

## **Diplomarbeit**

Analyse der Sicherheit impliziter Autorisierung durch Referenzenvergabe  
bei CORBA Object Request Brokern

INSTITUT FÜR ANGEWANDTE MIKROELEKTRONIK UND DATENTECHNIK  
UNIVERSITÄT ROSTOCK

Diplomarbeit

**Analyse der Sicherheit impliziter  
Autorisierung durch  
Referenzenvergabe bei CORBA  
Object Request Brokern**

Christoph Becker

20. September 2006

Gutachter:

Prof. Dr.-Ing. Ralf Salomon  
Universität Rostock

Dipl.-Ing. Sebastian Staamann  
PrismTech GmbH

# Danksagung

Ich bedanke mich bei Prof. Ralf Salomon für die gute Betreuung der Arbeit.

Der Firma PrismTech, insbesondere Sebastian Staamann möchte ich für das in mich gesetzte Vertrauen und die hilfreiche Unterstützung danken.

Ein herzliches Dankeschön verdienen Nicolas Noffke für seinen jederzeit sachkundigen Beistand und Thomas Bertow für sein unendliches Repertoire an aufmunternden Worten während der gesamten Bearbeitung.

Allen namentlich nicht erwähnten Mitarbeitern von PrismTech Berlin, die mir bei kleineren oder auch größeren Problemen zur Seite gestanden haben, möchte ich ebenfalls für ihre Hilfsbereitschaft danken.

Ein besonderer Dank gebührt meiner Lebensgefährtin für ihre moralische Unterstützung, ihr Verständnis und die unermüdliche Motivationsbereitschaft.

Meinen Eltern danke ich, weil sie nie das Vertrauen in mich und meine Arbeit verloren haben und wie viele Freunde oft auf mich verzichten mußten.

# Inhaltsverzeichnis

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Einleitung</b>                       | <b>1</b>  |
| 1.1      | Thematischer Hintergrund . . . . .      | 2         |
| 1.2      | Motivation und Zielstellung . . . . .   | 3         |
| 1.3      | Gliederung der Arbeit . . . . .         | 3         |
| <b>2</b> | <b>Grundlagen</b>                       | <b>5</b>  |
| 2.1      | Was ist CORBA . . . . .                 | 5         |
| 2.2      | Anwendungsbeispiel . . . . .            | 10        |
| 2.3      | Sicherheitsanforderungen . . . . .      | 11        |
| 2.4      | Angriffsszenarien . . . . .             | 15        |
| 2.5      | Bestehende Lösungen . . . . .           | 17        |
| <b>3</b> | <b>Material, Methoden und Werkzeuge</b> | <b>19</b> |
| 3.1      | Betrachtete Produkte . . . . .          | 19        |
| 3.2      | Methoden . . . . .                      | 22        |
| 3.3      | Werkzeuge . . . . .                     | 25        |
| <b>4</b> | <b>Ergebnisse</b>                       | <b>27</b> |
| 4.1      | MICO . . . . .                          | 28        |
| 4.2      | ORBacus . . . . .                       | 35        |
| 4.3      | Orbix . . . . .                         | 39        |
| 4.4      | Sun ORB . . . . .                       | 41        |
| 4.5      | Visibroker . . . . .                    | 45        |
| 4.6      | Weblogic . . . . .                      | 48        |

|          |   |           |
|----------|---|-----------|
| 4.7      | WebSphere . . . . .                       | 51        |
| 4.8      | Komplexität der Objektschlüssel . . . . . | 53        |
| <b>5</b> | <b>Diskussion</b>                         | <b>56</b> |
| 5.1      | Komplexität . . . . .                     | 56        |
| 5.2      | Machbarkeitsstudie . . . . .              | 58        |
| 5.3      | Lösungsmöglichkeiten . . . . .            | 60        |
| <b>6</b> | <b>Zusammenfassung</b>                    | <b>63</b> |
| <b>A</b> | <b>Anhang</b>                             | <b>65</b> |
| A.1      | Quelltextauszüge . . . . .                | 65        |
| A.2      | Aufbau der Objektschlüssel . . . . .      | 69        |

# Tabellenverzeichnis

|      |  |    |
|------|--|----|
| 3.1  | Übersicht der benutzten Request Broker . . . . .           | 20 |
| 4.1  | Transiente MICO Objektschlüssel . . . . .                  | 29 |
| 4.2  | Transiente MICO Objektschlüssel mehrerer POA . . . . .     | 29 |
| 4.3  | Persistente MICO Objektschlüssel . . . . .                 | 31 |
| 4.4  | Persistente MICO OKs in POA-Hierarchie . . . . .           | 31 |
| 4.5  | Transienter ORBacus Objektschlüssel . . . . .              | 36 |
| 4.6  | Teile transienter ORBacus Objektschlüssel . . . . .        | 36 |
| 4.7  | Transiente Schlüsselteile verschiedener POAs . . . . .     | 36 |
| 4.8  | Persistenter ORBacus Objektschlüssel . . . . .             | 37 |
| 4.9  | Zwei Transiente Orbix Objektschlüssel . . . . .            | 39 |
| 4.10 | Orbix Objektschlüsselabschnitte . . . . .                  | 40 |
| 4.11 | Orbix - verschiedene POAs . . . . .                        | 40 |
| 4.12 | Transienter SUN ORB Objektschlüssel . . . . .              | 42 |
| 4.13 | Transiente SUN Objektschlüssel verschiedener POA . . . . . | 43 |
| 4.14 | SUN POA-Namen, Kodierung . . . . .                         | 43 |
| 4.15 | Transienter Visibroker Objektschlüssel . . . . .           | 46 |
| 4.16 | Persistenter Visibroker Objektschlüssel . . . . .          | 46 |
| 4.17 | Visibroker Zeitstempel . . . . .                           | 47 |
| 4.18 | Transienter Weblogic Objektschlüssel . . . . .             | 49 |
| 4.19 | Weblogic Objektzähler . . . . .                            | 50 |
| 4.20 | Transienter WebSphere Objektschlüssel . . . . .            | 51 |
| 4.21 | Vergleich der Objektschlüsselkomplexität . . . . .         | 53 |

## Tabellenverzeichnis

---

|  |    |
|--|----|
| 4.22 Umrechnung Komplexität in Zeitaufwand . . . . . | 55 |
|--|----|

# Abbildungsverzeichnis

|      |   |    |
|------|---|----|
| 2.1  | Scheinbarer und tatsächlicher Weg eines Objektaufrufs . . . | 6  |
| 2.2  | Aufbau einer Interoperablen Objektreferenz . . . . .        | 8  |
| 2.3  | Komponenten der corbaloc-URL . . . . .                      | 16 |
| 3.1  | Anordnung der POA . . . . .                                 | 24 |
| 4.1  | Aufbau transienter MICO-Objektschlüssel . . . . .           | 30 |
| 4.2  | Aufbau persistenter MICO-Objektschlüssel . . . . .          | 31 |
| 4.3  | Aufbau transienter ORBacus-Objektschlüssel . . . . .        | 37 |
| 4.4  | Aufbau persistenter ORBacus-Objektschlüssel . . . . .       | 38 |
| 4.5  | Aufbau Orbix-Objektschlüssel . . . . .                      | 41 |
| 4.6  | Aufbau von SUN-ORB Objektschlüsseln . . . . .               | 44 |
| 4.7  | Aufbau transienter Visibroker-Objektschlüssel . . . . .     | 47 |
| 4.8  | Aufbau persistenter Visibroker-Objektschlüssel . . . . .    | 48 |
| 4.9  | Aufbau transienter Weblogic-Objektschlüssel . . . . .       | 50 |
| 4.10 | Aufbau transienter WebSphere-Objektschlüssel . . . . .      | 52 |

# 1

## Einleitung

Diese Arbeit befasst sich mit der Sicherheit von verteilten Systemen die auf CORBA basieren. Die Zugriffssteuerung innerhalb dieser Systeme findet mit Hilfe von Objektreferenzen statt. Diese ermöglichen den Zugriff auf andere Objekte innerhalb des Gesamtsystems. In dieser Arbeit werden die Implikationen betrachtet, die durch die Vergabe dieser Referenzen entstehen und die Erzeugung der Objektreferenzen untersucht.

Um Problemen mit Begriffsdefinitionen aus dem Weg zu gehen, werden die verbreiteten englischen Begriffe und Namen verwandt. Dies betrifft in der Hauptsache die Wörter, die aus der CORBA Spezifikation stammen und damit auch die darauf basierende Literatur prägen. Fachbegriffe werden aus dem Deutschen benutzt, wenn dies möglich und sinnvoll ist.

Alle Quelltexte und viele der erzeugten Analysedaten, sowie die Testprogramme die nachfolgend benannt werden, befinden sich vollständig auf der beigelegten CD-ROM.

## 1.1 Thematischer Hintergrund

Viele Informationssysteme werden heute als verteilte Systeme realisiert. Ein verteiltes System ist nach [TvS03] ein Zusammenschluss unabhängiger Computer, welches sich für den Benutzer als ein einzelnes System präsentiert.

Einerseits können mehrere Rechner für die Erledigung einer Aufgabe zu deren Beschleunigung benutzt werden, beispielsweise zur Berechnung komplizierter Sachverhalte wie Wetterprognosen. Andererseits kann der mehrfache Einsatz gleichartiger Systeme die Ausfallsicherheit erhöhen. Durch die Vervielfachung der Systemkomponenten erhöht sich auch die Komplexität. Um die Systemkomplexität beherrschen zu können, wurden neuartige Softwarekonzepte entwickelt.

Eines dieser Konzepte ist die *Common Object Request Broker Architecture* (CORBA). Sie ist eine weit verbreitete und über die Jahre gereifte Architektur für verteilte Systeme<sup>1</sup>. Die konkrete Realisierung dieser Architektur sind *Object Request Broker* (ORB). Die ORB vermitteln als sogenannte *Middleware* zwischen mehreren Systemen und vermindern die aus dem Konzept heraus entstehende Komplexität auf ein erträgliches Maß. Der Anwendungsprogrammierer muss sich nicht mehr um einzelne Protokolle und deren Spezifikation kümmern um eine verteilte Anwendung entwickeln zu können. CORBA selbst „spezifiziert im Wesentlichen eine Ordnung zur Realisierung verteilter objektorientierter Anwendungen“ [Say97, S. 15] in heterogenen Netzen.

Durch den Einsatz verteilter Systeme und deren weite Verbreitung muss besonders großer Wert auf die Sicherheit gelegt werden, um Risiken zu minimieren. Setzt beispielsweise eine Bank CORBA ein, um die Auszahlungen ihrer Geldautomaten erfassen zu können, so ist es nicht nur für die Bank wichtig, dass diese Transaktionen nachvollziehbar sind und nicht manipuliert werden können, sondern auch für den Kunden.

---

<sup>1</sup>engl.: *distributed computing*

## 1.2 Motivation und Zielstellung

Bis heute ist die Annahme verbreitet, dass ein Zugriff auf CORBA-Objekte nur durch zuvor autorisierte Personen oder Systeme mittels übergebener Objektreferenzen stattfinden kann.<sup>2</sup> Praktisch würde dies bedeuten, dass nur die Polizei, die Stadtverwaltung und alle von mir benachrichtigten Personen bei einem Umzug meine neue Adresse erhalten. Niemand sonst? Jeder der meine Adresse bekommt, wurde durch mich autorisiert?

Im Rahmen dieser Arbeit wird untersucht, inwiefern die Beschaffung typischerweise nicht geschützter Informationen, wie Interoperable Objektreferenzen, es ermöglicht, illegal Zugriff auf entfernte, verteilte Systemen zu erlangen. Daraus resultierend wird der hierbei durchschnittlich benötigte Aufwand (Zahl der nötigen Versuche oder geschätzte Zeit) ermittelt. Als repräsentative Untersuchungsobjekte wird die Arbeit gängige ORB-Produkte wie MICO, Orbix, SUN-ORB, VisiBroker, WebLogic-ORB und WebSphere-ORB verwenden und analysieren.

Um die praktische Relevanz der theoretisch ermittelten Risiken beurteilen zu können, werden Werkzeuge entwickelt, die versuchen, illegalen Zugriff auf entfernte Objekte zu erhalten. Im Anschluss an die Vorstellung dieser Werkzeuge werden Vorschläge zur Verbesserung der Sicherheit gemacht, die verschiedenen Lösungsansätzen, mit und ohne Änderungen an bestehenden Systemen, folgen.

## 1.3 Gliederung der Arbeit

Die Arbeit beginnt mit einer kurzen Einführung in CORBA und erklärt, was Objektreferenzen sind und woher diese stammen. In Kapitel drei werden die untersuchten Programme und die für diese Arbeit notwendigen Methoden betrachtet. Im Anschluss daran befinden sich die Ergebnisse der einzelnen Analysen die wiederum im fünften Kapitel, der Diskussion

---

<sup>2</sup>Natürlich existiert auch die Möglichkeit über einen Namensdienst oder ein Implementation Repository eine Referenz zu erhalten, nur ist dies nicht der Gegenstand dieser Arbeit.

zusammengefasst und diskutiert werden. Eine Machbarkeitsstudie und deren Ergebnis sind ebenso Teil dieses Kapitels, wie Lösungsvorschläge zur Beseitigung der ermittelten Sicherheitslücken. Die Arbeit schließt mit den letzten Worten der Zusammenfassung.

*Zweifel zu haben ist ein unangenehmer,  
sich in Sicherheit zu wiegen ein absurder Zustand.*

Voltaire

# 2

## Grundlagen

### 2.1 Was ist CORBA

Die Common Object Request Broker Architecture ist eine durch die *Object Management Group*<sup>1</sup> (OMG), einem internationalen nicht gewinnorientierten Gremium, standardisierte Architektur zum Aufbau verteilter Systeme. Die Hauptziele der OMG sind die Entwicklung und Standardisierung von Softwarekonzepten.

CORBA ist die Middleware-Plattform<sup>2</sup> der OMG. Hauptbestandteil des CORBA Standards ist der *Object Request Broker*, kurz ORB. Er tritt als Vermittler auf und versetzt Systemkomponenten (Programme oder deren Teile) in die Lage, sogenannte Objektfernaufrufe<sup>3</sup> durchführen zu können. Die einzelnen Systemkomponenten kennen weder den eigenen Standort im

---

<sup>1</sup><http://www.omg.org>

<sup>2</sup>Middleware wird im Deutschen gelegentlich als *Integrationsplattform* bezeichnet.

<sup>3</sup>auch bekannt als RMI (engl.: *remote method invocation*)

Gesamtsystem, noch den des Kommunikationspartners.<sup>4</sup> Ein weiterer Bestandteil ist die *Interface Definition Language*<sup>5</sup> (OMG IDL) als abstrakte Sprache zur Schnittstellenbeschreibung. Mit IDL ist eine einfache Abstraktionsprache gefunden worden, die jederzeit ein Quelltextgrundgerüst für eine Anwendung in vielen Programmiersprachen erzeugen kann.<sup>6</sup>

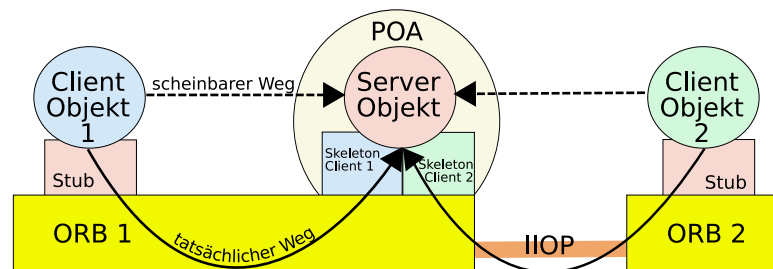


Abbildung 2.1: Scheinbarer und tatsächlicher Weg eines Objektaufrufs

Abbildung 2.1 zeigt den prinzipiellen Aufbau eines CORBA-Systems und den Weg, den ein Objektaufruf nimmt. Ein Clientobjekt will auf die von einem anderen Objekt (Server) bereitgestellte Funktionalität zugreifen. Aus Sicht der Objekte, interagieren sie so, wie sie auch ohne CORBA miteinander interagieren würden. Der Client stellt seine Anfrage in CORBA aber an einen Stellvertreter des Serverobjektes, den sogenannten Stub. Dieser setzt die Anfrage in ein generisches ORB-Format um und übergibt sie an den ORB. Im Folgenden sei der Einfachheit halber angenommen, dass der ORB die Adresse des Zielobjektes kennt. Über diese Adresse findet er den passenden *Portable Object Adapter* (POA), eine Art Registratur, bei der das Serverobjekt vom Programmierer angemeldet wurde. Der POA leitet die Anfrage an ein Stellvertreterobjekt (Skeleton) weiter, dass die Anfrage dann an das Serverobjekt stellt. Das Skeleton simuliert für den Server das Clientobjekt bzw. dessen Schnittstellen. Dem Server erscheint ein Aufruf so, als würde der Client direkt mit ihm kommunizieren.

<sup>4</sup>In der Literatur auch als Ortstransparenz [HJS01, S. 14] oder „location transparency“ [Sie96, S. 306][BVD01, S. 12] bezeichnet.

<sup>5</sup>ISO 14750

<sup>6</sup>Beispielsweise C++, JAVA oder Python.

Hätte der ORB keine Adresse für das Zielobjekt erhalten, so gibt es in CORBA die Möglichkeit, anhand des vom Client verlangten Objekttyps (*TypeId* bzw. *RepositoryId*) ein passendes Objekt heraus zu suchen. Dieser Mechanismus wird über ein *Implementation Repository* bereitgestellt und ist optional.

Wo kommen Stubs und Skeletons her? Die in der Interface Definition Language verfassten Objektdefinitionen werden mit Hilfe eines IDL-Compilers in Quelltexte für die jeweils gewünschte Programmiersprache umgewandelt. Diese, auch Gerüste genannten Quelltexte, dienen beim Objektaufzuruf dazu, dem Client die Schnittstelle des Servers zur Verfügung stellen zu können und umgekehrt.

Damit Objektaufrufe in verteilten (und gegebenenfalls vernetzten) Systemen funktionieren, müssen die Vermittler miteinander kommunizieren können. Deshalb kommt für die Kommunikation dieser verteilten Systeme das *General Inter-ORB Protocol* (GIOP) zum Einsatz. GIOP definiert ein einfaches Format, um Anfragen und Antworten von Client und Servant zu transportieren, ohne sich auf Transportprotokolle festzulegen [BVD01, S. 81f]. Das *Internet Inter-ORB Protocol* (IIOP) ist eine GIOP-Ergänzung, die GIOP auf TCP/IP abbildet [Kli00, S. 128][OMG04, 15.2].

