INSTITUT FÜR INFORMATIK
DER UNIVERSITÄT KOBLENZ-LANDAU

Diplomarbeit

Entwurf und Implementierung eines Sicherheitsprotokolls in das Job-Management-System CODINE

Torsten Bieder

Oktober 1996

Aufgabenstellung und Betreuung:

Dipl.-Math. Fritz Ferstl, Genias Software GmbH Neutraubling
Prof. Dr. Wolfgang Gentzsch, Genias Software GmbH Neutraubling
Prof. Dr. Christoph Steigner, Universität Koblenz-Landau

1 Vorwort

Die vorliegende Diplomarbeit entstand im Rahmen meiner Ausbildung zum Diplom-Informatiker an der Universität Koblenz-Landau und wurde extern bei der Genias Software GmbH in Neutraubling durchgeführt. An dieser Stelle möchte ich mich ganz besonders bei den Betreuern meiner Arbeit Fritz Ferstl, Prof. Wolfgang Gentzsch, und Prof. Christoph Steigner bedanken, die mir die Durchführung dieses Projekts erst ermöglichten.

Mein Dank gilt aber auch allen Mitarbeitern der Genias Software GmbH, sowie allen Diplomanden und Praktikanten, die mich zu jeder Zeit durch praktische Tips und Tricks unterstützten und so dazu beitrugen, daß ich meine Diplomarbeit erfolgreich abschließen konnte.

Meine Zeit bei Genias hat mir nicht nur einen praxisorientierten Einblick in die Arbeitsweise eines Betriebes mittlerer Größe verschafft, sondern hat mir darüber hinaus auch sehr viel Freude bereitet - ich habe viele neue Erfahrungen sammeln können, vor allem Fachwissen, aber auch Dinge aus dem Bereich des kollegialen Zusammenlebens.

Natürlich möchte ich auch ganz herzlich meiner Familie danken, meiner Verlobten Daniela, meiner Mutter und meinen Großeltern, für die geleistete seelische und finanzielle Unterstützung während meines Studiums.

