Ansatz nicht ohne weiteres realisierbar. Will man mit den Pre-Produced Responses effektiver arbeiten, so soll man auf die Nonce Extension verzichten. Das stellt allerdings ein Sicherheitsproblem dar, in dem das Risiko eines Replay-Angriffes zum Opfer zu fallen größer wird. So gesehen hat man eine Entscheidung zu treffen ob man, sich für die höhere Sicherheit oder für die Performanz des OCSP-Responder entscheidet.
Um weder die Performanz noch die Sicherheit zu gefährden soll man einen Trade-Off zwischen den beiden Aspekten ausmachen, mit dem man eine wesentliche bessere Performanz erzielt ohne, dass die Sicherheit beeinträchtigt wird.

Ein guter Ansatz ist der Kompromiss, dass man auf das Nonce möglichst verzichtet und stattdessen mit einem Zeitstempel arbeitet, welches das Ziel hat Replay-Angriffe zu verhindern. Dabei ist zu beachten, dass auch Angriffe auf das Zeitstempel-Verfahren realisierbar sind.

In dieser Arbeit wurde nicht auf das Nonce Extension verzichtet, sondern es wird betrachtet ob ein Nonce im Request vorhanden ist oder nicht und dementsprechend reagiert. Es ist dem Entwickler überlassen, ob er das Nonce nicht behandelt und ausschließlich mit oder ohne einem Zeitstempel im OCSP-Responder arbeiten will. Die Kombination zwischen dem Zertifikatscache und dem Responsecache hat in dieser Diplomarbeit gute Ergebnisse geliefert, auch wenn der Einsatz von Nonce-Extensions nicht ganz ausgeschlossen wird.

Literaturverzeichnis

[JohEi] Johannes Buchmann. *Einführung in die Kryptographie.* Springer Verlag, 3. Auflage, 2004.

[PKCS1] RSA Laboratries. *PKCS #1 v2.1: RSA Cryptography Standard.* Juni 2002. ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf.

[pki] Johannes Buchmann. *Skript zur Vorlesung Public-Key-Infrastrukturen,* TU-Darmstadt, Sommersemester 2005. http://www.cdc.informatik.tudarmstadt.de/lehre/SS05/vorlesung/PKI/resources.html.

[PKCS10] RSA Laboratries. *PKCS #10 v1.7: Certification Request Syntax Standard.* Mai 2000. ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-10/pkcs-10v1_7.pdf.

[PKCS12] RSA Laboratries. *PKCS 12 v1.0: Personal Information Exchange Syntax.* Juni 1999. ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-12/pkcs-12v1.pdf.

[hsm] Hardware Security Modules. http://www.eracom-tech.com/hsm.0.html.

[chipka] Wolfgang Rankl and Wolfgang Effing. *Handbuch der Chipkarten.* Hanser Verlag, 4.Auflage, 2004.

[SIGG01] Bundesnetzagentur. *Gesetz über Rahmenbedingungen für elektronische Signaturen (Signaturgesetz -SigG).* Mai 2001. http://www.bundesnetzagentur.de/media/archive/2247.pdf.

[rfc2251] Network Working Group. *Lightweight Directory Access Protocol (v3).* http://www.rfc-editor.org/rfc/rfc2251.txt.

[rfc3280] Network Working Group. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.* http://www.ietf.org/rfc/rfc3280.txt.

[rfc2527] Network Working Group. *Internet X.509 Public Key Infrastructure Certificate Policy and Certification Practices Framework.* http://www.ietf.org/rfc/rfc2527.txt.

[rfc3279] Network Working Group. *Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.* http://www.ietf.org/rfc/rfc3279.txt.

[rfc2560] Network Working Group. *X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP*. http://www.ietf.org/rfc/rfc2560.txt.

[EckItSec] Claudia Eckert. *IT-Sicherheit.* Oldenburg Verlag, 3.Auflage, 2004.

[rfc2459] Network Working Group. *Internet X.509 Public Key Infrastructure Certificate and CRL Profile*. http://www.ietf.org/rfc/rfc2459.txt.

[L-OCSP] PKIX Working Group. *Lightweight OCSP Profile for High Volume Environments*. http://www.ietf.org/internet-drafts/draft-ietf-pkix-lightweight-ocsp-profile-10.txt.

[rfc3379] Network Working Group. *Delegated Path Validation and Delegated Path Discovery Protocol Requirements*. http://www.ietf.org/rfc/rfc3379.txt.

[DraftSCVP] Internet Draft. *Server-based Certificate Validation Protocol (SCVP)*. http://www.ietf.org/internet-drafts/draft-ietf-pkix-scvp-31.txt.

[Attacks] William R. Cheswick, Steven M. Bellovin, Aviel D. Rubin. *Firewalls and Internet Security: Repelling the Wily Hacker, Second Edition*. Addison Wesley Professional, 2003.

[flexsec] FlexSecure GmbH. http://www.flexsecure.de/.