Damit die Anfrage in Abbildung 2.1 von einem Objekt zum anderen vermittelt werden kann, muss diese adressiert werden. Doch wo wird die Adresse festgelegt? Zunächst wird ein Serverobjekt bei einem Objektadapter (POA) registriert. Dadurch erhält es innerhalb des POAs eine eindeutige Adresse. Der POA selbst ist innerhalb des ORB ebenfalls eindeutig identifizierbar. Dies geschieht in der Regel durch einen Namen. Der ORB erzeugt für jedes Objekt eine Adresse, es ist die Anschrift des Objektes innerhalb des ORBs<sup>7</sup>. Die Objektadresse innerhalb eines ORB ist der *Objektschlüssel*<sup>8</sup>. Mit ihm ist ein Zugriff auf das Objekt möglich. Damit (wie in Abbildung 2.1) Client 1 und Client 2 auf das Serverobjekt zu-

---

<sup>7</sup>Alle Objekte und Objektadapter die zu einem ORB gehören, also von ihm erzeugt wurden, sind Teil einer Domäne. Man bezeichnet diese häufig auch als ORB-Domäne.

<sup>8</sup>engl.: *object key*

greifen können, ist eine ORB-übergreifende Adressierung notwendig. Für diese Adressierung und Zustellung von Anfragen über ORB-Grenzen hinweg und zur Zusammenarbeit mit den ORB anderer Hersteller, wird IIOP benutzt. In IIOP existieren die Objektschlüssel als Bestandteil einer *Interoperable Object Reference* (IOR), der Interoperablen Objektreferenz. Zur Veranschaulichung ist die IOR in Abbildung 2.2 graphisch dargestellt.<sup>9</sup> Der Fokus dieser Arbeit liegt auf dem Objektschlüssel (rot) innerhalb der IOR.

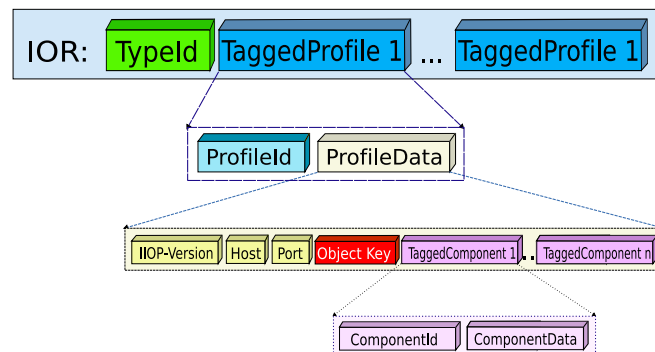


Abbildung 2.2: Aufbau einer Interoperablen Objektreferenz

Für die Generierung jedes Objektschlüssels ist der jeweilige *Portable Object Adapter* (POA) verantwortlich. An einem Objektadapter (POA) können nicht nur Objekte, sondern auch andere POA registriert werden, wodurch der Aufbau einer Hierarchie möglich ist. Die Wurzel dieser Hierarchie bildet immer der sogenannte RootPOA. Bei ihm können Kind-POAs registriert werden. Objektadapter werden mit sogenannten *Policies* initialisiert. Diese legen fest, wie der POA Objekte benennen, also den Objektschlüssel erzeugen soll<sup>10</sup> und wie der Lebenszyklus<sup>11</sup> der Objekte definiert ist. Bei der Namensgebung unterscheidet man zwischen vom System vergebenen Objektschlüssel und vom Benutzer vergebenen Objektschlüsseln.

<sup>9</sup>Die Definition der IOR ist im Anhang in den Quelltexten A.1 und A.2 zu finden.

<sup>10</sup>*IdAssignmentPolicy*

<sup>11</sup>*LifespanPolicy*

Objekte haben ferner zwei mögliche vorbestimmte Lebensläufe. Der Erste nennt sich **transient** und bedeutet, dass das Objekt aufhört zu existieren, wenn der POA bei dem es registriert wurde aufhört zu existieren. **Persistent** nennt man Objekte, die länger existieren können, als der POA. [OMG04, 11.3.8.2] Des Weiteren befinden sich Objekte im „aktiven“ oder „inaktivem“ Zustand. Aktiv heisst in diesem Kontext, dass das Objekt benutzt werden kann und nicht, dass es aktiv etwas tut. Meist wird die sogenannte *Implizite Aktivierung* benutzt, dabei werden Objekte, wenn sie bei einem POA registriert werden, automatisch durch diesen aktiviert.

Wie gelangt man an die Objektreferenzen um diese weitergeben zu können? Durch die ORB-Funktionen `object_to_string()` und `string_to_object()` kann eine Objektreferenz in eine Zeichenkettenform und wieder zurück in eine Referenz konvertiert werden [OMG04, 13.6.9]. Um einen Eindruck davon zu bekommen, wie eine IOR aussieht, ist nachfolgend eine IOR eines ORBacus Objektes abgebildet:

```
IOR:0000000000000001749444c3a48656c6c6f4170702f48656c6c6f3a312
e30000000000001000000000000007800010200000000166c6f63616c686f
73742e6c6f63616c646f6d61696e00975a0000002aabacab3131313436343
233313539005f526f6f74504f41006e756c6c0000cafebabe445507770000
000000000000001000000010000001c000000000010001000000200010
020050100010001010900000000
```

Um auf ein Objekt zu verweisen kann alternativ zur IOR, auch das für Menschen besser verständliche `corbaloc` URL-Format [OMG04, 13.6.10.1] benutzt werden. Ein Beispiel<sup>12</sup> für eine `corbaloc`-Adresse ist:

```
corbaloc:iiop:1.1@555xyz.com/mypoa/object0001
```

Während `corbaloc:` dem Format und `iiop:1.1` dem Protokoll und dessen Version entspricht, steht `555xyz.com` für die Adresse des Rechners, auf dem das Objekt zu finden ist. Daneben bezeichnet `mypoa/object0001`

---

<sup>12</sup>Das Beispiel ist fiktiv.

das Objekt, wobei es sich um den Objektschlüssel handelt. Er kann Informationen über seinen POA enthalten. Der POA und der ORB sorgen dafür, dass ein Objektaufruf mit Hilfe des Objektschlüssels in der IOR oder der corbaloc Adresse auch an die richtige Inkarnation eines Objektes weitergeleitet wird.

Damit ein Objektaufruf zwischen ORB verschiedener Hersteller auf verschiedenen Rechnerarchitekturen funktioniert, wurde die *Common Data Representation* (CDR) eingeführt. Der Client-ORB wandelt alle Daten in dieses Format um und überträgt sie an den ORB der Gegenseite. Dieser kümmert sich selbstständig um die Rückwandlung dieser Daten. Somit entstehen keine Probleme mit der Anordnung der Bytes (Little vs Big Endian) und der Datenstrom ist klar definiert.

### 2.2 Anwendungsbeispiel

Die Übermittlung der aktuellen Messdaten einer Wetterstation erfolgt beispielsweise zwischen Client und Server, die auf jeweils verschiedenen Rechnern laufen. Zwischen beiden besteht eine ungesicherte Netzwerkverbindung. Das heisst, es wird weder SSL noch eine andere Verschlüsselung benutzt. Diese Teststellung wäre unter anderem für eingebettete Systeme realistisch. Die Datenübermittlung soll von dem in der Wetterstation befindlichen Client auf den Server, der mit einer Datenbank gekoppelt ist, erfolgen.

Zunächst benötigt der Client eine Möglichkeit, den Server physisch zu erreichen. Diese ist durch die Netzwerkverbindung gesichert. Der logische Zugriff, also der Datentransport, soll mit Hilfe von CORBA realisiert werden.

Der Client benötigt also die Adresse des Servers in Form der Objektreferenz. Der Einfachheit halber kann man annehmen, dass dem Client die Adresse in Form einer Datei, die die IOR enthält, mitgeteilt wird. Er hat nun alle notwendigen Daten, um mit dem Server interagieren zu können. Der Zugriff ist über die IOR gewährleistet. Desweiteren geht man davon

aus, dass entweder der Client berechtigt ist, diese Dienste in Anspruch zu nehmen oder der Server geeignete Mechanismen besitzt, eine Berechtigung zu prüfen. Derartige Kontrollmechanismen benötigen häufig eine umfangreiche Implementierung. Dieser Umfang ist jedoch nicht immer erwünscht oder zweckmäßig.

Im Bereich eingebetteter Systeme geht man oftmals davon aus, dass dasjenige Objekt autorisiert ist, welches eine IOR besitzt oder dass die Vergabe einer IOR eine Autorisierung darstellt. Diese Annahme ist als implizite Autorisierung bekannt und basiert auf der Annahme, dass es mit vertretbarem Aufwand nicht möglich ist, an eine gültige Objektreferenz zu gelangen.

Dies wirft natürlich die Frage auf, ob und wie Dritte (z. B. Angreifer) ohne legitime Übergabe einer IOR in deren Besitz gelangen können und somit Zugriff auf das entsprechende Objekt erhalten.

### 2.3 Sicherheitsanforderungen

An informationsverarbeitende Systeme werden heute hohe Sicherheitsanforderungen gestellt. Diese Anforderungen sind gleichzeitig auch Ziele, die bei der Integration der Software zu erreichen sind. Im Besonderen interessieren Anforderungen an die Zugriffssicherheit wie Autorisierung und Authentifikation.

Findet eine Autorisierung statt? Wenn ja wie? Wie sicher ist diese und kann sie umgangen werden? Analoge Fragen stellen sich auch zur Authentifikation.

#### 2.3.1 Authentifikation

Authentifikation ist die Identitätsüberprüfung (oder -kontrolle) einer Gegenstelle. Identifizierung und Authentifikation werden oft synonym benutzt. Jedoch besteht ein Unterschied insofern, dass bei der Authentifikation eine Identität überprüft wird und bei der Identifizierung nur das Vorhandensein einer Identifizierungseigenschaft nötig ist. Das Vorlegen

eines Personalausweises ist beispielweise die Identitätsbehauptung (Identifikation), die Überprüfung durch einen Polizisten hingegen eine Authentifizierung (Identitätsüberprüfung). Zu beachten ist hierbei, dass der Polizist der Ausgabestelle des Personalausweises und dem Dokument an sich, dass heisst dessen Integrität trauen muss. Dies ist das Prinzip des vertrauenswürdigen Dritten, der die Identität bestätigt hat. Dies wird beispielsweise bei SSL-Zertifikaten so gehandhabt. Eine Certification Authority (CA) kontrolliert die Identität des Antragstellers und bestätigt mit der eigenen Signatur, dass die im Zertifikat vorgegebene Identität, die des Antragstellers ist. Derjenige, dem das signierte Zertifikat vorgelegt wird, muss der CA trauen und Mechanismen besitzen, die Korrektheit der CA-Signatur zu überprüfen. Dann kann er auch der vorgelegten Identitätsbehauptung trauen.

Der Nachweis der eigenen Identität kann über viele Wege erfolgen. Zum einen über Besitz (man hat etwas wie einen Schlüssel), via Wissen (man weiß etwas, Beispiel: PIN, Passwort) oder über körperliche (biometrische) Merkmale (man ist/kann etwas). Die verbeitete 2-Faktor-Authentifizierung findet statt, wenn zwei dieser Möglichkeiten kombiniert werden (Beispiel: Geldautomat - man hat etwas, die Karte und man weiss etwas, die PIN).

Authentifizierung ist die Grundlage für andere Anforderungen wie Vertraulichkeit, Verbindlichkeit oder Autorisierung.

### 2.3.2 Autorisierung

In der Informationstechnologie bezeichnet Autorisierung die Zuweisung und Überprüfung von Zugriffsrechten auf Daten oder Diensten an Nutzern.<sup>13</sup> Sie kann explizit oder implizit erfolgen.

Explizite Autorisierung bedeutet, dass ein Mechanismus einem Nutzer (oder auch einem Prozess) den Zugriff auf ein definiertes Objekt erlaubt hat. Dies kann gegebenenfalls überprüft werden. Diese Form der Autorisierung setzt Authentifikation voraus, es sei denn, es gibt die Zulassung für

---

<sup>13</sup><http://de.wikipedia.org/wiki/Autorisierung>, Stand 09:54, 20. Jul 2006

anonyme Benutzer. Dieser Fall kann Autorisierung an sich ad absurdum führen.<sup>14</sup>

Zur Vereinfachung von Autorisierungsmechanismen, wird meist *Role Based Access Control*<sup>15</sup> (RBAC) eingesetzt. Dem Rollensystem liegt die Idee zugrunde, dass Berechtigungen zur Nutzung geschützter Komponenten direkt an Rollen bzw. Aufgaben geknüpft ist. Die Aufgaben werden durch die Rollen modelliert. Es muss nun sichergestellt werden, dass die Subjekte (Nutzer) nur in die für sie vorgesehenen Rollen schlüpfen können. Der Nutzer selbst steht im RBAC nicht im Mittelpunkt, sondern seine Aufgabe und die dafür nötigen Rechte.

Mit Rollen können auch sogenannte Rollenhierarchien erzeugt werden. Explizite Autorisierung auf einer Hierarchiestufe kann eine implizite Autorisierung auf anderen Hierarchiestufen bewirken.

Beispiel einer Rolle: Ein Hausmeister erhält einen Generalschlüssel, durch den er Zutritt zu allen Räumen mit Ausnahme der Büroräume eines Bürohause hat. Ein Firmenchef, der eine Etage gemietet hat, bekommt Zugang zum Gebäude und allen gemieteten Räumen. Die Rechte des Hausmeisters resultieren aus seinen Aufgaben und begründen sich nicht auf die Person an sich.

Als Beispiel einer Rollenhierarchie bekommen in einer Bank bekommen alle Angestellten einen Hausschlüssel, aber nur Kassenprüfer einen Tresorschlüssel. Ein Kassenprüfer wird aber auch als Angestellter geführt - übt also auch diese Rolle aus. Er hat in der Rolle des Angestellten auch das Recht, einen Hausschlüssel zu besitzen.

### 2.3.3 Verbindung zu CORBA

Bei der Realisierung von CORBA-basierten Applikationen wird von Anwendungsprogrammierern oftmals die Annahme gemacht, dass die Verga-

---

<sup>14</sup>Wenn anonym Zugriff erlangt werden kann, so ist eine Authentifikation der eigenen Person prinzipiell sinnfrei. Dass Anwendungen denkbar sind, in denen es trotzdem Sinn hat die eigene Identität zu beweisen, sei unbestritten.

<sup>15</sup>Oftmals Rollenbasierte Systeme genannt.[Eck06, S. 245]

be einer Objektreferenz an ein Nutzerobjekt eine implizite Autorisierung, ähnlich einer Capability darstellt.

Eine Capability ist nach [Eck06, S. 555] als geschützter Zugriffsausweis definiert, der Zugriff auf das benannte Objekte gestattet und gleichzeitig auch die Berechtigungen regeln kann. Um die Capabilities fälschungssicher zu machen, werden zunehmend Verschlüsselungs- und Signaturmechanismen eingesetzt. Da Capabilities nicht an Subjekte gebunden sind, können sie in einfacher Weise weitergegeben werden. Eine kontrollierte Weitergabe ist nicht möglich.

Beispielsweise werden für ein Musikkonzert verschiedene Eintrittskarten verkauft (VIP, Sitz- und Stehplätze). Diese Information ist auf der Karte vermerkt. Damit nun das Personal am Getränkestand nicht jedesmal die Karten kontrollieren muss, werden am Einlass die Karten gegen Farbbänder getauscht. Ein grünes Band ist nun der Ausweis für den VIP Status. Eine erneute Berechtigungsprüfung erfolgt nicht. Der Veranstalter versucht aber fälschungssichere Farbbänder zu produzieren, die sich nicht vom Besucher (Subjekt) trennen lassen. Er kämpft also mit den Schwächen einer auf Capabilities basierenden Autorisierung.

Ein weiteres Einsatzgebiet für Zugriffsausweise ist Kerberos<sup>16</sup>. Ein nach der Authentifizierung erhaltenes Ticket dient als Capability und somit als Ausweis und Legitimation gegenüber anderen Diensten und Prozessen. So kann man auf weitere Authentifizierungsmaßnahmen verzichten. Kerberos kann beispielsweise auch beim sogenannten Single-Sign-on benutzt werden.

Weitere Informationen zu Capabilities und deren Notwendigkeit findet man in „The Confused Deputy, or why capabilities might have been invented“<sup>17</sup>.

Eine Objektreferenz kann ebenfalls als Capability angesehen werden. Nimm man nun an, dass eine Objektreferenz illegal erlangt werden kann, so erhält man das Recht zur Interaktion mit diesem Objekt, wenn keine

---

<sup>16</sup>Kerberos ist ein Protokoll, das den Zugang zu geschützten Systemen kontrolliert. Siehe RFC4120.

<sup>17</sup><http://www.cis.upenn.edu/~KeyKOS/ConfusedDeputy.html>

zusätzlichen Zugriffskontrollmechanismen vorhanden sind.

Diese Annahme wirft die Frage auf, ob und wie solche Objektreferenzen erzeugt werden können. In CORBA identifiziert der Objektschlüssel jedes Objektes innerhalb einer ORB-Domäne, die IOR systemweit. Während die Bestandteile der IOR wie Netzwerkadresse und Portnummer in der Regel aus der Kenntnis des Anwendungsszenarios gewonnen werden können, erfordert die Bestimmung des Objektschlüssels detaillierte Analysen. Eine solche ergibt, dass Objektschlüssel häufig durch einfache Mechanismen generiert werden, die die Vorhersage weiterer Objektschlüssel zulassen [SB06][Bec06]. Dadurch lassen sich weitere Schlüssel erraten und es werden nicht autorisierte Zugriffe auf Objekte des gleichen ORBs ermöglicht. Dies stellt eine potentielle Sicherheitslücke dar.

Für die Zugriffskontrolle ist es zwingend notwendig, dass IORs oder Objektschlüssel nicht erraten oder gefälscht werden können. Diese Forderung ergibt sich aus ihrer Verwendung als Capabilities.

### 2.4 Angriffsszenarien

Im Folgenden wird betrachtet, welche Möglichkeiten ein nicht autorisierter Angreifer besitzt.

Die Kenntnis von Host und Port, sowie der Signatur<sup>18</sup> des anzugreifenden Objektes kann man als gegeben hinnehmen. Host und Port können gegebenenfalls durch Netzwerkskans ermittelt werden - die Signatur ist für eine aktive Nutzung des Objektes jedoch essentiell. Dieses Wissen ist per se nicht geheim, man denke nur an eine Wissensabwanderung durch natürliche Mitarbeiterfluktuation. Die einzig fehlende Information ist der Objektschlüssel (Siehe Abbildung 2.3, corbaloc-URL).