Inhaltsverzeichnis

1 Vorwort 1

2 Einleitung 6

3 Zielsetzung und Lösungsweg 8

4 Zeittafel 10

5 Was ist eine sichere Kommunikation? 11

5.1 Sicherheitsaspekt: Vertraulichkeit (privacy) 12

5.1.1 Symmetrische Kryptographie-Verfahren 12

5.1.2 Beispiel: DES 13

5.1.3 Asymmetrische Kryptographie-Verfahren 15

5.1.4 Beispiel: RSA 16

5.1.5 Kommunikationsbeispiel 17

5.2 Sicherheitsaspekt: Authentifikation 19

5.2.1 Kommunikationsbeispiel 1 19

5.2.2 Beispiel: X.509 Zertifikate 20

5.2.3 Kommunikationsbeispiel 2 21

5.3 Sicherheitsaspekt: Datenintegrität 23

5.3.1 Beispiel: MD5 23

5.3.2 Kommunikationsbeispiel 24

5.4 Zusammenfassung 26

6 Was ist CODINE? 27

6.1 Komponenten von CODINE 27

6.2 Hosts und Benutzergruppen von CODINE 28

6.3 Arbeitsweise des CODINE-Systems 29

6.3.1 Job an *qmaster* übermitteln 29

6.3.2 Job zuteilen *(schedule)* 30

6.3.3 Job an *execd* übermitteln 31

6.4 Kommunikation von CODINE 31

7 Warum Sicherheit für CODINE? 33

8 Das SSL-Protokoll 35

8.1 Die Konzeption des SSL-Protokolls 35

8.2 Die Implementierung des SSL-Protokolls 38

8.3 Die Integration des SSL-Protokolls in CODINE 39

9 Die Konzeption von CSP 41

9.1 Erklärung einiger Begriffe 41

9.2 Ablauf von CSP 43

10 Die Implementierung von CSP 48

10.1 Beeinflussende Faktoren 48

10.2 Die Bestandteile von CSP 48

10.2.1 Die Header Files 49

10.2.2 Die Bibliothek *libsec.a* 56

10.2.3 Die SSL-Bibliothek *libcrypto.a* 61

10.2.4 Pfade und Dateien 61

10.2.5 Die Applikationen 64

11 Die Integration von CSP in CODINE 69

11.1 Einsatz von CSP in CODINE 69

11.1.1 Übersetzen des Codes 69

11.1.2 Generierung von Zertifikaten und Schlüsseln 69

11.1.3 Eingabe des Paßwortes 70
11.2 Änderungen im CODINE-Code 70
11.2.1 AIMK 71
11.2.2 *Makefile* 71
11.2.3 Einführung neuer *Message Tags* 71
11.2.4 Erweiterung der *Error Return Codes* der *commlib* . . . 72
11.2.5 Zusätzliche *Exit*-Funktion 72
11.2.6 Realisierung der *Connection List* 73
11.2.7 Änderung im Code der Komponenten 73
11.2.8 Änderung im Code vom *qmaster* 73
11.2.9 Substitution der Kommunikationsaufrufe 73
11.3 Mehraufwand durch den Einsatz von CSP 75

12 Zusammenfassung und Ausblick 77

Literaturverzeichnis 79

A Source Code 81

Abbildungsverzeichnis

1 Ablauf des DES-Algorithmus' 14
2 Alice und Bob tauschen eine vertrauliche Nachricht aus 18
3 Mallet fängt die Daten von Alice ab und gibt sich als Bob aus 20
4 Authentifizierung mit Zertifikaten 22
5 MD5 Arbeitsablauf 24
6 Die vertrauliche Nachricht wird mit einem MAC „geschützt" . 25
7 Ein Job wird an das CODINE System übermittelt 30
8 Ablauf des SSL-Protokolls 36
9 Das *CODINE Security Protocol* CSP 43
10 Client Announce 44
11 Server Response 45
12 Secure Message (Client → Server) 46
13 Secure Message (Server → Client) 47

2 Einleitung

Sicherheit gewinnt immer mehr an Bedeutung und nimmt vor allem in vernetzten Computer-Systemen einen immer höheren Stellenwert ein. Gerade mit der zunehmenden Verbreitung des *Internets* und damit auch des *World Wide Webs* (WWW) wächst im gleichen Maße das Bedürfnis, Daten sicher und geschützt zu übertragen, seien es nun interne, vertrauliche Firmendaten oder *Electronic Cash*, das bargeldlose Bezahlen im Netz.

Ein Merkmal der digitalen Kommunikation, ob erwünscht oder nicht, ist die Möglichkeit, die übertragenen Daten schnell und zielgerichtet maschinell auszuwerten. So gewonnene Informationen bedeuten unter Umständen viel Macht über den Einzelnen. Nicht umsonst verbieten einige Regierungen (z.B. die französische) den Einsatz von Kryptographie-Methoden, mit der Begründung, kriminellen Machenschaften eher auf die Spur zu kommen. Wenn es nach dem deutschen Innenminister Kanther ginge, dürften im *Internet* nur bekannte Verschlüsselungsalgorithmen verwendet werden und die Schlüssel sollten alle in einer zentralen Stelle in Berlin verwaltet werden. Aber Nutzen und Mißbrauch liegen auch hier wieder nahe beieinander, ähnlich wie bei dem viel diskutierten und umstrittenen *großen Lauschangriff* in Deutschland. Angenommen, die Regierungenen behandeln alle Daten vertraulich, so können Korruption und Datendiebstahl, die nicht auszuschließen sind, immer noch großen Schaden verursachen. Eben aus Angst vor Schaden wächst immer mehr der Wunsch nach Möglichkeiten, gewisse Daten geheim zu halten, genau das, womit sich die Wissenschaft der Kryptographie beschäftigt.