Dieser ist durch die OMG nicht standardisiert worden und somit herstellerabhängig [OMG04, 2.1.4]. Ob und wie man diese selbst produzieren

---

<sup>18</sup>Die Signatur ist hier im Kontext der funktionalen Programmiersprachen zu sehen. Sie beinhaltet die Anzahl und Typen der Methodenparameter und den Typ des Rückgabewertes, ggf. auch zu erwartende Ausnahmen (Exceptions).

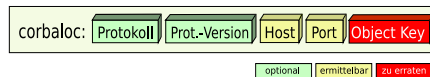


Abbildung 2.3: Komponenten der corbaloc-URL

kann, ist der Schwerpunkt dieser Arbeit und wird im Hauptteil eingehend erörtert.

Ist man bereits im Besitz einer IOR und ist das referenzierte Objekt immer noch aktiv, also ansprechbar, so besitzt man alle Daten für den Objektzugriff. Für den Zugriff auf andere Objekte muss häufig nur ein Zugriffsparameter geändert werden. Anhand einer IOR oder auch nur eines Objektschlüssels ist es möglich, sowohl den ORB-Typ und den POA-Namen oder die -Hierarchie, als auch Erzeugungszeiten beider zu ermitteln.

Evaluiert man die Mechanismen, die solche Objektschlüssel erzeugen und kann man diese nachstellen, so müsste es möglich sein, selbst gültige Objektschlüssel zu erzeugen. Mit Hilfe dieser kann illegal der Zugriff erlangt werden. Die Algorithmen sind natürlich im Quelltext zu finden. Open Source ORBs, so könnte man annehmen, sind eine leichtere Beute als Closed Source Produkte. Doch eine „Geheimhaltung“ des Algorithmus<sup>19</sup> hilft in diesem Fall nur mäßig, da das Ergebnis, die Objektschlüssel, ja in beliebiger Menge erzeugt werden können und somit auf den Algorithmus dahinter schließen lassen. Nach Jay Beale<sup>20</sup> kann Closed Source hier nur helfen, potentielle Sicherheitslücken oder sogenannte Geheimnisse etwas länger zu bewahren. Im weiteren Verlauf der Arbeit wird gezeigt, dass es für den Angreifer unerheblich ist, woher der ORB stammt. Ein Weg hinein findet sich immer.

Zuvor wurde davon ausgegangen, dass Host und Port bekannt sind. Allein mit diesen Daten ist es möglich einen *Denial-of-Service* (DoS) Angriff durchzuführen. Durch fehlerhafte Implementierung des ORBs kann man bereits mit einfachen Mitteln einen Dienst außer Gefecht setzen. So ist

---

<sup>19</sup>Dies wird oft auch als „security by obscurity“ bezeichnet.

<sup>20</sup><http://www.bastille-linux.org/jay/obscurity-revisited.html>

es gelungen, durch einen zufällig gewählten Objektschlüssel einen MICO-Server-ORB<sup>21</sup> zum Absturz zu bringen, da die Fehlerbehandlung für diesen eher seltenen Fall fehlerhaft war. In Anbetracht der Tatsache, dass viele Objektschlüssel Zeitstempel besitzen, ist diese Maßnahme geeignet den Zeitpunkt des ORB-Starts oder der Objekterzeugung minutengenau vorhersagen zu können. Ein Angriffsziel könnte sein, einen ORB gezielt zum Absturz zu bringen, um dadurch eine bessere Ausgangsposition zu erhalten.

### 2.5 Bestehende Lösungen

Durch den Absturz von MICO und der Existenz von Sicherheitslösungen für CORBA wird deutlich, dass bereits Probleme bei der Sicherheit existieren und weitere vermutet werden können.

Die Frage, welches System per se sicherer ist – eines, das hinter einer gut konfigurierten Firewall steht oder eines, welches „nackt“ über öffentliche Netze erreichbar ist, möge sich jeder selbst beantworten. Wie in [BS02] dargestellt wurde, führt „die nachträgliche Integration von Sicherheitslösungen in bestehende Java- und CORBA-basierte Anwendungen häufig zu praktischen Problemen, die durch vorhandene Firewall-Infrastrukturen noch verstärkt werden.“ Herkömmliche Firewalls arbeiten in der Regel Portbasiert, wenn auf einem Port ausgehender Verkehr (also eine Anfrage von innerhalb des geschützten Bereiches nach Außen) stattfindet, so ist zulässig, wenn hier wieder Daten hereinkommen (Antwort). Bei CORBA erfolgt diese Portzuweisung jedoch dynamisch und führt deshalb zu Problemen. Als Lösung wird hier auf sogenannte IIOP-Proxies oder Application Security Gateways als Application Level Firewalls verwiesen.

---

<sup>21</sup>Betrifft Mico 2.3.12 ohne Security Fix 001.

### 2.5.1 IIOP-Firewalls

Für CORBA kann eine spezielle IIOP-Firewall eingesetzt werden, um den systemspezifischen Belangen gerecht zu werden. Eine IIOP-Firewall implementiert Zugriffskontrollmechanismen die ein- und ausgehende IIOP/-GIOP Nachrichten einer Domäne kontrollieren und somit die Domäne zu schützen. Eine IIOP-Firewall kann auch Parameter überprüfen die an Objektmethoden übergeben werden, wenn sie Kenntnis über die Objekte und deren Schnittstellen besitzt. So wäre es möglich, eine manipulierten Geldautomaten zu ermitteln, der versucht, mehr als EUR 2000 auszuzahlen.<sup>22</sup> Diese Firewall agiert dann auf höheren Schichten<sup>23</sup> als TCP/IP indem es GIOP-Nachrichten analysiert und verändert. So ist es denkbar, dass eine solche Firewall oder ein Gateway vorhandene Objektreferenzen gegen eigene austauscht die besonders gesichert sind.<sup>24</sup> Beim Eintreffen einer Anfrage von Außen, kann somit geprüft werden, ob diese Referenz vom Gateway selbst erstellt wurde oder nicht. Der Aufruf des Objektes wird also sicherer gemacht vergleichen ungeprüfter Weiterleitung.

### 2.5.2 CORBASec

Eine weitere Lösung ist die *CORBA Security* (CORBASec) Spezifikation. Sie erweitert die CORBA-Spezifikation zusätzlich um Schnittstellen für Sicherheitsaspekte. Darin erfolgt eine Spezifizierung folgender Funktionalität: Identifikation und Authentifizierung, Zugriffskontrolle und Autorisierung, Auditing (Überwachung), Datenintegrität, und die Administration aller der genannten Policies. Allerdings hat sich CORBASec bis heute am Markt nicht durchsetzen können und keine Verbreitung erfahren.

---

<sup>22</sup>Wenn die maximale Geldmenge für Auszahlungen EUR 2000 entspricht.

<sup>23</sup>Bezogen auf das im ISO/OSI Schichtenmodell.

<sup>24</sup>US Patent 6981265, <http://www.patentstorm.us/patents/6981265.html>

*Erst zweifeln,  
dann untersuchen,  
dann entdecken.*

Henry Thomas Buckle

# 3

## Material, Methoden und Werkzeuge

In diesem Kapitel werden zunächst die im Rahmen dieser Arbeit genutzten Object Request Broker nach Sprache und Lizenz geordnet vorgestellt. Im Anschluss findet eine kurze Vorstellung der Methoden und Werkzeuge statt, mit deren Hilfe die Interoperablen Objektreferenzen analysiert werden.

### 3.1 Betrachtete Produkte

Die analysierten ORB lassen sich in verschiedene Klassen einteilen. In Tabelle 3.1 werden alle betrachteten ORB nach der Offenlegung ihres Quelltextes und ihrer Programmiersprache eingeordnet.

Tabelle 3.1: Übersicht der benutzten Request Broker

| ORB        | Open Source | Sprache |
|------------|-------------|---------|
| MICO       | ja          | C++     |
| ORBacus    | ja          | JAVA    |
| Orbix      | nein        | JAVA    |
| Sun JDK    | ja          | JAVA    |
| Visibroker | nein        | JAVA    |
| Weblogic   | nein        | JAVA    |
| WebSphere  | nein        | JAVA    |

### 3.1.1 MICO

MICO ist ein zehn Jahre alter, quelloffener und unter GNU Lizenz veröffentlichter ORB, der von Kay Römer und Arno Puder zu Lehrzwecken entwickelt wurde und heute maßgeblich von Object Security<sup>1</sup> weiterentwickelt wird. Der Name ist ein rekursives Akronym und steht für „MICO is CORBA“. Im Rahmen dieser Arbeit wurde mit der Version 2.3.12 mit und ohne Security Fix 001 gearbeitet. Der Quelltext kann unter <http://www.mico.org> heruntergeladen werden. MICO ist „CORBA 2.1 compliant“. <sup>2</sup>

### 3.1.2 ORBacus

Von IONA<sup>3</sup> eingekauft wurde der auch als C++ ORB erhältliche ORBacus. Er unterstützt den CORBA 2.6 Standard, liegt als Quelltext vor und wird über IONA kommerziell vermarktet. Speziell der Support ist bis 2010 auf aktuellen Systemen garantiert. Die Version 4.3 wurde für die Objektschlüsselanalyse genutzt und kann über <http://www.orbacus.com> bezogen werden.

---

<sup>1</sup><http://www.objectsecurity.com>

<sup>2</sup>[http://www.opengroup.org/press/7jun99\\_a.htm](http://www.opengroup.org/press/7jun99_a.htm)

<sup>3</sup><http://www.iona.com>

### 3.1.3 Orbix

Orbix 6 ist der kommerzielle ORB von IONA, der in zwei Varianten gepflegt wird. Einerseits als Versionszweig 3.x mit der Unterstützung von CORBA 2.1 andererseits für Hochlastsysteme im Bereich Telefonie oder Banken als Version 6.3. Genutzt wurde letztere, die Enterprise Edition als SOA/CORBA Alleskönner. Unter <http://www.iona.com/downloads> kann eine Testversion heruntergeladen werden.

### 3.1.4 Sun JDK

JDK steht für JAVA Development Kit, kurz JAVA, und ist eine von James Gosling Anfang der 1990er bei Sun Microsystems entwickelte objektorientierte Programmiersprache. Das Hauptaugenmerk lag auf dem Erreichen einer hohen Plattformunabhängigkeit<sup>4</sup>. Durch die Einführung von Funktionsfernaufrufen (*Remote Method Invocation*, RMI) 1997, wurde JAVA selbst zunächst als Gegenpart zu CORBA positioniert. Mit CORBA 2.2<sup>5</sup> kam dann das Language-Mapping IDL/JAVA und in JAVA wurde die IIOP Unterstützung integriert. Mit der Freigabe der Quelltexte erfreut sich JAVA einer großen Beliebtheit bei Open Source Entwicklern. Jedoch sind sowohl die Lizenzpolitik als auch die Performanz von JAVA umstritten.<sup>6</sup> Das aktuelle JAVA 5 „Tiger“ (eigentlich JAVA 1.5) unterstützt die CORBA 2.3<sup>7</sup> Spezifikation und kann als Bestandteil jeder JAVA-Umgebung unter <http://java.sun.com> bezogen werden.

### 3.1.5 Visibroker

Der Borland Enterprise Server, VisiBroker Edition<sup>8</sup>, wurde in den Versionen 7 und 6.5, JAVA Edition für diese Arbeit analysiert. Als CORBA 2.6 kompatibler Application Server ist die CORBA-Unterstützung im Laufe

---

<sup>4</sup>Sinngemäß aus [http://en.wikipedia.org/wiki/Java\\_programming\\_language](http://en.wikipedia.org/wiki/Java_programming_language)

<sup>5</sup>Siehe [http://www.omg.org/gettingstarted/history\\_of\\_corba.htm](http://www.omg.org/gettingstarted/history_of_corba.htm)

<sup>6</sup>Vgl. [http://en.wikipedia.org/wiki/Java\\_criticisms](http://en.wikipedia.org/wiki/Java_criticisms), Stand 27.08.2006

<sup>7</sup>Vgl. <http://java.sun.com/j2se/1.5.0/docs/guide/idl/compliance.html>

<sup>8</sup><http://www.borland.com/de/products/visibroker/>

der letzten Jahre durch den WebServices-Hype etwas in den Hintergrund getreten. Eine Testversion kann unter <http://borland.com/downloads> heruntergeladen werden.

#### 3.1.6 Weblogic

Der WebLogic Server von BEA ist ein Allrounder, der sich mittlerweile im WebServices Segment ausbreitet. Er bietet eine komplette Portallösung für Unternehmen und ist nicht als ORB zu sehen, wie beispielsweise MICO. Die Version 9.1 wurde als Evaluationskopie via <http://commerce.bea.com> bezogen und verursachte massive Probleme beim Testen. Ein Evaluationsupport war von Bea trotz mehrfacher Nachfrage nicht zu erhalten. Innerhalb der Arbeit konnten deshalb nur transiente Objekte untersucht werden.

#### 3.1.7 WebSphere

Der WebSphere Application Server 6 (WAS) ist Teil der WebSphere Produktlinie vom IBM. Wie Weblogic und Visibroker ist WebSphere eigentlich ein J2EE<sup>9</sup> Application Server, der auch CORBA unterstützt. CORBA wird zum Beispiel für den clientseitigen Anwendungszugriff benutzt.

1998 hieß der WAS noch Servlet Express<sup>10</sup>, war klein und bot nur Servlets an. Im Laufe der Zeit hat sich das Paket immer weiter vergrößert und schließlich die in dieser Arbeit verwandte Version WAS 6.1, bezogen von <http://www.software.ibm.com/webapp/>, erreicht.

### 3.2 Methoden

Um zu analysieren, ob und wie Objektreferenzen erraten werden können, bieten sich zwei Methoden an. Die erste liegt klar auf der Hand und ist der Blick in den Quellcode - die Quelltextanalyse. Als zweite Methode

---

<sup>9</sup>JAVA Platform Enterprise Edition

<sup>10</sup>Siehe [http://en.wikipedia.org/wiki/WebSphere\\_Application\\_Server](http://en.wikipedia.org/wiki/WebSphere_Application_Server), 25.08.2006

eignet sich die Betrachtung der Objektreferenzen selbst, um daraus auf den Erstellungsmechanismus schließen zu können.

#### 3.2.1 Referenzanalyse

Die Analyse von Objektreferenzen setzt voraus, dass der zu analysierende ORB für das Testsystem vorhanden ist und gegebenenfalls über eine Lizenz verfügt. Als Testsystem fungiert ein Ubuntu Debian 6.06 System auf einem AMD Athlon 1800+ mit 512MB Arbeitsspeicher. Für alle lizenzpflichtigen Systeme<sup>11</sup> wurde eine vom Hersteller bereitgestellte Evaluationslizenz genutzt. Die JAVA ORB wurden entweder unter JAVA 5 oder, sofern vorhanden, unter der mitgelieferten JAVA VM<sup>12</sup> betrieben.

Um eine für jeden ORB gleiche Testumgebung sicherstellen zu können, ist für die JAVA ORB eine Testapplikation (siehe Abschnitt 3.3.2) zu entwickeln. C++ ORB sind im Vergleich zu den JAVA Varianten nicht im gleichen Maße portabel, deshalb werden hier ähnliche Beispielprogramme für jeden ORB separat entwickelt.

Um aus den gewonnenen Objektschlüsseln auf die zugrundeliegenden Mechanismen schließen zu können, ist eine wohldefinierte Vorgehensweise erforderlich. Zunächst werden 1000 transiente Objekte mit einem POA erzeugt. Dieser und alle folgenden Schritte werden jeweils zwei Mal wiederholt. So kann bereits im ersten Schritt erkannt werden, welcher Teil des Schlüssels ORB- bzw. POA-abhängig ist und welche Teile sich von Objekt zu Objekt unterscheiden. Aufgrund [SB06] und [Bec06] besteht die Vermutung, dass die Objekte jeweils nur durchgezählt werden.

In einem zweiten Schritt wird mit vier POA gearbeitet, die alle Kinder des RootPOA sind. Daraus lässt sich ermitteln, welche aus dem ersten Schritt vermuteten Teile den ORB und welche den POA identifizieren. Desweiteren kann damit eine Vermutung über die Form der „Objektnummer“ bestätigt werden.

---

<sup>11</sup>Orbix, Visibroker, Weblogic und WebSphere

<sup>12</sup>VM steht für Virtuelle Maschine (engl.: *virtual machine*)

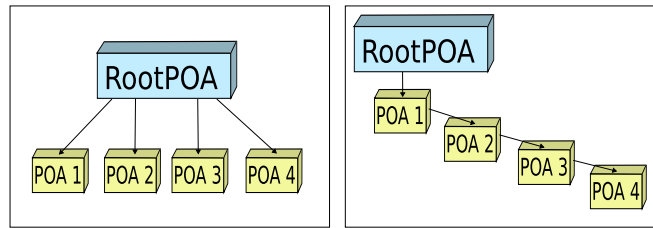


Abbildung 3.1: Anordnung der POA

Anschließend werden die vier POA (Siehe Abbildung 3.1) in einer Hierarchie angeordnet. Wenn sich die POA-Hierarchie in irgendeiner Form im Objektschlüssel widerspiegelt, so ist diese Methode geeignet, dies herauszufinden. Um abschließend getroffene Vermutungen bestätigen zu können, ist die POA-Hierarchie im letzten Schritt auf vier POA anzuwenden, während vier weitere direkt dem RootPOA untergeordnet sind (Kombination der Strukturen aus Abbildung 3.1). Die so zur Verfügung stehenden acht POA lassen eine vergleichende Analyse zu und bieten direkte Vergleichsmöglichkeiten.

### 3.2.2 Quellcodeanalyse

Insofern der zu untersuchende ORB als Quelltext vorliegt, werden die Mechanismen zur Erzeugung der Objektschlüssel direkt dort nachgeschlagen. Für die Bereitstellung eines neuen Objektschlüssels ist bei transienten Objekten in der Regel der POA<sup>13</sup> zuständig. Der Blick in den Quelltext und die Ableitung brauchbarer Ergebnisse setzt Kenntnisse der internen Mechanismen und Programmierparadigmen voraus. Prinzipiell muss nur die POA-Implementierung gefunden und analysiert werden, selten auch die des ORB-Kerns. Die OpenSource-ORB sind in der jeweils benutzten Version auf der beiliegenden CD vollständig enthalten weshalb auf eine umfassende Wiedergabe entsprechende Quelltexte verzichtet und im Regelfall nur darauf verwiesen wird.

<sup>13</sup>Die `IdAssignmentPolicy` ist so eingestellt, dass das System die Schlüssel erzeugen soll.

## 3.3 Werkzeuge

### 3.3.1 Analyseprogramm

Eine IOR in Zeichenkettenform ist schwer zu analysieren, wenn sie nicht in ihre Bestandteile zerlegt wird. Viele davon sind ohnehin für diese Arbeit uninteressant. Ein Programm, mit dem sich eine IOR dekodieren lässt, liegt in der Regel jedem ORB bei. Dadurch werden die einzelnen Komponenten für den Menschen besser verständlich dargestellt. Die Variante des freien JAVA ORB JacORB (*dior*), wird für die Ziele dieser Arbeit so verändert, dass nun auch mehrere IORs in einem Schritt dekodiert werden können. Desweiteren ist es möglich, sich die Objektschlüssel sowohl als ASCII-Text (URL-Form), als auch als HEX-Darstellung anzeigen zu lassen. Mittels Doppelkreuz (#) übergebene Kommentare hinter den IORs bleiben bei der Referenzdekodierung erhalten und erleichtern somit die spätere Zuordnung der Objektschlüssel zur IOR. Dies ist insbesondere bei vielen POA notwendig.

Ein zweites Analyseprogramm, *orbinfo*, ebenfalls basierend auf JacORB, soll anhand der Ergebnisse der Arbeit implementiert werden. Es wird zu einer IOR oder einem Objektschlüssel möglichst viele Informationen ermitteln wie beispielsweise der ORB-Typ, die Erstellungszeit und viele mehr.

### 3.3.2 Testprogramm

Die zu entwickelnden Testprogramme zur Objektschlüsselgenerierung sollen grundsätzlich wie folgt vorgehen:

1. Auswertung der Kommandozeilenargumente
2. Erzeugung des ORB
3. Erzeugung der POA
4. Erzeugen der Servants (Objektinkarnationen)
5. Eintritt in die Generatorschleife

- a) Aktivierung der Objekte
- b) Ermittlung der Referenz, Schreiben der IOR
- c) Deaktivierung der Objekte

6. Aufräumen

7. Ausgabe der Statistik

Die POA können jeweils für das Binden persistenter und transienter Objekte konfiguriert werden. Desweiteren ist es möglich, die POA in einer Hierarchie aufzubauen. Eigene Versuche haben gezeigt, dass sich an der prinzipiellen Systematik keine Änderungen ergeben, wenn mehr als ein POA pro Ebene aktiv ist.

Die Anzahl erzeugbarer Objekte wird programmtechnisch auf  $10^6$  begrenzt, da in der Praxis keine größeren Mengen an Objekten zur POA-Laufzeit zu erwarten sind.

Weiterhin können persistente und transiente Objekte erzeugt werden, jedoch nicht gleichzeitig. Eine Ausnahme hierzu stellt der RootPOA dar, der immer nur transiente Objekte verwalten kann. Um zeitliche Abhängigkeiten innerhalb der Schlüssel besser finden zu können, muss eine Einstellung existieren um die Erzeugung einzelner POAs oder Objektinkarnationen zeitlich steuern zu können. Ebenfalls möglich und für das Testen selbsterzeugter Objektreferenzen vorteilhaft, ist die Möglichkeit den ORB laufen zu lassen (`ORB->run()`) und somit für Anfragen von Außen erreichbar zu machen.

Dieses Testprogramm wird in einer Machbarkeitsanalyse die Serverobjekte erzeugen und als Server fungieren. Im Rahmen dieser Analyse wird ein Programm zu entwickeln sein, das auf den Ergebnissen dieser Arbeit basiert und den Zugriff auf andere Objekte mit Hilfe von selbst erzeugten Objektreferenzen ermöglicht.

# 4

## Ergebnisse

Die durch die Analyse gewonnenen Ergebnisse werden nach ORB getrennt betrachtet. Des Weiteren sind bei den OpenSource-ORB die Ergebnisse der Referenzanalyse und die der Quelltextanalyse nicht separat angefügt. Zur besseren Vergleichbarkeit der Aufwandsabschätzung für das „reverse engineering“ der zugrunde liegenden Algorithmen wurde zunächst jedoch für alle Request Broker die Referenzanalyse durchgeführt. Eine Zusammenfassung der Ergebnisse für alle ORB erfolgt im letzten Abschnitt dieses Kapitels.

Das Format der Objektschlüssel wird zunächst graphisch dargestellt und im Anhang mit Hilfe einer kontextfreien Grammatik<sup>1</sup>, der *Erweiterten Backus-Naur-Form*<sup>2</sup> (EBNF), beschrieben.

Die angegebenen Zahlen für die verschiedenen Möglichkeiten, aus denen

---

<sup>1</sup>Bezogen auf die Chomsky-Hierarchie ist die Grammatik  $G \in \text{Typ}_2$ .

<sup>2</sup>Die Erweiterte Backus-Naur-Form ist in ISO/IEC 14977:1996(E) standardisiert. Siehe auch <http://www.cs.man.ac.uk/~pjj/bnf/bnf.html#EBNF>.

ein Schlüssel bestehen kann, sind mit Annahmen versehen. Dies dient im Rahmen dieser Arbeit der besseren Vergleichbarkeit der ORB untereinander. Die Zahlen geben die jeweils maximal vermuteten Möglichkeiten wieder, die Zeiten dahinter stellen lediglich einen Anhaltspunkt dafür dar, wie lange man maximal benötigt, um den gesamten Suchraum<sup>3</sup> zu durchsuchen. In der Realität wird, wenn Gleichverteilung vorliegt, nur die Hälfte davon benötigt werden. Versuche haben gezeigt, dass die Suche am Anfang des Wertebereiches bei Zählern mit frühen Erfolgen verbunden ist. Dies folgt aus der Natur eines Zählers.

## 4.1 MICO

### 4.1.1 Transiente Objekte

Bei der Referenzanalyse wird nach dem in Abschnitt 3.2.1 beschriebenen Verfahren vorgegangen. Im Interesse der Übersichtlichkeit wird auf eine allzu detaillierte Darstellung verzichtet und auf die der Arbeit beiliegenden Analyserohdaten verwiesen.<sup>4</sup>

Um eine Vorstellung von einem MICO-Objektschlüssel zu bekommen, ist nachfolgend einer in HEX- und URL-Darstellung zu sehen:

```
2F 31 35 37 32 2F 31 31 35 31 33 33 36 32 31 32 2F 30 2F 5F 30  
/1572/1151336212/0/_0
```

Erzeugt man mit Hilfe des Testprogramms viele IOR und extrahiert den Objektschlüssel in URL-Form<sup>5</sup>, so kann man schnell auf den Mechanismus schließen, der einzelne Objekte voneinander unterscheidet. Die Objektschlüssel in Tabelle 4.1 unterscheiden sich nur in der letzten Stelle bzw. den Stellen hinter dem letzten Schrägstrich. Dies ist ein Zähler für das laufende Objekt.

Im weiteren Analyseverlauf wird nur noch die URL-Form benutzt, da offensichtlich der gesamte Schlüssel als Zeichenkette behandelt wird, die aus

---

<sup>3</sup>Der Suchraum ist der gesamte Wertebereich für mögliche gültige Schlüssel.

<sup>4</sup>Siehe CD am Ende dieser Arbeit.

<sup>5</sup>Aufruf: `dior -f datei-mit.iors -u > datei-fuer.urls`

einzelnen numerischen Komponenten besteht. Der gefundene Objektzähler

Tabelle 4.1: Transiente MICO Objektschlüssel

| Objekt | Objektschlüssel         |
|--------|-------------------------|
| 0      | /1572/1151336212/0/_0   |
| 1      | /1572/1151336212/0/_1   |
| 2      | /1572/1151336212/0/_2   |
| 3      | /1572/1151336212/0/_3   |
| 10     | /1572/1151336212/0/_01  |
| 99     | /1572/1151336212/0/_99  |
| 100    | /1572/1151336212/0/_001 |
| 101    | /1572/1151336212/0/_101 |

verhält sich bei der Benutzung mehrerer POA (Siehe Tabelle 4.2<sup>6</sup>) als ORB-weiter Objektzähler. Die vorletzte Stelle wird für die Identifikation des POA benutzt. Dabei ist diese Stelle immer numerisch und unabhängig vom POA-Namen oder dessen Stellung in einer vermeintlichen Hierarchie. Es handelt sich um einen POA-Zähler.

Tabelle 4.2: Transiente MICO Objektschlüssel mehrerer POA

| POA | Objekt | Objektschlüssel       |
|-----|--------|-----------------------|
| 1   | 0      | /1572/1151336212/0/_0 |
| 2   | 0      | /1572/1151336212/1/_1 |
| 3   | 0      | /1572/1151336212/2/_2 |
| 1   | 1      | /1572/1151336212/0/_3 |
| 2   | 1      | /1572/1151336212/1/_4 |
| 3   | 1      | /1572/1151336212/2/_5 |

Die erste und zweite Stelle des Schlüssels (jeweils durch einen Schrägstrich getrennt) lassen Raum für Mutmaßungen. Während der erste Teil immer drei- bis fünfstellig ist (Beispiele: 1572, 32669, 316, 763, 1246), so ist der zweite Teil immer konstant zehnstellig und beginnt in der Regel mit 11. Begibt man sich auf die Suche nach ähnlichen Werten, so kommt man

<sup>6</sup>Aus [mico-200606261736.iors.oid.url](http://mico-200606261736.iors.oid.url)

zur Vermutung, dass es sich um einen Zeitstempel handelt. Die Ursache hierfür ist die Korrelation des Wertes mit dem UNIX Timestamp<sup>7</sup> korreliert. Umgerechnet ergibt dieser Stempel den 26.06.2006 um 17:36 Uhr, die Startzeit des ORBs. Der erste Teil wächst bei aufeinanderfolgenden Versuchen stetig, nimmt bei späteren Versuchen jedoch scheinbar willkürlich einen geringen Wert an und steigt erneut. Zunächst könnte man einen Zufallswert in den Grenzen von 1-99999 vermuten, doch steht dies mit der vorhandenen Stetigkeit im Widerspruch. Näheres ist nur durch die Quelltextanalyse zu klären.

Die Quelltextanalyse kann einerseits die Ergebnisse der Referenzanalyse untermauern, andererseits auch bisher Verborgenes an den Tag bringen. Auch liefert sie Hinweise darauf, von welcher Qualität die Referenzanalyse ist und worauf möglicherweise bei den nachfolgenden ORB geachtet werden sollte.

Die Suche nach dem Objektschlüssel beginnt praktischerweise in der POA-Implementation, da die Objekte am POA registriert werden. Betrachtet man dazu die Quelltexte A.3 und A.4, findet man in ersterem einen Zählmechanismus und im zweiten den Hinweis auf die ORB-Erstellungszeit und die Prozess-ID des ORB. Diese ist die erste, durch die Referenzanalyse nicht aufklärbare Variable.

Mit diesem Wissen ergibt sich folgender Aufbau für transiente Objektschlüssel des MICO-ORB<sup>8</sup>:



Abbildung 4.1: Aufbau transienter MICO-Objektschlüssel

#### 4.1.2 Persistente Objekte

Der Aufbau von Schlüsseln persistenter Objekte ist leicht verändert gegenüber dem transienter Objekte. Tabelle 4.3 zeigt die ersten drei Schlüssel

<sup>7</sup>Anzahl der Sekunden seit dem Beginn der UNIX Ära am 1.1.1970 1:00Uhr.

<sup>8</sup>Eine detaillierte Darstellung in EBNF ist im Anhang zu finden.

eines POAs mit solchen Objekten. Man erkennt, dass in diesem Fall der Schlüssel mit einem Text beginnt. „Default“ ist der Standard-ORB-Name. Ein solcher Name wird zur besseren Identifikation an ORB vergeben, damit sie später auch noch für die Aufrufzustellung auffindbar sind. Nach dem Trennzeichen findet man den Namen des POAs an dem der Servant angemeldet ist, anschließend vermutlich wieder die Prozessnummer des ORBs und dessen Erstellungszeit.

Liegt eine POA-Hierarchie vor, so findet man auch die gesamte Hierar-

Tabelle 4.3: Persistente MICO Objektschlüssel

| lfd. Objekt | Objektschlüssel                             |
|-------------|---|
| 0           | Default/mypoaeins/%5C/23271%5C/1151920396_0 |
| 1           | Default/mypoaeins/%5C/23271%5C/1151920396_1 |
| 2           | Default/mypoaeins/%5C/23271%5C/1151920396_2 |

chie im Schlüssel wieder (siehe Tabelle 4.4). Der hintere Abschnitt folgt ähnlichen Prinzipien wie in Abschnitt 4.1.1 beschrieben. Konsultiert man

Tabelle 4.4: Persistente MICO OKs in POA-Hierarchie

| Objektschlüssel              |                          |
|------------------------------|--------------------------|
| Default/eins/                | %5C/6940%5C/1157548901_0 |
| Default/eins/zwei/           | %5C/6940%5C/1157548901_1 |
| Default/eins/zwei/drei/      | %5C/6940%5C/1157548901_2 |
| Default/eins/zwei/drei/vier/ | %5C/6940%5C/1157548901_3 |

die gleichen Quellen wie bereits für die Analyse transienter Objekte, so bestätigt sich der vermutete Aufbau. Als Anlaufpunkt dient hier wieder die POA-Implementation. Persistente Objektschlüssel des MICO-ORB sehen damit wie folgt aus:

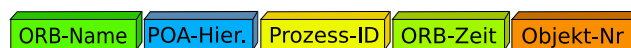


Abbildung 4.2: Aufbau persistenter MICO-Objektschlüssel

### 4.1.3 Fazit

Für das Herausfinden eines MICO Objektschlüssels ohne Vorwissen, muss bei persistenten Objekten eine POA-Hierarchie „erraten“ werden. Nimmt man an, dass die gesamte Zeichenkette nicht länger als zehn Zeichen ist und nur aus Groß- und Kleinbuchstaben, sowie Zahlen besteht, ist die Wahrscheinlichkeit<sup>9</sup> hier einen Treffer zu landen  $1 : 62^{10}$ , also aussichtslos. Man ist darauf angewiesen, den Suchbereich der POA-Namen eingrenzen zu können. Da zuvor angenommen wurde, dass auch die TypeID bekannt ist – man will ja mit dem Objekt interagieren – so wird im Folgenden für das Erraten der Hierarchie eine Wahrscheinlichkeit von  $1 : 1,28 \times 10^6$  angesetzt. Dieser Wert ergibt sich aus der Überlegung, das bekannt sein könnte, dass die POA-Hierarchie maximal aus drei Ebenen (unterhalb des RootPOA) besteht und die POA-Namen aus einem Pool von 50 Möglichkeiten entnommen werden. Dies soll einen Wörterbuchangriff simulieren. Die POA-Namen könnten beispielsweise durch ehemalige Mitarbeiter bekannt sein. Bei persistenten Objekten befindet sich der ORB-Name im Objektschlüssel wieder. Der Einfachheit halber sei angenommen, dass der Default-Name benutzt wird, andernfalls ergibt sich eine ähnliche Komplexität wie bei der POA-Hierarchie. Da bei transienten Objekten kein POA-Name, sondern nur ein POA-Zähler vorhanden ist, sei dieser auf  $10^3$  Möglichkeiten begrenzt. Diese Grenze ist zwar willkürlich, jedoch ist nicht anzunehmen, dass sie überschritten wird.

Um die Prozess-ID eines ORBs erraten zu können, muss der Bereich in dem diese ID liegen kann bekannt sein. Dieser ist jedoch vom Betriebssystem abhängig. Die benutzte Debian 3.1 Linux Distribution zählt diese bis  $2^{16} = 65536$  hoch und beginnt wieder von vorn. Werden insgesamt zur Lebenszeit des Servers<sup>10</sup> nur sehr wenige Prozesse erzeugt bzw. wenige neu erzeugt so ist die Wahrscheinlichkeit am Anfang des Wertebereiches am größten und nimmt zum Ende hin ab.

Die Erstellungszeit des ORB sollte geschätzt werden können. Nimmt

---

<sup>9</sup>26 Klein- plus 26 Großbuchstaben plus 10 Ziffern

<sup>10</sup>Gemeint ist hier die Zeit zwischen Ein- und Ausschalten des Rechners.

man dafür eine Genauigkeit von einem Tag (86400 Sekunden) an, so ergeben sich 86400 verschiedene Möglichkeiten. Der Aufwand, diese Zahl zu erraten steigt also linear mit dem Zeitraum an, der durchsucht werden soll. Die Anzahl der Objekte, die an einem gültigen POA ausprobiert werden müssen, sei auf  $10^5$  begrenzt. Dabei sei gleichzeitig angenommen, dass 10 Prozent der vermuteten Objekte noch aktiv sind.

Eine gängige Praxis zur Erzeugung von Objekten oder Objektadaptern gibt es nicht. Untersuchungen darüber existieren nicht und dürften auch keinen Konsens finden. Die jeweilige Objektmenge pro Zeit ist immer abhängig vom Gesamtsystem und dem Gesamtkonzept. Die Zahl der Objekte ist insofern willkürlich gewählt. Oftmals ist es so, dass diese Zahl linear abhängig von der Lebenszeit des POAs ist. Dies soll nachfolgend außer Acht gelassen werden, da viele Variablen bereits geschätzt sind und das Betrachten dieser Abhängigkeit eine nicht vorhandene Genauigkeit vortäuschen würde.

Die Gesamtkomplexität eines MICO Objektschlüssels beträgt:

**Transiente Objekte:**

$$65536 \times 86400 \times 10 = 5,6 \times 10^{10} \equiv 648d$$

**Persistente Objekte:**

$$1,28 \times 10^6 \times 65536 \times 86400 \times 10 = 7,2 \times 10^{16} \equiv 2,3 \times 10^6a$$

Die Angaben spiegeln jeweils die maximale Anzahl der Versuche wider, die nötig sind, um einen gültigen Objektschlüssel zu generieren. Dahinter ist angegeben, wie lange man dafür benötigt, wenn ein Versuch pro Millisekunde durchgeführt werden kann [Bec06].

Die mittlere Wahrscheinlichkeit, einen transienten MICO-Objektschlüssel zu erraten, beträgt  $1 : 2,8 \times 10^{10}$ . Nimmt man an, man sei im Besitz der technischen Lösung, die 1000 Mal pro Sekunde einen Schlüssel ausprobieren kann, so würde man durchschnittlich 324 Tage auf das Erraten verwenden. Bei einem persistenten Objekt ist der Aufwand ca.  $10^6$  Mal höher und liegt damit gänzlich im Bereich des Unmöglichen.