Die vorliegende Ausarbeitung soll keine umfassende Einführung in die Kryptographie darstellen, hierfür ist bereits ausgezeichnete Literatur vorhanden, die dieses Thema besser und ausführlicher abdecken kann, als es im Rahmen dieser Diplomarbeit möglich ist. Ich möchte nur soweit in dieses Gebiet vordringen, wie es zum Verständnis notwendig ist. An entsprechender Stelle werde ich jedoch dann auf weiterführende Literatur verweisen. Der Kern der Ausarbeitung soll viel mehr in der eigentlichen Arbeit bei der Genias Software GmbH Neutraubling liegen, nämlich der Entwurf und die Implementierung eines Sicherheitsprotokolls in die bestehende Anwendung CODINE (*Computing in Distributed Networked Environments*), einem Job-Management-System.

Zum Thema der Kryptographie ist gerade in letzter Zeit ausreichend Literatur veröffentlicht worden, die sich sowohl mit den Grundlagen, als auch mit weiterführenden Gebieten beschäftigt. Als das Standardwerk der Kryptographie ist das Buch [20] *Applied Cryptography* von Bruce Schneier zu nennen.

Für den Schnelleinstieg in dieses Themenfeld kann ich den Artikel [9] aus der Zeitschrift *c't* empfehlen. Im WWW kann man sich unter der URL (*Uniform Resource Locater*) aus [22] ebenfalls einen schnellen Überblick über dieses Gebiet verschaffen. Außerdem findet man auch hier viele Verweise auf weitere Quellen, seien es Literatur und Software in jeglicher Form, oder *Newsgroups*, die Diskussionsrunden im Internet.

Gerade das Internet und insbesondere das World Wide Web haben sich als wahre Fundgrube von Informationen für das Gebiet der Kryptographie erwiesen. Zunächst ist es nötig, relevante Quellen aus den unglaublich großen Datenmengen herauszufiltern. Dann jedoch erhält man einen umfassenden, aktuellen und weltweiten Wissensstand zu diesem Thema. Zudem hat man schnellen Zugriff auf die neueste Software und bleibt z.B. mit Hilfe von *Mailing Listen* auf dem aktuellen Stand der Entwicklung. Aber auch umgekehrt werden manche Softwareprojekte erst durch eine Veröffentlichung im Internet möglich. Gerade im Bereich der Kryptographie haben oft kleinste Fehler fatale Folgen. Durch eine möglichst weite Publizierung werden solche Fehler mit größerer Wahrscheinlichkeit entdeckt, sodaß öffentliche Software oft ein höheres Sicherheitsniveau als kommerzielle erlangt. Gute Beispiele dafür sind *Pretty Good Privacy (PGP)* [8] oder die Implementierung des *SSL-Protokolls* von Eric Young [25], deren Entwicklung jedoch noch nicht vollständig abgeschlossen ist. Im weiteren Verlauf dieser Arbeit werde ich noch weiter auf das SSL-Protokoll eingehen, das ich erst durch das Hilfsmittel Internet anwenden konnte.

Kennzeichnend für die Informatik ist das Vorkommen englisch sprachiger Fachausdrücke, die gerade in diesem Gebiet, wie in kaum einem anderen, Verwendung finden. In der vorliegenden Diplomarbeit habe ich versucht einen Mittelweg zu finden, allgemein bekannte Ausdrücke im Englischen zu belassen und da zu übersetzen, wo es auch geläufige Ausdrücke in der deutschen Sprache gibt. Dies war nicht immer ganz einfach, aber ich hoffe dennoch für den Leser mit Kenntnissen in der Informatik in den meisten Fällen den richtigen Weg gewählt zu haben.