Gelangt man aber in den Besitz einer IOR, so sind folglich die ORB-Zeit,

der ORB-Namen, die POA-Hierarchie und die Prozess-ID bekannt. Die Wahrscheinlichkeit des Erratens sinkt dramatisch. Die Komplexität, also die Anzahl der maximalen Versuche um ein weiteres Objekt am selben POA zu erraten sinkt für alle Objekte auf 10, im Mittel also 5.

Innerhalb von 5ms ist ein weiteres aktives Objekt an diesem POA zu finden. Um alle  $10^5$  Objekte zu testen, benötigt man 100s.

Zum Auffinden von Objekten an einem anderen POA dieses ORBs, muss man zunächst nur die POA-Hierarchie variieren. Im umfangreichsten Fall kann man  $1 : 1,28 \times 10^6$  Möglichkeiten dafür ansetzen. Man benötigt im Mittel dreieinhalb Stunden um ein gültiges Ergebnis zu erhalten. Dies setzt allerdings voraus, dass man nicht erkennen kann, ob ein POA existiert und für jeden (eventuell auch ungültigen) POA alle Objekte ausprobiert werden müssten. Bei MICO ist dies nicht der Fall, denn er liefert die Ausnahme<sup>11</sup> `OBJECT_NOT_EXIST`, wenn der POA nicht existiert und `OBJ_ADAPTER`, wenn das Objekt nicht existiert.<sup>12</sup> Berücksichtigt man dies bei obigen Berechnungen, so verringern sich diese um den Faktor 10.

Die Gesamtkomplexität unter Berücksichtigung möglicher Optimierungen eines MICO Objektschlüssels ergibt sich wie folgt, wenn keine Daten bekannt sind:

**Transiente Objekte ( $t_1$ ):**

$$65536 \times 86400 = 5,6 \times 10^9 \equiv 64,8\text{d}$$

**Persistente Objekte ( $t_2$ ):**

$$1,28 \times 10^6 \times 65536 \times 86400 = 7,2 \times 10^{15} \equiv 228310\text{a}$$

Falls eine IOR bekannt ist, sind viele Variablen gegeben und müssen nicht mehr evaluiert werden. Die Möglichkeiten reduziert sich auf nachfolgende Werte und benötigen in etwa:

**Objekte am gleichen POA ( $t_3$ ):**

$$10 \equiv 10\text{ms}$$

---

<sup>11</sup>Exception

<sup>12</sup>Man würde hier jedoch erwarten, dass `OBJECT_NOT_EXIST` dann geworfen wird, wenn nur das Objekt nicht existiert und `OBJ_ADAPTER` in allen anderen Fällen.

**Objekte an anderen transienten POAs ( $t_4$ ):**

$$10^3 \equiv 1\text{s}$$

**Objekte an anderen persistenten POAs ( $t_5$ ):**

$$1,28 \times 10^6 \equiv 21\text{min}$$

Ist eine gültige Objektreferenz bekannt, so ist es relativ leicht möglich, andere gültige Objektreferenzen zu erzeugen.

## 4.2 ORBacus

Die Darstellung der ORBacus Objektschlüssel erfolgt byteweise als HEX-Wert. Dort, wo es sinnvoll erscheint, wird gleichzeitig auf die URL-Form zurückgegriffen.

### 4.2.1 Transiente Objekte

In Tabelle 4.5 sieht man einen transienten Objektschlüssel an POA „null“ (Kind von RootPOA) in HEX- und URL-Darstellung. Man kann erkennen, dass die POA-Hierarchie (RootPOA/null) einen Teil des Schlüssels ausmacht. Die einzelnen Elemente der Hierarchie werden offensichtlich mittels Nullbyte<sup>13</sup> verbunden. Auffällig ist auch die Zeichenkette (ab Byte fünf): 1'146'423'159. Sie legt die Vermutung nahe, dass es sich um einen Zeitstempel handelt. Der Grund dafür ist, dass sie mit dem UNIX Timestamp synchron läuft. Umgerechnet ergibt dieser Stempel den 30.04.2006 20:52Uhr - die Startzeit des ORBs. Die ersten vier Byte stellen scheinbar ein magic<sup>14</sup> dar, das den ORB identifiziert. Bei der Analyse aufeinander folgender Objektschlüssel bemerkt man, dass sich nur ein kleiner Teil des Schlüssels ändert. In Tabelle 4.6 ist zu sehen, dass die letzten Stellen einen linearen Zähler darstellen. Die Bytes vor der POA-Zeit, CA FE BA BE, sind

---

<sup>13</sup>Ein Nullbyte (0x00) wird in der Programmiersprache C häufig für das Terminieren einer Zeichenkette benutzt.

<sup>14</sup>Als magic werden oft Zeichenketten benannt, die implementierungs- oder versionsabhängige Zustände oder Ähnliches definieren.

Tabelle 4.5: Transienter ORBacus Objektschlüssel

| HEX                                       | URL             |
|---|-----------------|
| AB AC AB 31 31 31 34 36 34 32 33 31 35 39 | ...11146423159  |
| 00 5F 52 6F 6F 74 50 4F 41 00 6E 75 6C 6C | .._RootPOA.null |
| 00 00 CA FE BA BE 44 55 07 77 00 00 00 00 | .....DU.w.....  |

bereits als JAVA-magic bekannt und tauchen in allen ORBacus Objektschlüsseln als Konstante auf. Davor sind jeweils zwei Nullbytes vorhanden. Tabelle 4.7 zeigt die letzten acht Byte von Objektschlüsseln des ersten Objektes an verschiedenen POA. Dabei ist es für diesen Teil belanglos, ob diese vom gleichen ORB erzeugt wurden oder nicht. Es ändern sich jeweils maximal vier Bytes. Es deutet also darauf hin, dass der erste Teil der letzten acht Byte POA- und die letzten vier der acht Byte Objekt-spezifisch sind. Die letzten vier Byte entsprechen der laufenden Objektnummer innerhalb des POAs. Konvertiert man die ersten vier der letzten acht Bytes (0x44 55

Tabelle 4.6: Teile transienter ORBacus Objektschlüssel

| lfd. Objekt | letzten acht Byte       |
|-------------|-------------------------|
| 0           | 44 55 07 6E 00 00 00 00 |
| 1           | 44 55 07 6E 00 00 00 01 |
| 2           | 44 55 07 6E 00 00 00 02 |
| 16          | 44 55 07 6E 00 00 00 10 |
| 255         | 44 55 07 6E 00 00 00 FF |
| 256         | 44 55 07 6E 00 00 01 00 |

Tabelle 4.7: Transiente Schlüsselteile verschiedener POAs

| POA | letzten acht Byte       |
|-----|-------------------------|
| 1   | 44 55 07 6E 00 00 00 00 |
| 2   | 44 53 58 98 00 00 00 00 |
| 3   | 44 55 07 77 00 00 00 00 |

07 6E) in Dezimaldarstellung, so erhält man 1'146'423'150. Beachtet man

jedoch, dass diese vier Byte ständig wachsen und das sich dieses Wachstum langsam vollzieht, so kann man vermuten, dass Zeitabhängigkeit vorliegt. Ist 1'146'423'150 ebenfalls die Anzahl der Sekunden seit dem 1.1.1970, so bezeichnet dieser Wert den 30.04.2006 um 20:52Uhr. Dies entspricht dem Zeitpunkt der POA-Erstellung.

Daraus schlussfolgernd läßt sich folgender Aufbau<sup>15</sup> für transiente ORBacus Objektschlüssel feststellen:



Abbildung 4.3: Aufbau transienter ORBacus-Objektschlüssel

Die vollständige und detailliertere Darstellung befindet sich ebenfalls im Anhang.

#### 4.2.2 Persistente Objekte

In Tabelle 4.8 sieht man einen persistenten Objektschlüssel. Im Vergleich zum transienten fällt auf, dass das vierte Byte statt 0x30 den Wert 0x31 besitzt. Ab dem fünften Byte folgt dann sofort die POA-Hierarchie, ebenfalls durch zwei Nullbytes terminiert. Es schließt sich die bekannte Kombination aus JAVA-magic, POA-Zeit und Objekt-Nummer an. Betrachtet man mehrere Objektschlüssel<sup>16</sup>, so bestätigt sich diese Vermutung. Ein persis-

Tabelle 4.8: Persistenter ORBacus Objektschlüssel

| HEX                                 | URL          |
|-------------------------------------|--------------|
| AB AC AB 30 5F 52 6F 6F 74 50 4F 41 | ...0_RootPOA |
| 00 6E 75 6C 6C 00 00 CA FE BA BE    | .null.....   |
| 44 A8 13 6A 00 00 00 00             | D.....       |

tenter ORBacus Objektschlüssel kann nach folgender Bildungsvorschrift

<sup>15</sup>Darstellung in EBNR.

<sup>16</sup>Wie in der Datei orbacus\_060702\_2051.iors.pers2p.oid auf der CD.

erzeugt werden:

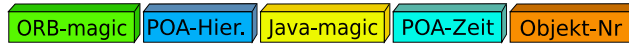


Abbildung 4.4: Aufbau persistenter ORBacus-Objektschlüssel

### 4.2.3 Fazit

Für das Erraten eines ORBacus Objektschlüssels ohne Vorwissen werden die gleichen Annahmen wie in Abschnitt 4.1.3 getroffen.

Die Erstellungszeit des POA stellt keine große Hürde dar. Ein Tag hat 86400 Sekunden also 86400 verschiedene POA-Erstellungszeiten. Der Aufwand, diese Zahl zu erraten, steigt also linear mit der Zeit, bzw. dem Zeitraum, der für die Erstellung vermutet wird. Die Anzahl der Objekte ist wieder auf  $10^5$  begrenzt.

Die Gesamtmenge der Möglichkeiten sieht wie folgt aus, wenn ORB und POA-Erstellung auf einen Tag genau vorhergesagt und die POA-Namen eingegrenzt werden können:

**Transiente Objekte ( $t_1$ ):**

$$86400^2 \times 1,28 \times 10^6 \times 10 = 9,6 \times 10^{16} \equiv 3 \times 10^6 \text{a}$$

**Persistente Objekte ( $t_2$ ):**

$$86400 \times 1,28 \times 10^6 \times 10 = 1,1 \times 10^{12} \equiv 34,9 \text{a}$$

Bei object keys persistenter Objekte ist die mittlere Wahrscheinlichkeit  $1 : 5 \times 10^{11}$ . Wenn man pro Millisekunde einen Versuch unternehmen kann, so wäre man im Mittel rund 16 Jahre beschäftigt. Ist nun durch eine IOR, die ORB-Zeit, die POA-Hierarchie und POA-Zeit bekannt, so müssen in beiden Fällen nur noch alle Objekte getestet werden. Die Anzahl der Möglichkeiten und die Zeit um diese auszuprobieren sinkt auf:

**Objekte am gleichen POA ( $t_3$ ):**

$$10 \equiv 10 \text{ms}$$

**Objekte an einem anderen POA ( $t_4, t_5$ ):**

$$86400 \times 1,28 \times 10^6 = 1,1 \times 10^{11} \equiv 3,5a$$

Allerdings bleibt hierbei unberücksichtigt, dass die Zeit zwischen der Erstellung beider POA weniger als 24h beträgt. Geht man von fünf Minuten aus, so ist man in ( $t_6 =$ ) 4,4 Tagen am Ziel.<sup>17</sup>

**4.3 Orbix**

Durch Probleme bei der Erzeugung persistenter Objekte können nur transiente Objektschlüssel betrachtet und in HEX-Darstellung analysiert werden. Der Analysegegenstand, ein transienter Orbix Objektschlüssel ist nachfolgend dargestellt:

```
3A 3E 02 31 31 0C 78 3B 83 D7 33 46 E8 61
7F 30 D6 B1 08 00 00 00 00 00 00 01
```

Zum besseren Verständnis vergleicht man (Tabelle 4.9) zwei aufeinanderfolgende Schlüssel, die Zugunsten der Übersicht willkürlich zerlegt wurden. Man erkennt sofort, dass sich nur die letzte Stelle verändert hat. Wie

Tabelle 4.9: Zwei Transiente Orbix Objektschlüssel

| Eins                    | Zwei                    |
|-------------------------|-------------------------|
| 3A 3E 02 31 31 0C 78 3B | 3A 3E 02 31 31 0C 78 3B |
| 83 D7 33 46 E8 61 7F 30 | 83 D7 33 46 E8 61 7F 30 |
| D6 B1 08 00 00 00 00 00 | D6 B1 08 00 00 00 00 00 |
| 00 00 01                | 00 00 02                |

aus Abschnitt 4.1 und Abschnitt 4.2 bereits bekannt, könnte es sich hier ebenfalls um einen Objektzähler handeln. Tabelle 4.10 zeigt die letzten acht Stellen von Objektschlüsseln eines POAs. Die Vermutung mit dem Objektzähler hat sich bestätigt. Als nächstes ist die Unterscheidung verschiedener POA von Interesse. MICO und ORBacus haben hierfür die

<sup>17</sup> $5 \times 60 \times 1,28 \times 10^6 = 3,8 \times 10^8$   
 $3,8 \times 10^8 \div 1000 \div 3600 \div 24 = 4,444d$

Tabelle 4.10: Orbix Objektschlüsselabschnitte

| Objektnr. | Schlüsselabschnitt      |
|-----------|-------------------------|
| 0         | 00 00 00 00 00 00 00 01 |
| 1         | 00 00 00 00 00 00 00 01 |
| 2         | 00 00 00 00 00 00 00 01 |
| 3         | 00 00 00 00 00 00 00 01 |
| 255       | 00 00 00 00 00 00 00 FF |
| 256       | 00 00 00 00 00 00 01 00 |
| 1024      | 00 00 00 00 00 00 04 00 |

POA-Hierarchie im Klartext in den Schlüssel integriert. Bei Orbix ist dies offensichtlich nicht der Fall. Tabelle 4.11 zeigt fünf Schlüsselabschnitte<sup>18</sup> verschiedener POA. Während die ersten sechs Byte konstant sind, tre-

Tabelle 4.11: Orbix - verschiedene POAs

| lfd. Nr. | Schlüsselabschnitt                  |
|----------|-------------------------------------|
| 1        | 78 3B 83 D7 33 46 E8 61 7F 30 D6 B1 |
| 2        | CC 31 9B 3F 69 39 B1 6C 91 06 C7 C1 |
| 3        | 2C 42 80 B1 85 5F 72 5B E2 E4 71 A4 |
| 4        | E3 00 BB 09 FA 19 5B 6C 05 F5 60 0A |
| 5        | CB 9C EE 5A F7 6B A5 84 34 1B 67 3A |

ten vom siebten bis zum 18. Byte Varianten auf, die keinem Algorithmus zu folgen scheinen. Es könnte sich um eine Zufallsbytefolge handeln oder aber um den Hash bzw. Teil eines Hashes des POA-Namens oder der POA-Hierarchie. Erzeugt man aber die gleichen POA nochmal (also mit gleichem Namen), so differiert dieser Wert ebenfalls.

Die Bedeutung der ersten sechs Bytes ist momentan noch unbekannt. Sicher ist nur, dass diese für transiente Objektschlüssel konstant 3A 3E 02 31 31 0C sind. In einer über ein Internetforum bezogenen IOR eines persistenten Objektes ist der Schlüsselteil 3A 3E 02 31 32 0C. Denkt man kurz über die CDR, die Transfersyntax, nach, so fällt auf, dass die 02

<sup>18</sup>Jeweils Byte sieben bis 18 Byte.

der Länge der nachfolgenden Zeichenkette 31 31 bzw. 31 32 entspricht, während 0C die Länge des POA-Schlüssels anzeigt. Dementsprechend wäre 08 die Länge des Objektzählers. Mag dies alles Zufall sein, so funktioniert es doch gut.



Abbildung 4.5: Aufbau Orbix-Objektschlüssel

Ein Orbix Objektschlüssel bietet nicht viele Dinge, die zu erraten sind. Die POA-ID ist mit zwölf Byte sehr lang und ergibt  $256^{12} = 8 \times 10^{28}$  Möglichkeiten. In diesem Fall ist der Versuch des Erratens vollkommen zwecklos. Die Chancen verbessernden Annahmen können nicht gemacht werden.

Der Objektzähler hingegen erlaubt es, alle Objekte eines POAs abzusuchen. Nachfolgend ist die Gesamtzahl der Möglichkeiten unter den bekannten Annahmen für die Objekte aufgeführt:

**Transiente und/oder persistente Objekte ( $t_{1,4}$ ):**

$$256^{12} \times 10 = 8 \times 10^{29} \equiv 2,5 \times 10^{19} \text{a}$$

**Objekte am gleichen POA ( $t_3$ ):**

$$10 \equiv 10 \text{ms}$$

Orbix kann als resistent gegenüber dem Erraten ohne Vorwissen angesehen werden. Ebenso ist es, statistisch gesehen, unmöglich eine POA-ID zu erraten. Die Ermittlung weiterer Objekte an einem bekannten POA ist allerdings genauso gering, wie bei allen vorgenannten ORB.

## 4.4 Sun ORB

Bevor mit der Analyse begonnen wird, muss man sich ins Gedächtnis rufen, dass der Sun JAVA ORB bei der Veröffentlichung von JAVA 1.5 Tiger ein Redesign erfuhr. Ob dies auch die Objektschlüssel betrifft, soll mit Hilfe von Testdaten der 1.4.2er Version ermittelt werden. Nachfolgend wird nur

der 1.5er ORB betrachtet, auf den 1.4.2er wird gegebenenfalls explizit hingewiesen.

Zunächst betrachtet man in Tabelle 4.12 den Objektschlüssel in HEX- und URL-Darstellung mit jeweils zwölf Byte pro Zeile (nicht-druckbare Zeichen sind durch einen Punkt gekennzeichnet). Man erkennt, dass die POA-Struktur im Schlüssel abgelegt wird und selbiger anscheinend mit einem magic beginnt (Byte 1-3).

Tabelle 4.12: Transienter SUN ORB Objektschlüssel

| HEX                                 | URL         |
|-------------------------------------|-------------|
| AF AB CB 00 00 00 00 20 61 F5 C2 47 | .....       |
| 00 00 00 01 00 00 00 00 00 00 00 02 | .....       |
| 00 00 00 08 52 6F 6F 74 50 4F 41 00 | ...RootPOA. |
| 00 00 00 05 6E 75 6C 6C 00 00 00 00 | ...null.... |
| 00 00 00 08 00 00 00 01 00 00 00 01 | .....       |
| 14                                  | .           |

Vergleicht man nun Schlüssel verschiedener Objekte desselben POA, so stellt man auch hier fest, dass an einer Stelle (dem vorletzten Byte) hochgezählt wird (Tabelle 4.13). Variiert man zusätzlich den POA, so ändert sich das sechste Byte von hinten derart, dass es für jeden neuen POA inkrementiert. Erzeugt man nun an einem ORB verschiedene POA und für diese jeweils mehrere Objekte, so können diese Abhängigkeiten gut gefunden werden. In Tabelle 4.13 sind auf die letzten zehn Byte reduzierte Objektschlüssel dargestellt. Byte fünf ändert sich pro POA-Änderung und Byte neun pro Objektänderung. Was sich ebenfalls ändert, sind die POA-Namen im Objektschlüssel. Beginnend ab Byte 28 findet sich der RootPOA und anschließend der jeweils zuständige POA. Es handelt sich hierbei um CDR kodierte Zeichenketten. Im long-Wert, vier Bytes vor der Zeichenkette, wird deren Länge zuzüglich des Nullbytes kodiert. In Tabelle 4.14 werden vier solcher Schlüsselteile dargestellt. 08 52 6F 6F 74 50 4F 41 00<sup>19</sup> steht für acht Zeichen, dem „RootPOA“ plus Nullbyte und wurde

<sup>19</sup>Siehe Tabelle 4.12

Tabelle 4.13: Transiente SUN Objektschlüssel verschiedener POA

| POA | Objekt | Schlüsselabschnitt            |
|-----|--------|-------------------------------|
| 1   | 0      | 08 00 00 00 01 00 00 00 00 14 |
| 2   | 0      | 08 00 00 00 02 00 00 00 00 14 |
| 3   | 0      | 08 00 00 00 03 00 00 00 00 14 |
| 1   | 1      | 08 00 00 00 01 00 00 00 01 14 |
| 2   | 1      | 08 00 00 00 02 00 00 00 01 14 |
| 3   | 1      | 08 00 00 00 03 00 00 00 01 14 |
| 1   | 16     | 08 00 00 00 01 00 00 00 10 14 |
| 2   | 16     | 08 00 00 00 02 00 00 00 10 14 |
| 3   | 16     | 08 00 00 00 03 00 00 00 10 14 |

Tabelle 4.14: SUN POA-Namen, Kodierung

| Schlüsselabschnitt                              | POA-Name   |
|---|------------|
| 00 00 00 05 6E 75 6C 6C 00 00 00 00             | null       |
| 00 00 00 07 6F 6E 65 6F 6E 65 00 00             | oneone     |
| 00 00 00 0A 74 77 6F 74 77 6F 74 77 6F 00 00 00 | twotwotwo  |
| 00 00 00 0B 74 68 72 65 65 74 68 72 65 65 00 00 | threethree |

in der Tabelle weggelassen. Jeweils angehängte „00“ dienen dem *CDR-Alignment* und *Padding* [OMG04]. Betrachtet man nun wieder den Start des Schlüssels, so sind für transiente Objekte die ersten acht Byte immer AF AB CB 00 00 00 00 20. Betrachtet man persistente Objektschlüssel oder blickt man in den Quelltext, so kann man feststellen, dass persistente Schlüssel immer mit AF AB CB 00 00 00 00 22 beginnen. Byte vier bis acht spiegelt also die *LifespanPolicy* des POAs wieder. Nur mit Hilfe des Quelltextes<sup>20</sup> ist dieser Verdacht zu bestätigen. Die nachfolgenden vier Bytes (Bsp.: 59 79 A9 BA) variieren von ORB zu ORB, wachsen stetig und beginnen bei Überlauf wieder von vorn. Dass es sich hier um einen Zeitstempel handelt liegt auf der Hand. Zeitexperimente belegen, dass sich der Wert pro Sekunde um 1000 vermehrt. Es ist also anzunehmen, dass es sich um einen Millisekundenzähler handelt, wie ihn `System.currentTimeMillis()`

<sup>20</sup>In Datei `com/sun/corba/se/impl/orbutil/ORBConstants.java`, Zeile 87f.

bereitstellt. In der ORB-Implementierung findet sich auch genau dieser Punkt. Durch eine Typumwandlung werden nur die letzten 32 Bit betrachtet, dadurch rotiert dieser Wert alle 49,7 Tage.



Abbildung 4.6: Aufbau von SUN-ORB Objektschlüsseln

Für persistente Objekte muss keine gesonderte Betrachtung stattfinden, da nur die ORB-ID anders belegt wird. Durch Variation der für persistente POA typischen Einstellungen, wie Server-Id und Server-Port, gelangt man zu dem Ergebnis, dass dort die `PersistentServerId`<sup>21</sup> statt der Zeitangabe zu finden ist.

Um nun einen SUN JAVA ORB object key ohne Vorwissen erraten zu können, ist es notwendig, dass man bei transienten Objekten den ORB-Zeitstempel (ORB-ID) und die gesamte POA-Hierarchie errät. Der Zeitstempel kann  $256^4 = 4,3 \times 10^9$  verschiedene Werte annehmen. Die POA-Hierarchie ist theoretisch unendlich komplex, kann aber durch bereits in vorherigen Abschnitten getroffene Annahmen auf  $1,28 \times 10^6$  Möglichkeiten begrenzt werden. Da für persistente Objekte die ORB-ID (als `PersistentServerId`) die gleiche Komplexität aufweist, treffen diese Annahmen gleichermaßen zu.

#### Transiente und persistente Objekte ( $t_{1,2}$ ):

$$4,3 \times 10^9 \times 1,28 \times 10^6 \times 10 = 5,5 \times 10^{16} \equiv 1,7 \times 10^6 \text{a}$$

Dieses Ergebnis verspricht zunächst ein hohes Maß an Sicherheit. Betrachtet man jedoch die Wahrscheinlichkeiten, wenn bereits eine gültige IOR bekannt ist, so verringern sich diese Zahlen drastisch. Ist also ein „Informati-  
onsleck“ aufgetreten oder sind ohnehin IORs bekannt, dann besteht akuter

<sup>21</sup>Sie bezeichnet den Servernamen oder eine Servernummer, die den ORB später bei einem Neustart wieder korrekt identifiziert. Dieser Mechanismus ist für den persistenten Objektaufwurf wichtig.

Handlungsbedarf. Nicht vergessen werden darf jedoch, dass bezüglich der Wahrscheinlichkeit für Objekte anderer POAs auch hier bestimmte Annahmen vorausgesetzt sind. Die praktische Sicherheit kann deutlich über oder unter dem angegebenen Werten liegen.

**Objekte am gleichen POA ( $t_3$ ):**

$10 \equiv 10\text{ms}$

**Objekte an anderem POA ( $t_4, t_5$ ):**

$1,28 \times 10^6 \equiv 21\text{min}20\text{s}$

## 4.5 Visibroker

Objektschlüssel des Borland Visibroker weisen generell die gleichen Merkmale wie alle zuvor behandelten Schlüssel auf. Somit erscheint es zweckmäßig, den Aufbau der Objektschlüssel nicht so detailliert darzustellen, wie bisher. Es wird nur noch auf Eigenheiten eingegangen und der Schlüsselauflaufbau graphisch dargestellt. Die EBNF findet sich, wie für alle anderen Schlüssel auch, im Anhang. Da es sich nicht um ein OpenSource Produkt handelt und sich transiente von persistenten Schlüsseln kaum unterscheiden, wird keine Quelltextanalyse und auch keine textuelle Trennung der Schlüsseltypen vorgenommen.

Tabelle 4.15 zeigt einen transienten Visibroker 7 Objektschlüssel, der ein an POA „null“ gebundenes Objekt referenziert, Tabelle 4.16 das entsprechende persistente Pedant. Ein Test mit Visibroker 6.5 ergab, dass diese Schlüssel identisch aufgebaut sind. Innerhalb aller Analysen konnte kein Unterschied zwischen beiden Versionen festgestellt werden.<sup>22</sup> Bei den ersten vier Byte handelt es sich wieder um ein magic, das jedoch auch die LifespanPolicy des POAs anzeigt. Die bereits bekannte POA-Hierarchie ist ebenfalls beim Visibroker im Objektschlüssel wiederzufinden. Der Vergleich mehrerer Objekte eines POAs zeigt außerdem, dass vom Objektzähler ebenfalls Gebrauch gemacht wird. Die einzige „Spezia-

---

<sup>22</sup>Bezogen auf den Schlüsselaufbau.

Tabelle 4.15: Transienter Visibroker Objektschlüssel

| HEX                                    | URL           |
|--|---------------|
| 00 56 42 01 00 00 00 06 2F 6E 75 6C 6C | .VB...../null |
| 00 20 20 00 00 00 04 00 00 00 00 00 00 | .....         |
| 02 18 34 34 ED 39                      | .....9        |

Tabelle 4.16: Persistenter Visibroker Objektschlüssel

| HEX                                    | URL        |
|--|------------|
| 00 50 4D 43 00 00 00 04 00 00 00 06 2F | .PMC...../ |
| 6E 75 6C 6C 00 20 20 00 00 00 04 00 00 | null.....  |
| 00 00                                  | ..         |

lität“ ist eine Art Zeitstempel bei transienten Objekten. Die letzten acht Bytes (zwei Variablen vom Typ long oder ein long long) des Schlüssels beinhalten diesen Stempel. Versucht man jedoch den Timestamp 00 00 02 18 34 34 ED 39<sup>23</sup> umzurechnen, so stößt man auf eine scheinbar zu große Zahl.

$$0x000002183434ED39 = 2'302'978'354'489$$

Diese Zahl entspricht keinesfalls den Sekunden oder Millisekunden eines UNIX-Zeitstempels. Betrachtet man dies jedoch über einen langen Zeitraum, so kommt man zu der Annahme, dass es sich um zwei long-Werte handelt, die miteinander die korrekte Zeit ergeben könnten. In Tabelle 4.17 sind zwei Stempel, deren Erstellungszeit und die dazugehörige UNIX-Zeit in Millisekunden<sup>24</sup> aufgeführt. Berechnet man die Differenz der Stempel, so müssten diese ca. 102 Tage auseinander liegen, in Wirklichkeit sind es aber nur 84 Tage. Spaltet man nun die acht Bytes in zwei long Werte auf und nimmt an, dass 0x02 15 für das Ergebnis relevanter ist<sup>25</sup>, also Vielfache von 2<sup>32</sup> darstellt, so macht man die Beobachtung, dass dieser Wert

<sup>23</sup>aus Tabelle 4.15

<sup>24</sup>Der Bereich der Sekunden und Millisekunden wurde mit Nullen gefüllt, da die Erstellungszeit nur minutengenau vorliegt.

<sup>25</sup>Aus engl.: *most significant*.

Tabelle 4.17: Visibroker Zeitstempel

| Stempel                 | Erstellungszeit  | currentTimeMillis() |
|-------------------------|------------------|---------------------|
| 00 00 02 15 4C 80 65 DC | 24.04.2006 17:24 | 1'145'892'230'000   |
| 00 00 02 18 34 33 7A A4 | 03.07.2006 13:43 | 1'141'735'400'000   |

( $2^{32} = 2'289'217'568'768$ ) nahezu das Doppelte des Wertes ist, der die eigentliche UNIX-Zeit darstellt. Halbiert man diesen, multipliziert also mit  $2^{31}$  statt  $2^{32}$  und addiert die restlichen Bytes des zweiten long hinzu, so erhält man<sup>26</sup>

$$533 \times 2^{31} + 1'283'483'100 = 1'145'892'267'484.$$

Dies ist der bis auf 37s genaue, anhand der bekannten Erstellungszeit erwartete Zeitstempel. Der Stempel  $s = l_1 l_2$  folgt also der Formel

$$l_1 = t \bmod 0x7FFFFFFF \quad (4.1)$$

$$l_2 = t \div 0x7FFFFFFF \quad (4.2)$$

Verfährt man beim zweiten Stempel aus Tabelle 4.17 analog, so kommt man zu einem identischen Ergebnis. Ein transienter Schlüssel ist wie folgt aufgebaut:



Abbildung 4.7: Aufbau transienter Visibroker-Objektschlüssel

Bei persistenten Objekten wird auf diesen Zeitstempel ersatzlos verzichtet. Ein Einfluß der ServerId auf die Schlüssel konnte ebenfalls nicht festgestellt werden. Der persistente Schlüssel folgt folgender Erzeugungsvorschrift:

<sup>26</sup>Die Hexadezimalwerte wurden in das Dezimalsystem überführt.



Abbildung 4.8: Aufbau persistenter Visibroker-Objektschlüssel

Unter der Maßgabe, dass die Erstellungszeit auf einen Tag genau vorhergesagt werden kann und sonst nichts bekannt ist, beträgt die Anzahl der Möglichkeiten für Visibroker Objektschlüssel:

**Transiente Objekte ( $t_1$ ):**

$$86 \times 10^6 \times 1,28 \times 10^6 \times 10 = 1,1 \times 10^{15} \equiv 35 \times 10^3 \text{a}$$

**Persistente Objekte ( $t_2$ ):**

$$1,28 \times 10^6 \text{ times } 10 = 1,28 \times 10^7 = 3,6 \text{h}$$

Können Aussagen über die POA-Hierarchie bezüglich des Aufbaus bzw. der Namen getroffen werden, wird der Wertebereich deutlich verkleinert und man benötigt erheblich weniger Zeit. Es erfolgt eine Näherung an die Komplexität der Objektschlüssel, wenn bereits eine Interoperable Objektreferenz bekannt ist und weitere Annahmen wie bei den anderen Untersuchungen<sup>27</sup> getroffen werden:

**Objekte am gleichen POA ( $t_3$ ):**

$$10 \equiv 10 \text{ms}$$

**Objekte an anderen POAs ( $t_{4,5}$ ):**

$$1,28 \times 10^6 \equiv 21 \text{min}$$

Auch hier zeigt sich, dass wenn eine IOR bekannt ist, ein potentielles Sicherheitsproblem besteht.

## 4.6 Weblogic

Wie bereits beim Visibroker, werden an dieser Stelle zur Redundanzvermeidung nur noch die Endergebnisse vorgestellt. In 3.1.6 wurde dargelegt,

<sup>27</sup>Ca. 10% der Objekte eines POAs sind aktiv, maximal  $10^5$  Objekte sind vorhanden und die maximale Komplexität der POA-Hierarchie beträgt  $1,28 \times 10^6$ .

Tabelle 4.18: Transienter Weblogic Objektschlüssel

| Zeile | HEX                                 | URL          |
|-------|-------------------------------------|--------------|
| 1     | 00 42 45 41 08 01 03 00 00 00 00 01 | .BEA.....    |
| 2     | 00 00 00 00 00 00 00 00 00 00 00 17 | .....        |
| 3     | 49 44 4C 3A 48 65 6C 6C 6F 41 70 70 | IDL:HelloApp |
| 4     | 2F 48 65 6C 6C 6F 3A 31 2E 30 00 00 | /Hello:1.0.. |
| 5     | 00 00 00 04 32 35 39 00 00 00 00 01 | ....259..... |
| 6     | 42 45 41 11 00 00 00 36 00 00 00 00 | BEA....6.... |
| 7     | 00 00 00 00 00 00 00 00 73 B2 F0 BD | .....s...    |
| 8     | 7F FF FF 02 00 00 00 18 52 4D 49 3A | .....RMI:    |
| 9     | 5B 42 3A 30 30 30 30 30 30 30 30 30 | B:00000000   |
| 10    | 30 30 30 30 30 30 30 00 00 00 00 01 | 000000.....  |
| 11    | 31                                  | 1            |

dass Probleme bei der Erstellung persistenter Objekte und der Portzuweisung der Servants eine umfangreiche Analyse verhinderten und hier nur die transienten Objektschlüssel analysiert werden. Tabelle 4.18 zeigt den Schlüssel des zehnten Objektes eines POAs unterhalb des „RootPOA“. Auffällig ist, dass die RepositoryId (TypeId) im Schlüssel vorkommt. Bekannt und fast schon erwartet hat man ein ORB-magic, hier BEA. Eine Besonderheit, die sich ohne Quelltext nicht klären lässt, sind die letzten acht Bytes der siebten Zeile, hier 00 00 00 00 73 B2 F0 BD. Die ersten vier Bytes sind entweder alle 00 oder FF während die zweite Hälfte anscheinend mit Zufallswerte gefüllt wird. Eine Korrelation mit der Zeit konnte nicht festgestellt werden. Da diese Bytefolge „nur“ pro ORB einmalig ist, erhöht dies leicht die Sicherheit. Weshalb die RepositoryId in den Schlüssel integriert wurde, kann nicht gesagt, historische Gründe dürfen aber vermutet werden. Erst in CORBA 2.1 wurde mit IIOP ein Protokoll eingeführt, in dem der Objekttyp zu finden ist. In Zeile fünf ist im Klartext ein POA-Zähler mit dem Offset 258 zu finden. Am Ende der vorletzten Zeile steht die Länge des Objektzählers (es wird ORB-weit hochgezählt) dem der Objektzähler selbst in ASCII folgt. Gut zu erkennen ist dies in Tabelle 4.19.

Tabelle 4.19: Weblogic Objektzähler

| lfd. Objekt | Länge       | Schlüsselteil |
|-------------|-------------|---------------|
| 1           | 00 00 00 01 | 31            |
| 2           | 00 00 00 01 | 32            |
| 9           | 00 00 00 01 | 39            |
| 10          | 00 00 00 02 | 31 30         |
| 99          | 00 00 00 02 | 39 39         |
| 100         | 00 00 00 03 | 31 30 30      |



Abbildung 4.9: Aufbau transienter Weblogic-Objektschlüssel

Weblogic object keys sind die mit Abstand kompliziertesten. Nicht allein die Grammatik, auch die Erzeugungsvorschriften sind komplex und zeigen die Verschachtelung des Schlüssels in sich. Sollte beim Design ein Ziel gewesen sein, eine Analyse zu verhindern, so ist dies, zumindest teilweise, gelungen. Die Erzeugung der ORB-ID konnte mit legalen Mitteln<sup>28</sup> nicht nachvollzogen werden. Trotzdem ist das Erraten eines gültigen Schlüssels kein unmögliches Unterfangen. Die Anzahl der Möglichkeiten verdeutlicht dies:

**Transiente Objekte ( $t_1$ ):**

$$256^4 \times 2 \times 50 \times 10 = 4,3 \times 10^{12} \equiv 136a$$

Diese Zahl ergibt sich aus den Annahmen, dass maximal 50 POA existieren, 10 Prozent aller Objekte die getestet werden, aktiv sind und die ORB-ID zufällig ist. Nimmt man aber an, dass bereits eine IOR bekannt ist, reduziert sich diese Zahl dramatisch:

**Objekte am gleichen POA ( $t_3$ ):**

$$10 \equiv 10ms$$

<sup>28</sup>Zu den illegalen Mitteln gehört reverse engineering bzw. Dekompilieren wie es beispielsweise mit JAD möglich ist.