3 Zielsetzung und Lösungsweg

Das Ziel dieser Diplomarbeit ist es, die Kommunikation von CODINE, einem Softwareprodukt der Genias GmbH Neutraubling, sicherer zu gestalten, um den Anwender und Betreiber von CODINE vor möglichem Schaden zu bewahren. Diese Zielsetzung kann durch ein Sicherheitsprotokoll erreicht werden, das zuerst entworfen werden muß. Beim Entwurf steht natürlich die Sicherheit an erster Stelle, aber genauso müssen auch die Gegebenheiten des Umfeldes berücksichtigt werden, in das das Protokoll implementiert werden soll. Die Implementierung selbst benötigt verschiedene Mechanismen der Kryptographie, die möglichst transparent für Entwickler und Anwender arbeiten sollen. Eine Realisierung des Sicherheitsprotokolls in Form einer Bibliothek definiert eine klare Schnittstelle zum System und erleichtert die Einsetzbarkeit.

Im Laufe dieser Ausarbeitung möchte ich nun obige, allgemein formulierte Aufgabenstellung schrittweise präzisieren und den Lösungsweg, der gewählt wurde, aufzeigen und begründen.

Nach den einleitenden Kapiteln beschreibt Kapitel 5, was unter einer sicheren Kommunikation zu verstehen ist. Es werden die drei Sicherheitsaspekte Vertraulichkeit, Authentifikation und Datenintegrität vorgestellt und anhand von Kommunikationsbeispielen erläutert. Zudem wird aufgezeigt, wie diese Aspekte in der Computerpraxis realisiert werden können. Eine Zusammenfassung der wichtigsten Ergebnisse schließt dieses Kapitel ab.

In Kapitel 6 wird das Umfeld des neuen Sicherheitsprotokolls, nämlich CODINE, näher beschrieben. Dabei werden die Komponenten und die Arbeitsweise von CODINE dargestellt und ein Abschnitt beschäftigt sich mit der Kommunikation von CODINE, die sicherer zu gestalten ist.

Nachdem aufgezeigt wurde, was eine sichere Kommunikation ausmacht und wie CODINE arbeitet, ist es nun in Kapitel 7 möglich, die Motivation zur Aufgabenstellung zu beschreiben, warum es notwendig ist, ein Sicherheitsprotokoll für das Job-Management-System CODINE zu entwerfen und zu implementieren.

In Kapitel 8 werden erste Lösungswege beschritten. Es werden Konzeption und Implementierung des *Secure Socket Layer* (SSL) Protokolls vorgestellt, das alle geforderten Sicherheitsaspekte erfüllt und leicht in bestehende Anwendungen zu integrieren ist. Schließlich wird erläutert, warum dieses Sicherheitsprotokoll in seiner ursprünglichen Form nicht in CODINE eingesetzt werden kann.

Die drei folgenden Kapitel beschäftigen sich ausführlich mit der Lösung des Problems. Es wird das *CODINE Security Protocol* (CSP) vorgestellt. Dieses Sicherheitsprotokoll wurde im Rahmen dieser Diplomarbeit entwickelt und ist an die speziellen Bedürfnisse von CODINE angepaßt.

Kapitel 9 beschreibt die Konzeption von CSP. Detailliert werden die einzelnen Phasen des neuen Protokolls dargestellt und es wird erläutert, welche Aktionen dabei ausgeführt werden.

Die Implementierung von CSP wird in Kapitel 10 vorgestellt. Es werden die einzelnen Bestandteile des Protokolls beschrieben, ihr Aufbau und ihre Funktionsweise.

Das folgende Kapitel 11 erläutert schließlich, wie CSP in CODINE einzusetzen ist, welche Änderungen im CODINE-Code vorzunehmen sind und welche Auswirkungen sich daraus für das Gesamt-System, für den Administrator und für den Benutzer ergeben.

Zum Abschluß werden die Ergebnisse dieser Arbeit im Kapitel 12 zusammengefaßt. Es wird aufgezeigt, was erreicht wurde, aber auch, was noch zu tun ist, um CSP beim Kunden von CODINE in der Praxis einsetzen zu können.

4 Zeittafel

Diese Zeittafel soll einen Überblick meiner Arbeit bei Genias Software GmbH verschaffen. Hieraus kann man ersehen, daß etwa je ein Drittel der Zeit an Einarbeitung/Planung, Implementierung/Testen und Erstellung der Ausarbeitung aufgewendet werden mußte. Vielleicht können diese Erkenntnisse bei der Planung und Durchführung anderer Diplomarbeiten hilfreich sein.

9/95	Einarbeitung in CODINE und in die Client-/Server-Kommunikation
9/95 - 10/95	Installation und Testen von *identd*[1] im Genias Rechnernetzwerk Einarbeitung in die Methoden der Kryptographie
10/95	Implementation der DES-Verschlüsselung in verschiedene Client-/Server-Testprogramme
11/95 - 12/95	Einarbeitung in das SSL-Protokoll und dessen Implementation *SSLeay* Integration des SSL-Protokolls in wiederum verschiedene Client-/Server-Programme
1/96	Entwurf eines Security-Protokolls für CODINE
1/96 - 3/96	Implementierung des neuen Protokolls in CODINE
4/96	Testen dieser Implementierung Programmierung einiger Hilfsprogramme für CODINE zur Schlüsselgenerierung und zum Zertifikatmanagement
4/96 - 6/96	Erstellung der schriftlichen Ausarbeitung der Diplomarbeit
7/96	Abschluß der Diplomarbeit mit den letzten Feinkorrekturen an Code und Ausarbeitung

[1]Das *Identification Protocol* wird in [11] beschrieben und die Implementation *identd* ist unter [6] verfügbar. Dieser *Identification Deamon* dient zur Authentifizierung von Benutzern im Rechnernetz.

5 Was ist eine sichere Kommunikation?

Bevor ich ein Sicherheitsprotokoll für CODINE vorstellen kann, ist es sinnvoll, zunächst zu erläutern, welche Anforderungen an ein solches Protokoll gestellt werden. Allgemein läßt sich sagen, daß es das Ziel eines Sicherheitsprotokolls ist, die Kommunikation sicher zu gestalten, Kommunikation im Hinblick auf digitalen Datenaustausch in Computernetzwerken.

Eine sichere Kommunikation sollte zumindest die drei Sicherheitsaspekte

- *Vertraulichkeit*
- *Authentifikation*
- *Datenintegrität*

erfüllen, die ich in den folgenden Abschnitten anhand von Kommunikationsbeispielen näher erläutern will. Zudem werde ich für jeden Sicherheitsaspekt Methoden beschreiben, die diese Aspekte in der Computerpraxis realisieren können.

Die Sicherheit dieser Methoden sollte jedoch auf keinen Fall auf der Geheimhaltung der Algorithmen beruhen; mit Hilfe eines Disassemblers wären sie zu leicht zu überwinden. Man muß also davon ausgehen, daß Unbefugte immer eine Möglichkeit finden, einen Algorithmus zu durchleuchten. So gibt es beispielsweise schon diverse Hilfsmittel um „verschlüsselte“ Texte von Standardtextverarbeitungen ohne Kenntnis des Paßwortes, nur durch Analyse des Algorithmus', zu entschlüsseln.

Der richtige Weg ist also, Methoden der Kryptographie so zu gestalten, daß die Sicherheit einer Nachricht ganz alleine von der Wahl des Paßwortes abhängt. Eine Geheimhaltung des Algorithmus' sollte nicht mehr notwendig sein. Ganz im Gegenteil, eine möglichst weite Veröffentlichung, beispielsweise im Internet, trägt eher noch zur Sicherheit bei, denn für jeden Programmierer von Kryptographie-Software ist es fast unmöglich alle Sicherheitslücken einer Implementation zu überschauen, die jedoch durch eine Veröffentlichung eventuell geschlossen werden können. Dies ist auch einer der Gründe, warum ich auf „bewährte“ Software, wie *SSLeay* ([23] - [26]), zurückgegriffen und die Methoden der Kryptographie nicht neu implementiert habe.