**Objekte an anderem POA ( $t_4$ ):**50  $\equiv$  50ms

Beim Ausprobieren der einzelnen POA wird davon ausgegangen, dass man bereits beim ersten Zugriffsversuch feststellen kann, ob dieser POA existiert. Die gesamte Anzahl möglicher Objekte wird also nur bei „lebenden“ POA geprüft und käme noch hinzu, ist aber meist nicht signifikant<sup>29</sup>.

**4.7 WebSphere**

WebSphere Objektschlüssel unterscheiden Persistenz und Transienz nur anhand einer Variablen, deshalb treffen die folgenden Aussagen auf beiderlei Schlüsselarten zu. Sie verfügen über ein magic, eine einem Zeitstempel ähnliche ORB-ID, die POA-Hierarchie und einen Objektzähler. Tabelle 4.20 zeigt einen transienten Schlüssel am POA „null“. In Zeile eins sieht

Tabelle 4.20: Transienter WebSphere Objektschlüssel

| HEX                                 | URL           |
|-------------------------------------|---------------|
| 4C 4D 42 49 00 00 00 16 C6 D6 EB D8 | LMBI.....     |
| 00 15 00 05 04 6E 75 6C 6C          | ....null..... |
| 00 04 00 00 00 00                   | .....         |

man die ersten vier Bytes 4C 4D 42 49 - diese sind für WebSphere Objektschlüssel spezifisch und stellen ein magic dar. Die nachfolgenden vier Bytes sind die einzigen, wodurch sich grundsätzlich persistente und transiente Objekte unterscheiden lassen. Byte acht ist für persistente Objekte 16 und für transiente 15. Anschließend folgt die vier Byte lange ORB-ID, die zufällig erscheint. Die folgenden vier Bytes sind jeweils zwei short a zwei Byte. Der Erste gibt die absolute Startposition des Objektzählers an und der Zweite die relative, gemessen ab der Position hinter dieser Angabe, also einschließlich ab Byte 17. Die Differenz beider Angaben ist immer

<sup>29</sup>In diesem Fall doch, da sich die Zeit um 20% erhöht.

0x10. Die zweite Angabe kann man auch als die Gesamtlänge der POA-Hierarchie ansehen, diese folgt nämlich nach folgendem Schema: Länge des POA-Namen (in einem Byte) und der POA-Name selbst, als ASCII Zeichenkette. Beendet wird der Schlüssel mit der Länge des Objektzählers (zwei Byte) und dem Objektzähler selbst (vier Byte).



Abbildung 4.10: Aufbau transienter WebSphere-Objektschlüssel

Der WebSphere ORB nutzt einen Großteil der Mechanismen, die ebenfalls von anderen ORB benutzt werden. Da diese bereits vorgestellt wurden, wird an dieser Stelle nicht näher darauf eingegangen. Die Komplexität des Schlüssels ergibt sich in der Hauptsache aus einer zufälligen ORB-ID und der POA-Hierarchie. Die Anzahl verschiedener Schlüssel ist für:

**Transiente und persistente Objekte ( $t_{1,2}$ ):**

$$256^4 \times 1,28 \times 10^6 \times 10 = 5,5 \text{ times } 10^{15} \equiv 174 \times 10^5 \text{a}$$

Die bereits in vorherigen Abschnitten getroffenen Annahmen, 50 verschiedene POA-Namen in drei Hierarchieebenen und 10 Prozent aktive Objekte, wurden hierbei berücksichtigt. Ist eine Objektreferenz bekannt, muss nur noch wenig herausgefunden werden:

**Objekte am gleichen POA ( $t_3$ ):**

$$10 \equiv 10 \text{ms}$$

**Objekte an anderem POA ( $t_{4,5}$ ):**

$$1,28 \times 10^6 \equiv 21 \text{m}20 \text{s}$$

Abschließend kann man sagen, dass auch bei diesem ORB keine echte Sicherheit vorliegt und er sich folglich in die Masse der anderen ORB einreicht.

## 4.8 Komplexität der Objektschlüssel

Die Zahlen in Tabelle 4.21 entsprechen der Anzahl von Möglichkeiten für Objektschlüssel. Die Einzelwerte sind dem Fazit der einzelnen ORB entnommen und hier zusammengefasst. Zur besseren Übersicht und Vergleichbarkeit bieten sich Komplexitätsklassen an – Je höher die Klasse, desto höher die Komplexität. Die Klasse spiegelt die Zehnerpotenz wider, die die maximale Anzahl der Möglichkeiten reflektiert. So gehören  $5,6 \times 10^9$  Möglichkeiten in die Klasse 10, da 5,6 aufgerundet und folglich aus  $10^9$   $10^{10}$  wird. Klasse 1 bedeutet, dass die Anzahl der Möglichkeiten höchstens  $10^1 = 10$  ist. Aus den untersuchten Varianten resultierend, ergeben sich für jeden ORB maximal fünf Ergebnisse. Dies resultiert aus den untersuchten Varianten.

Tabelle 4.21: Vergleich der Objektschlüsselkomplexität

| ORB        | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ | $\bar{t}$ |
|------------|-------|-------|-------|-------|-------|-----------|
| MICO       | 10    | 16    | 1     | 3     | 6     | 7,2       |
| ORBacus    | 17    | 12    | 1     | 11    | 11    | 10,4      |
| Orbix      | 30    | –     | 1     | 30    | –     | 20,3      |
| SUN ORB    | 17    | 17    | 1     | 6     | 6     | 9,4       |
| Visibroker | 15    | 7     | 1     | 6     | 6     | 7,0       |
| Weblogic   | 12    | –     | 1     | 2     | –     | 3,0       |
| WebSphere  | 16    | 16    | 1     | 6     | 6     | 9,0       |

$t_1$  Möglichkeiten für transiente Objektschlüssel unter der Annahme, dass nichts bekannt ist. Ist jedoch der POA-Name oder die POA-Hierarchie Teil des Schlüssels, so wurde angenommen, dass es maximal 50 POA-Namen gibt die bekannt und in maximal drei Ebenen miteinander verbunden sind. Ferner ist angenommen, dass 10 Prozent der Objekte am POA aktiv und diese zusätzlich über den Schlüsselraum gleichverteilt sind.

$t_2$  Wie  $t_1$ , nur für persistente Objekte.

$t_3$  Verschiedene Möglichkeiten für Objekte am gleichen POA. 10 Prozent der Objekte am POA sind aktiv.

$t_4$  Hier ist eine IOR bekannt und es zählt die Anzahl der Möglichkeiten für andere transiente Objekte an anderen POA.

$t_5$  Wie  $t_4$ , nur für persistente Objekte.

$\bar{t}$  Der Durchschnittswert der Komplexität soll hier in einer Zahl zusammengefasst werden um einen Vergleich zu gewährleisten. Diesem Anspruch kann dieser Durchschnittswert nur begrenzt gerecht werden, da einerseits Annahmen zu diesen Ergebnissen führten und andererseits nicht bei allen ORB alle Werte verfügbar waren.

Um den Komplexitätsklassen maximal benötigte Zeiten für das Erraten zuordnen zu können, ist diese Beziehung in Tabelle 4.22 dargestellt. Es wird gezeigt, dass ab Klasse neun ein Erraten sinnlos wird. Mit Klasse 20 wird das Erdalter von 4,6 Milliarden Jahren überschritten.

Tabelle 4.22: Umrechnung Komplexität in Zeitaufwand

| Komplexitätsklasse | Zeitäquivalent                  |
|--------------------|---------------------------------|
| 1                  | 50ms                            |
| 2                  | 500ms                           |
| 3                  | 5s                              |
| 4                  | 50s                             |
| 5                  | 8min                            |
| 6                  | 80min                           |
| 7                  | 13h                             |
| 8                  | 6d                              |
| 9                  | 58d                             |
| 10                 | 578d                            |
| 11                 | 16a                             |
| 12                 | 158a                            |
| 13                 | 1578a                           |
| 15                 | 158 Jahrtausende                |
| 20                 | 3,4 Erdenleben                  |
| 30                 | $3,4 \times 10^{10}$ Erdenleben |

# 5

## Diskussion

In diesem Kapitel wird erörtert, ob die in der Einleitung getroffene Vermutung des illegalen Zugriffs zutreffend ist. Es werden die Komplexität der Objektschlüssel diskutiert und Lösungen angeboten, die die dargelegten Schwachstellen beseitigen können.

### 5.1 Komplexität

Die im Ergebnisteil getroffenen Annahmen bezüglich der Komplexität der POA-Hierarchien haben auf viele Wahrscheinlichkeiten einen sehr großen Einfluss. Gelingt es, diese Variable deutlich zu verringern, so würden sich die benötigten Zeiten verringern. Die POA-Komplexität und die benötigte Zeit für das Erraten sind linear abhängig. Weiterhin ist diese Komplexität nur angenommen, Erfahrungen aus der Praxis konnten nicht in die Arbeit integriert werden.

Weiterhin vorhandene Zeitstempel innerhalb der Schlüssel können, je nach praktischer Ausprägung eines Systems, mehr oder weniger gut vorhergesagt werden. Ist beispielsweise bekannt, dass ein ORB am 1.1.2006 1:00 Uhr neu gestartet wurde und dabei alle Dienste neu initialisiert hat, so ist sicher, dass diese Zeitstempel alle auf ein Datum kurz nach dem 1.1.2006 1:00 Uhr zeigen. Die Komplexität reduziert sich, die Wahrscheinlichkeit des Erratens erhöht sich. Werden keinerlei Zeitstempel oder stringifizierte Daten über den ORB, POA und das Objekt in den Objektschlüssel eingearbeitet und stattdessen auf andere als die vorgestellten Mechanismen zurückgegriffen, ist ebenfalls viel gewonnen.

In Tabelle 4.21 sieht man, dass die Komplexität der Schlüssel stark variiert. Dies verwundert insofern, da die meisten ORB die gleichen Mechanismen zur Schlüsselerzeugung einsetzen. Orbix geht mit dem Einsatz eines Hashes für die POA-Identifikation einen eigenen Weg. Dieser ist im Vergleich zu den anderen ORB auch sicherer, da die Anzahl der notwendigen Versuche um einige Zehnerpotenzen höher liegt. Leider wurde versäumt, diesen Mechanismus auch für die Objekte einzusetzen. Der hier immer eingesetzte Zähler bietet keine Sicherheit und lädt zum Ausprobieren anderer Objekte förmlich ein.

Da ein Designziel der meisten ORB die Ausführungs- oder Verarbeitungsgeschwindigkeit ist, liegt es natürlich nahe, einfache Methoden zur Schlüsselerzeugung zu nutzen. Wichtig ist sicher auch, diese Schlüssel wieder schnell einem Objekt zuordnen zu können. Gleiches trifft auch auf die POA-Hierarchie und den POA zu. Wird der Weg zum zuständigen POA gleich mit im Schlüssel abgelegt, muss kein aufwändiges Listenmanagement innerhalb des ORB für die Zuordnung sorgen.

Es wurden also Designziele berücksichtigt, die Sicherheitskonzepte entweder völlig ausblendeten oder aber bewusst auf diese verzichteten.

## 5.2 Machbarkeitsstudie

Die für die Objektschlüssel ermittelten Komplexitäten lassen vermuten, dass man ohne Vorwissen keinen, mit Vorwissen jedoch schnell und einfach einen Zugriff auf ein Objekt erhalten kann. Am Beispiel von MICO wird versucht, eine Angriffssoftware zu schreiben, die es ermöglicht, Objektreferenzen zu generieren und zu testen.

Da für das Auffinden weiterer Objekte am selben POA durchweg die geringste Komplexität vorliegt, soll dies zuerst versucht werden. Anschließend sollen Objekte an anderen POA gefunden und überprüft werden. Für dieses Szenario wird das Tool zur Erzeugung der Objektreferenzen derart modifiziert, dass der ORB in den Zustand „run“ geschaltet wird und mindestens ein weiteres Objekt am selben und eines an einem weiteren POA aktiv sind. Die IOR eines gültigen aktiven Objektes sei gegeben.

Auf der beiliegenden CD befindet sich das Programm „iGuess“. Es ist ein JAVA Programm, dass MICO-Objektschlüssel erzeugen und ausprobieren kann. Wird ein aktives Objekt gefunden, so kann man den Objekttyp ebenfalls testen. Die Anzahl der zu untersuchenden POA und Objekte lässt sich übergeben. Ein typischer Aufruf mit einer bekannten IOR und die Programmausgabe sieht wie folgt aus:

```
$ ./start -mc IOR:010000000e0000004...0000 -o 1000 -p 1000 \
-t IDL:Hello:1.0
*****
* I Guess - Object Key Guessing Tool *
* Version 0.4.3 - (C) christoph.becker@prismtech.com *
*****

found active Object (Obj# 10 POA# 0)
corbaloc:iiop:1.0@localhost.localdomain:52501//6323/1158735167/0/_01
Type: IDL:Hello:1.0

*****
Statistic
=====
```

```
Objects to be tested: 1000000
active/tested Objects: 1/1999
active/tested POAs: 1/1000
-----
Time total:          6243ms
Time per Object     3ms
-----
known Types:        1
Object not exists: 999
Bad Op Exceptions: 0
Object Adapter E.: 999
Comm Failure Ex. : 0
System Exceptions: 0
Other Exceptions : 0
*****
```

Man kann hier deutlich erkennen, dass ein Objekt gefunden werden konnte. Die TypeID entspricht auch dem gesuchten „IDL:Hello:1.0“. An dieser Stelle könnte man nun mittels eines weiteren Programmes mit dem gefundenen Objekt interagieren.

Die Aufwandsabschätzung mit der im Verlauf dieser Arbeit gerechnet wurde, hat sich für große Testszenarien (mehr als 10'000 Objekte) bestätigt. Die Testzeit von 1ms pro Schlüsseltest ist realistisch.

Ob und inwiefern sich Verbesserungen durch Parallelisierung erreichen lassen, wurde nicht geprüft. Jedoch ist zu vermuten, dass der ORB, der das aktive Objekt verwaltet, an seine Leistungsgrenzen stoßen wird. Aufgrund der Last, die ein Angriff erzeugt, sind ORB auf schnellen (und damit teureren) Rechnern eher angreifbar als ORB auf alten leistungsschwachen Rechnern. Die verfügbare Bandbreite spielt sicher auch eine Rolle, wurde aber vernachlässigt, da von guter Vernetzung ausgegangen werden kann. Jede Firma, die verteilte Dienste anbietet oder selbst in Anspruch nimmt, wird sicherstellen, dass diese auch genutzt werden können.

In Zukunft kann „iGuess“ so erweitert werden, dass Angriffe auf Schlüssel parallel und auch für andere ORB durchgeführt werden können. Ebenfalls denkbar ist die direkte Kopplung mit einer bisher nicht vorhandenen „Schadfunktion“.

## 5.3 Lösungsmöglichkeiten

Die Machbarkeitsstudie zeigt, dass die angenommenen Sicherheitslücken praktische Relevanz besitzen. Doch wie kann man diese Lücken schließen oder die Sicherheit anderweitig erhöhen? Muss dafür der ORB geändert werden?

### 5.3.1 Zeitstempel - Bereichserweiterung

Eine Lösung, die Veränderungen am ORB vornimmt, verbessert den Umgang mit Zeitstempeln. Diese werden meist dazu eingesetzt, um den ORB eindeutig identifizieren zu können. Kein Stempel existiert zwei Mal. Zwei ORB mit selbem Stempel wurden zwar zur selben Zeit erzeugt, besitzen aber trotzdem unterschiedliche Merkmale wie Host oder Port.

Setzt man nun den Zeitstempel im Sekunden oder Millisekundenbereich auf den Nanosekundenbereich um und integriert diese Werte vollständig in den Schlüssel, dann steigt die Komplexität, sowohl für den ORB, als auch für den POA und gegebenenfalls pro Objekt beachtlich. Sie betrüge

$$(8,6 \times 10^{10})^3 = 6,4 \times 10^{32}.$$

Nimmt man nun an, der Zeitrahmen könnte auf fünf Minuten reduziert werden, beträgt der Wert immernoch:

$$(3 \times 10^8)^3 = 2,7 \times 10^{25}$$

Wird eine gültige Objektreferenz bekannt, so hätte man für ein anderes Objekt desselben POA noch  $3 \times 10^8$  Möglichkeiten allein für den Zeitstempel wenn man wieder annimmt, dieses Objekt sei innerhalb von fünf Minuten nach dem anderen Objekt erzeugt worden. Für die Komplexitätsklasse acht kann ein maximaler Zeitraum von sechs Tagen angenommen werden. Ob das Objekt dann noch existiert?

Um das Bekanntwerden eines Zeitstempels im Klartext zu vermeiden und somit die Rückgewinnung von Informationen, wie Startzeit zu verhindern, ist der Einsatz von Einwegfunktionen zweckmäßig. Einerseits wird

der zur Verfügung stehende Wertebereich besser ausgenutzt, andererseits wird der besagte Stempel geschützt. Das einzige Problem, dass auch nach dieser Modifikation besteht, ist die implizite Autorisierung durch den Erhalt einer gültigen IOR. Allerdings bestünde der Zugriff dann nur noch auf dieses Objekt. Alle anderen Objekte wären durch die hohe Komplexität geschützt.

### 5.3.2 ORB-basiertes Signieren

Wenn Objektreferenzen durch den ORB (bzw. den POA) signiert werden würden, so ist das Erzeugen anderer Schlüssel für Außenstehende zwar möglich, jedoch ist immer die Signatur falsch und der Zugriff kann abgewiesen werden. Der Zugriff auf das Objekt anhand einer abgefangenen IOR ist aber noch möglich. Um zu verhindern, dass der Aufruf vom Client zum Server nicht manipuliert wurde, kann eine Verschlüsselung oder Signierung des Datenverkehrs in beide Richtungen helfen. Somit ist zwar das Abhören und Wiederholen der Nachrichten möglich, eine Datenveränderung würde aber erkannt werden.