5.1 Sicherheitsaspekt: Vertraulichkeit (privacy)

Eine Kommunikation ist *vertraulich*, wenn niemand anderes als der beabsichtigte Empfänger die Botschaft lesen kann. In konventionellen Rechnernetzen werden im allgemeinen alle Daten im Klartext versendet. Jeder, der sich irgendwie Zugang zum Netz verschaffen kann, hat auch prinzipiell den Zugriff auf die übermittelten Daten. Um nun die Vertraulichkeit digitaler Nachrichten zu gewährleisten, werden diese mit geeigneten Verfahren verschlüsselt. Die Verschlüsselungsverfahren lassen sich in *symmetrische (secret key)* und *asymmetrische (public key)* Algorithmen aufteilen.

5.1.1 Symmetrische Kryptographie-Verfahren

Symmetrische Verfahren zeichnen sich dadurch aus, daß zum Entschlüsseln einer Nachricht derselbe Schlüssel benötigt wird, mit dem sie verschlüsselt wurde.

`[ {M}key ]key = M`

M steht hier für eine digitale Nachricht

{M}key heißt, daß M mit *key* verschlüsselt wird
[M] key heißt, daß M mit *key* entschlüsselt wird

Der Schlüssel *key* muß daher auch geheim gehalten werden; nur der Sender und der beabsichtigte Empfänger dürfen den Schlüssel kennen.

Der Vorteil von symmetrischen Verfahren liegt in ihrer hohen Verarbeitungsgeschwindigkeit, der Nachteil ist, daß beide Kommunikationspartner über einen gemeinsamen geheimen Schlüssel verfügen müssen. Wollen so zum Beispiel n Teilnehmer jeweils untereinander sicher kommunizieren, müssen $\frac{n(n-1)}{2}$ Schlüssel verwaltet werden. In einer multi-user Umgebung wird so eine sichere Schlüsselverwaltung unpraktikabel.

Natürlich besteht auch die Möglichkeit, daß sich alle Teilnehmer denselben Schlüssel teilen, um sich vor Angriffen von außen zu schützen. Dies ist allerdings nur in kleinen Netzwerken vorstellbar, bei denen sich alle Beteiligten untereinander vertrauen können. Bei größeren Firmennetzen oder gar dem Internet ist ein solcher Ansatz dann nicht mehr sinnvoll durchführbar.

5.1.2 Beispiel: DES

Als Beispiel für ein symmetrisches Kryptographieverfahren will ich DES anführen. DES steht für **D**ata **E**ncryption **S**tandard und wurde erstmals 1977 vom National Bureau of Standards veröffentlicht [16] und beruht auf einer Entwicklung von IBM. Er gilt als weithin sicher, obwohl oder gerade weil er schon relativ alt und damit auch praxisbewährt ist.

DES ist ein blockorientierter Algorithmus, d.h. standardmäßig wird ein 56 Bit langer Schlüssel immer auf 64 Bit lange Datenblöcke angewandt. Hierfür muß zunächst der 56-Bit Schlüssel in 16 verschiedene 48-Bit Unterschlüssel zerlegt werden. Dann werden die 64 Bit des Datenblocks nach vorgeschriebener Weise permutiert und in 16 Runden werden die 16 Subschlüssel mit den Datenbits im wesentlichen *xor* verknüpft, wobei die 64 resultierenden Datenbits wieder als Eingabe für die nächste Runde dienen. Abschließend erfolgt eine letzte Permutation des Datenblockes, die eine Umkehrung der ersten Permutation darstellt. Die Abbildung 1 (entnommen aus [12]) auf Seite 14 beschreibt diesen Vorgang nochmals anschaulich.

Zum Entschlüsseln muß der Vorgang nur in umgekehrter Reihenfolge ausgeführt werden. Die hier dargestellte Form bildet die einfachste Art der DES-Verschlüsselung (ECB-Modus). DES kann jedoch auch in anderen Modi betrieben werden:

Electronic Code Book (ECB)
Diese Methode wurde zuvor beschrieben, einfache 64-Bit Blöcke werden mit dem Schlüssel sequentiell verschlüsselt.

Cipher Block Chaining (CBC)
Im CBC Modus werden die letzten 8 Bits des letzten verschlüsselten Blocks mit dem nächsten 64-Bit Datenblock verknüpft, bevor auch dieser verschlüsselt wird. Für den ersten Datenblock werden diese 8 Bits zufällig generiert und heißen *Initialization Vector*, kurz IV, der beiden Kommunikationsteilnehmern, wie der geheime Schlüssel, bekannt sein muß. CBC hat gegenüber ECB den Vorteil, daß ein verschlüsselter Datenblock nicht mehrfach hintereinander unbemerkt gesendet werden kann und daß zwei gleiche Datenblöcke verschlüsselt nicht die gleiche Gestalt haben.

Dann gibt es noch den **Output Feedback (OFB)** Modus und den **Cipher Feedback (CFB)** Modus, die ähnlich wie CBC arbeiten, nur daß die Bits, die mit den Datenblöcken verknüpft werden, auf eine andere Weise gewonnen werden.

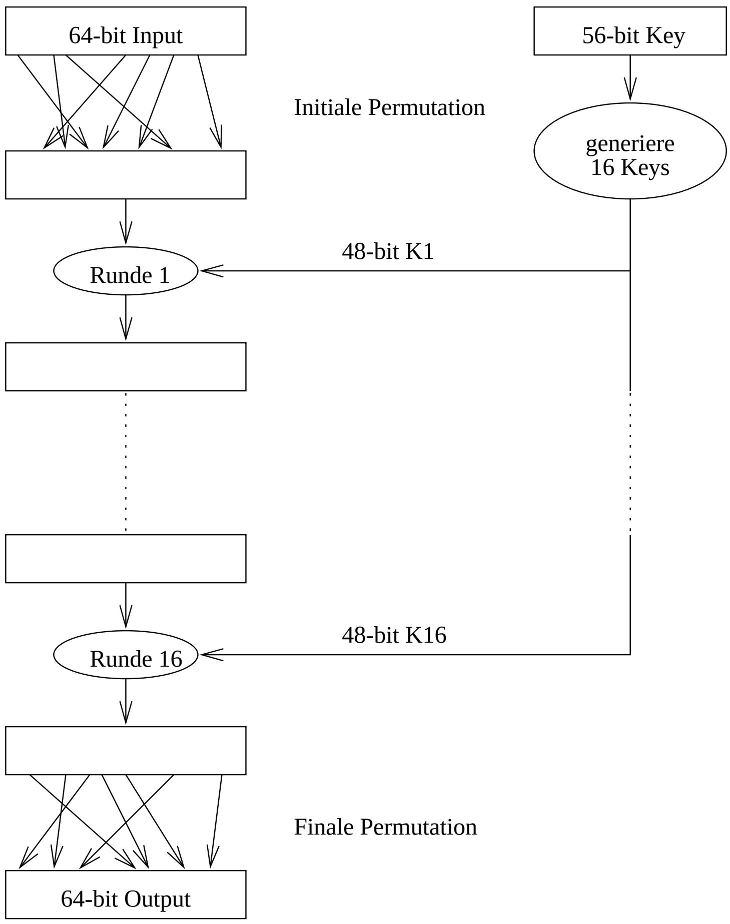

Abbildung 1: Ablauf des DES-Algorithmus'

Encrypt - Decrypt - Encrypt (EDE)
EDE kann mit allen vorher genannten Modi kombiniert werden und unterscheidet sich von der einfachen DES-Verschlüsselung dadurch, daß anstatt aus 64 Bit einen Schlüssel zu generieren, aus 192 Bit drei Schlüssel abgeleitet werden, um die Schlüssellänge effektiv zu verdreifachen. Diese drei Schlüssel werden nun nacheinander auf die Daten angewendet: Der erste verschlüsselt die Nachricht, der zweite wendet nun die Entschlüsselungsroutine auf die resultierenden Daten an und zuletzt werden die Daten wieder mit dem dritten Schlüssel verschlüsselt.