Um Replay-Attacken zu vereiteln, muss man über die Objektreferenzen hinaus Änderungen am ORB vornehmen. Man kann eine Transaktionsnummer einführen, die in die Objektreferenz einfließt und ebenfalls signiert wird. Wenn diese Transaktionsnummern fortlaufend sind, so darf der Empfänger keine Nachrichten verarbeiten, die eine bereits benutzte Transaktionsnummer besitzen.

### 5.3.3 Verschlüsselung

Auch bei CORBA ist Verschlüsselung eine gute Idee, besonders wenn die Informationen über einen ungesicherten Kanal wie das Internet geschickt werden müssen. Einige OpenSource ORB unterstützen SSL, kommerzielle Produkte sind ebenfalls verfügbar. Da beiden Kommunikationspartnern das Zertifikat des Anderen bekannt ist, kann man sogenannte Man-in-the-middle Angriffe erkennen und anschließend wieder eine sichere Lei-

tung herstellen. Diese Lösung geht davon aus, dass beide Partner jederzeit vertrauenswürdig sind. Ist einer der an der Kommunikation Beteiligten nicht vertrauenswürdig, so ist es zwar die Verbindung, aber an Sicherheit wurde nicht viel gewonnen. Nicht vertrauenswürdige Partner wären Geschäftspartner/-kunden die über eine Schnittstelle an die eigene Verwaltungssoftware angeschlossen sind und dadurch beispielsweise Nachbestellungen erhalten oder Ähnliches.

### 5.3.4 IIOP-Proxy, IIOP-Firewall

Eine Variante zur Erhöhung der Sicherheit, ohne Veränderungen am ORB vornehmen zu müssen, sind Proxies/Firewalls, die speziell für CORBA entwickelt wurden. In diesen Bereich fallen auch Security Gateways. Dies sind Programme, die den Verkehr von und zu CORBA-Systemen überwachen und gegebenenfalls unterbinden können. Durch Auslesen und Verändern der Felder im IIOP-Protokollkopf ist es diesen Programmen möglich, den Zugriff auf Objekte in einfacher Art zu kontrollieren und zu manipulieren. Je nach Produkt können sogar einzelne Aufrufparameter begrenzt werden. Bekannte Vertreter dieser Softwareklasse sind der *Xtradyne Domain Boundary Controller* (I-DBC) und IONAs *Wonderwall*. Diese Programme sind im Prinzip beides IIOP-Proxies bzw. Security Application Gateways. Einen etwas älteren Vergleich beider Produkte findet man bei [Jet01].

# 6

## Zusammenfassung

Die Ergebnisse der Arbeit zeigen, dass Objektreferenzen keinesfalls als Capabilities angesehen werden dürfen.

Es wurde theoretisch und praktisch nachgewiesen, dass es in wenigen Sekunden bis Stunden möglich ist, gültige Objektreferenzen zu erzeugen und diese zu nutzen. Alle sicherheitsrelevanten Implikationen auf Basis dieser Referenzen führen zu Problemen. Die scheinbare Unvorhersagbarkeit der herstellerabhängigen Schlüssel vermittelt ein Gefühl der Sicherheit. Dieses trügt.

In der Zukunft muss versucht werden, Lösungen, die angeboten wurden, umzusetzen oder neue, möglicherweise bessere Lösungen zu finden. Systemimmanent ist und bleibt ein gewisses Restrisiko. Doch durch den gezielten und vor allem bewussten Einsatz von Schutzprogrammen oder Sicherungsmechanismen, wird einerseits eine gesicherte Umgebung geschaffen und andererseits die Sensibilität für Sicherheitsprobleme erhöht.

Eine vollkommene Sicherheit gibt es nicht. Sie kann deshalb auch nicht erreicht werden, man kann sich ihr aber nähern.

# A

## Anhang

### A.1 Quelltextauszüge

#### A.1.1 IDL - IOP Definitionen

Listing A.1: IOP Modul - Profilcontainer

```
module IOP { // IDL
  typedef unsigned long ProfileId;

  struct TaggedProfile {
    ProfileId tag;
    sequence <octet> profile_data;
  };

  typedef sequence <TaggedProfile> TaggedProfileSeq ;

  // an Interoperable Object Reference is a sequence of
  // object-specific protocol profiles , plus a type ID.
  struct IOR {
    string type_id;
    sequence <TaggedProfile> profiles;
  };

  // Multiple profiles would be encapsulated in a TaggedProfile.
  typedef unsigned long ComponentId;

  struct TaggedComponent {
    ComponentId tag;
    sequence <octet> component_data;
  };
}
```

```

};
typedef sequence<TaggedComponent> TaggedComponentSeq;
};

```

## Listing A.2: IIOP IOR Profile

```

module IIOP { // IDL extended for version 1.1, 1.2, and 1.3
  struct Version {
    octet major;
    octet minor;
  };

  struct ProfileBody_1_0 { // renamed from ProfileBody
    Version iiop_version;
    string host;
    unsigned short port;
    sequence <octet> object_key;
  };

  struct ProfileBody_1_1 { // also used for 1.2 and 1.3
    Version iiop_version;
    string host;
    unsigned short port;
    sequence <octet> object_key;

    // Added in 1.1 unchanged for 1.2 and 1.3
    sequence <IOP::TaggedComponent> components;
  };
};

```

**A.1.2 MICO**

Für diese Arbeit wurde MICO 2.3.12 mit und ohne Security Fix 001 benutzt. Der gesamte Quelltext ist auf der beigelegten CD verfügbar.

## Listing A.3: MICO: new UniqueId, orb/poa\_impl.cc

```

1089 char *
1090 MICOPOA::UniqueIdGenerator::new_id ()
1091 {
1092     char * id;
1093
1094     /*
1095      * Generate a new unique id
1096      */
1097
1098     if (uid == NULL) {
1099         ulen = 1;
1100         uid = CORBA::string_alloc (ulen);
1101         assert (uid);
1102         uid[0] = '0';
1103         uid[1] = 0;

```

```

1104 }
1105 else {
1106     int i;
1107     for (i=0; i<ulen; i++) {
1108         if (uid[i] != '9')
1109             break;
1110         uid[i] = '0';
1111     }
1112     if (i == ulen) {
1113         CORBA::string_free (uid);
1114         uid = CORBA::string_alloc (++ulen);
1115         assert (uid);
1116         for (i=0; i<ulen-1; i++) {
1117             uid[i] = '0';
1118         }
1119         uid[ulen-1] = '1';
1120         uid[ulen] = 0;
1121     }
1122     else {
1123         uid[i] = uid[i] + 1;
1124     }
1125 }
1126 id = CORBA::string_alloc (ulen + pfxlen);
1127 assert (id);
1128 if (prefix) strcpy (id, prefix);
1129 strcpy (id+pfxlen, uid);
1130
1131 return id;
1132 }

```

Listing A.4: MICO: transient prefix

```

49 // line 2071-2081 - inside RootPOA constructor
50
51 /*
52  * Generate a unique prefix for transient POAs based
53  * on the PID and the current time.
54  */
55
56 OSMisc::TimeVal ct = OSMisc::gettime ();
57
58 oaprefix = "/";
59 oaprefix += xdec (OSMisc::getpid ()); // cb: process id
60 oaprefix += "/";
61 oaprefix += xdec (ct.tv_sec); // cb: system time in s
62

```

```
63 // line 2083-89
64 /*
65  * The Root POA has a TRANSIENT Lifespan Policy.
66  * Therefore, use / <pid> / <time>
67  * as the unique name
68  */
69
70 oaid = oaprefix;
```

## A.2 Aufbau der Objektschlüssel

### A.2.1 MICO

Folgende Definitionen sind bei der Betrachtung der EBNF-Darstellung der Objektschlüssel zu beachten: `timeorb` ist die Anzahl der Sekunden nach dem 1.1.1970. Dieser Wert ist in der URL-Form im Klartext lesbar. Eine `ascii-number` ist der HEX-Wert einer Ziffer im ASCII-Format. Die Null entspricht der 0x30, die Eins der 0x31 und so weiter. `s` ist ein Separator. `poa-h` entspricht der POA-Hierarchie mit `poa` als einzelnen POA-Namen, in dieser Arbeit oftmals „null“, „einseins“ usw. bezeichnet. `timepoa` entspricht ebenfalls der Sekunden seit Beginn der UNIX-Ära, wird aber direkt in die HEX-Darstellung umgewandelt, typische Werte beginnen mit 0x44, 0x45. Der Objektzähler `object#` zählt im Hexadezimalsystem die Anzahl der jemals an diesem POA aktivierten Objekte hoch.

Diese Hinweise gelten nicht ausschließlich für MICO, sie wurden nur beispielhaft an ihm erklärt.

#### Transient

```

<MICO-tkey>      ::= <process-id> <timeorb> <poa> <object#>;

<process-id>     ::= <s1> <zahl>;
<s1>            ::= "/";
<zahl>           ::= <zifferOhneNull> { <ziffer> }
<zifferOhneNull> ::= "1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9";
<ziffer>         ::= "0" | <zifferOhneNull>;
<timeorb>       ::= <s1> <zifferOhneNull>, 9 * <ziffer>;
<poa>            ::= <s1> <ziffer> { <ziffer> };
<s2>           ::= "/_";
<object#>        ::= <s2> <ziffer> { <ziffer> };

```

### Persistent

```

<MICO-pkey> ::= <orb> <poa-h> <process-id> <time_orb> <object#>;

<orb> ::= <ascii-zeichen>a { <ascii-zeichen> } <s1>;
<s1> ::= "/";
<poa-h> ::= <poa> { <s1> <poa> };
<poa> ::= <ascii-zeichen> { <ascii-zeichen> } <s2> ;
<s2> ::= "%5C"b, <s1>;
<object#> ::= <ziffer> { <ziffer> };
<time_orb> ::= <ziffernOhneNull> 9 * <ziffer> <s2>;
<process-id> ::= <zahl>c <s2>;
<s3> ::= "/_";
    
```

<sup>a</sup>Auf die Darstellung aller ASCII Zeichen ist an dieser Stelle zu Gunsten der Übersichtlichkeit verzichtet worden.

<sup>b</sup>Der Aufbau folgt der URL-Form. Ausdrücke wie „%5C“ deuten auf einen nicht darstellbares Byte hin, dessen Hexwert statt dessen benutzt wird.

<sup>c</sup>Definition im vorherigen Abschnitt.

### A.2.2 ORBacus

#### Transient

```

<ORBacus-tey> ::= <magic_orb> <time_orb> <poa-h>
                 <magic_java> <time_poa> <object#>;

<magic_orb> ::= "AB AC AB 30";
<time_orb> ::= 10 * <ascii-number>;
<ascii-number> ::= "30"|"31"|"32"|"33"|"34"|"35"|"36"|"37" |
                  "38"|"39";
<s> ::= "5F";
<poa-h> ::= <s> <poa> { <>nullbyte> <poa> } 2 * <>nullbyte>;
<poa> ::= <ascii-zeichen>1;
<time_poa> ::= 4 * <byte>2;
<>nullbyte> ::= "00";
<object#> ::= 4 * <byte>;
<magic_java> ::= "CA FE BA BE";
    
```

### Persistent

```

<ORBacus-pkey> ::= <magic_orb> <poa-h> <magic_java> <time_poa> <object#>;

<magic_orb>      ::= "AB AC AB 30";
<s>              ::= "00 5F";
<poa-h>         ::= <s> <poa> { <nullbyte> <poa> } 2 * <nullbyte>;
<poa>           ::= <ascii-zeichen>a;
<time_poa>      ::= 4 * <byte>;
<nullbyte>      ::= "00";
<object#>       ::= 4 * <byte>;
<magic_java>    ::= "CA FE BA BE";
    
```

<sup>a</sup>Auf die Darstellung aller ASCII Zeichen ist auch an dieser Stelle verzichtet worden.

### A.2.3 Orbix

```

<Orbix-key> ::= <magic_orb> <poa_policy> <poa-id> <object#>;

<magic_orb>      ::= "3A 3E";
<poa_policy>     ::= <len_policy>, "31 31"|"31 32"a;
<len_policy>     ::= "02";
<poa-id>         ::= <len_poa-id>, 12 * <byte>b;
<len_poa-id>     ::= "0C";
<object#>        ::= <len_object#>, 8 * <byte>c;
<len_object#>   ::= "08";
    
```

<sup>a</sup>Je nachdem, ob der POA persistente oder transiente Objekte verwaltet.

<sup>b</sup>Es handelt sich offensichtlich um eine Zufallsfolge.

<sup>c</sup>Als ein bei Null beginnender, inkrementierender Zähler.

## A.2.4 SUN-ORB

```

<Sun5-tkey> ::= <magic_orb> <poa_lt-policy> <orb-id>
               <const1> <poa-h> <object#> <const2>;

<magic_orb> ::= "AF AB CD 00";a
<poa_lt-policy> ::= "00 00 00", "20"|"22";
<orb-id> ::= 4 * <byte>;
<const1> ::= "00 00 00 01 00 00 00 00";
<poa-h> ::= <poa_deep> <poa> { <poa> };
<poa_deep> ::= 4 * <byte>;
<poa> ::= <poa_strlen> <poa_name>;
<poa_strlen> ::= 4 * <byte>;
<poa_name> ::= <ascii-char> { <ascii-char> };
<object#> ::= <len_object#> 8 * <byte> <const2>;
<len_object#> ::= "08";
<const2> ::= "14";b

```

<sup>a</sup>In Java 1.4.x betrug dieser Wert AF AB CB 00.

<sup>b</sup>In Java 1.4.x betrug dieser Wert 0A.

## A.2.5 Visibroker

### Transient

```

<VB-tkey> ::= <magic_orb> <poa-h> <object#> <time_poa>;
<magic_orb> ::= "00 56 42 01";
<poa-h> ::= <len_poa-h> <poa> { <poa> };
<poa> ::= "2F" <poa_name>;
<poa_name> ::= <ascii-char> { <ascii-char> };
<object#> ::= <len_object#> 4 * <byte>;
<len_object#> ::= "00 00 00 04";
<time_poa> ::= <t1> <t2>;
<t1> ::= 4 * <byte>;
<t2> ::= 4 * <byte>;

```

### Persistent

```

<VB-pkey> ::= <magic_orb> <const> <poa-h> <object#> ;
<magic_orb> ::= "00 56 42 01";
<const> ::= "00 00 00 04";
<poa-h> ::= <len_poa-h> <poa> { <poa> };
<poa> ::= "2F" <poa_name>;
<poa_name> ::= <ascii-char> { <ascii-char> };
<object#> ::= <len_object#> 4 * <byte>;
<len_object#> ::= "00 00 00 04";

```

## A.2.6 Weblogic

Zusammenfassend ergibt sich folgender Schlüsselaufbau:

```

<WL-tkey> ::= <magic_orb1> <const1> <IDL> <poa> <magic_orb2>
             <len_part2> <const2> <orb-id> <const3> <object#>;
<magic_orb1> ::= "00 42 45 41";
<const1> ::= "08 01 03 00 00 00 01"8 * <nullbyte>;
<nullbyte> ::= "00";
<IDL> ::= <len_IDL> <type> <nullbyte>;
<len_IDL> ::= 4 * <byte>;
<type> ::= ĪDL:" <ascii-char> { <ascii-char> };
<poa> ::= <len_poa#> <poa#>;
<poa#> ::= <zifferOhneNull> { <ziffer> };
<magic_orb2> ::= "42 45 41 11";
<len_part2> ::= 4 * <byte>;
<const2> ::= 8 * <nullbyte>;
<orb-id> ::= "00 00 00 00"|"FF FF FF FF", 4 * <byte>;;
<const3> ::= "7F FF FF 02<<rmi>;
<rmi> ::= "00 00 00 18 52 4D 49 3A 5B 42 3A 30 30 30",
           "30 30 30 30 30 30 30 30 30 30 30 00";
<object#> ::= <len_object#> <zifferOhneNull> { <ziffer> };

```

## A.2.7 WebSphere

Der Aufbau in EBNF:

```

<WB-key> ::= <magic_orb> <LC-policy> <poa-h> <object#>;
<magic_orb> ::= "4C 4D 42 49";
<LC-policy> ::= "00 00 00 15"|"00 00 00 16";
<poa-h> ::= <len_prefix> <len_poa-h> <poa> { <poa> };
<len_prefix> ::= 2 * <byte>;
<len_poa-h> ::= 2 * <byte>;
<poa> ::= <len_poa> <poa_name>;
<len_poa> ::= <byte>;
<poa_name> ::= <ascii-char> { <ascii-char> };
<object#> ::= <len_object#> 4 * <byte>;
<len_object#> ::= "00 04";

```

# Erklärung

Ich erkläre, diese Arbeit selbstständig angefertigt und die benutzten Unterlagen vollständig angegeben zu haben.

Berlin, 20. September 2006

---

Christoph Becker

## Literaturverzeichnis

- [Bec06] Christoph Becker. Vorhersage von Objektidentifikationen in CORBA, February 2006. Studienarbeit.
- [BS02] Gerald Brose and Sebastian Staamann. Application Security Gateways: Eine Komplettlösung für die Sicherheit von Applikationsservern. *Objektspektrum*, pages 50–53, July 2002.
- [BVD01] Gerald Brose, Andreas Vogel, and Keith Duddy. *Java Programming with CORBA*. Wiley, 3rd edition, 2001.
- [Eck06] Claudia Eckert. *IT-Sicherheit: Konzepte – Verfahren – Protokolle*. Oldenbourg, 2006.
- [HJS01] Johann Hofmann, Fritz Jobst, and Roland Schabenberger. *Programmieren mit COM und CORBA*. Hanser, 2001.
- [Jet01] Thomas Jetzler. IIOP firewalls. Master’s thesis, University of Zurich, July 2001.
- [Kli00] William Ruh; Thomas Herron; Paul Klinker. *IIOP Complete - Understanding CORBA and Middleware Interoperability*. Addison-Wesley, 2000.

- [OMG04] Object Management Group OMG. Common object request broker architecture: Core specification, version 3.0.3. <http://www.omg.org>, March 2004. formal document 04-03-01.
- [Say97] Michael Sayegh. *CORBA - Standard, Spezifikationen, Entwicklung*. O'Reilly Verlag, Köln, 1997.
- [SB06] Sebastian Staamann and Christoph Becker. Using corba object references as authorization tokens – using a good idea or not? <http://www.omg.org/news/meetings/workshops/>, February 2006. Realtime Workshop, Arlington.
- [Sie96] Jon Siegel. *CORBA - Fundamentals ans Programming*. Wiley, 1996.
- [TvS03] Andrew S Tanenbaum and Maarten van Steen. *Verteilte Systeme: Grundlagen und Paradigmen*. Addison Wesley, 2003.