Der genaue Ablauf der einzelnen Schritte der DES-Verschlüsselung kann bei [7] nachvollzogen oder auch in fast jedem Buch nachgelesen werden, das die Grundzüge der Kryptographie vermittelt, wie z.B. [12] oder [20], die sich auch eingehender mit den verschiedenen Modi von DES beschäftigen.

5.1.3 Asymmetrische Kryptographie-Verfahren

Bei asymmetrischen Verfahren benötigt jeder Teilnehmer ein Schlüsselpaar, einen privaten, geheimen Schlüssel und einen öffentlichen, für jeden frei zugänglich. Will nun A eine Nachricht an B senden, dann schaut A nach dem öffentlichen Schlüssel von B und verschlüsselt die Nachricht damit. Nur B kann dann die Nachricht mit seinem privaten Schlüssel entschlüsseln.

```
[ {M}public-key ]private-key = M
```

```
oder umgekehrt
```

```
[ {M}private-key ]public-key = M
```

Bei letzterer Variante handelt es sich nicht wirklich um ein Verschlüsselungsverfahren, weil der public-key für jeden zugänglich ist und somit kann jeder diese Nachricht entschlüsseln. Bei Nachrichten, die mit dem privaten Schlüssel „verschlüsselt“ werden, handelt es sich vielmehr um eine *signierte* Nachricht, weil nur der Urheber der Nachricht seinen privaten Schlüssel kennt. Jeder kann nun diese Signatur mit dem öffentlichen Schlüssel des Urhebers überprüfen. Das Buch [21] setzt sich ausführlich dem Thema der *Public-Key Cryptography* auseinander.

Gegenüber den symmetrischen Verfahren ist hier das Schlüsselmanagement wesentlich einfacher, jeder Teilnehmer braucht nur ein einziges

Schlüsselpaar, unabhängig von der Gesamtzahl der Teilnehmer. Allerdings sind asymmetrische Verfahren wesentlich langsamer in ihrer Verarbeitungsgeschwindigkeit. So ist DES um den Faktor 1000 schneller als der asymmetrische RSA-Algorithmus. Aus diesem Grund werden beide Verfahren in der Praxis kombiniert.

5.1.4 Beispiel: RSA

Das weitverbreitetste asymmetrische Verschlüsselungsverfahren ist RSA, ein Kürzel, das die Initialien der drei Entwickler **R**ivest, **S**hamir und **A**dleman repräsentiert. Es beruht darauf, daß vom Produkt zweier großer Primzahlen nicht mehr ohne weiteres auf die Zahlen selbst geschlossen werden kann und es überstand seit 1978 viele Jahre der Kryptoanalyse. Können jedoch schnelle Algorithmen zur Faktorisierung großer Zahlen gefunden werden, so fällt die Sicherheit von RSA.

Um einen öffentlichen und einen privaten Schlüssel zu erhalten, müssen zunächst zwei große Primzahlen p und q gefunden werden (bespielsweise 200 Stellen pro Zahl). Dann ist n das Produkt von p und q. Nun muß e teilerfremd zu $\varphi(n) = (p-1)(q-1)$ gewählt werden und man erhält den öffentlichen Schlüssel $\langle e,n \rangle$. Danach ist eine natürliche Zahl d zu finden, die die Gleichung $ed = 1 \bmod \varphi(n)$ erfüllt. Hieraus ergibt sich der private Schlüssel $\langle d,n \rangle$, der geheim gehalten werden muß.

Um nun eine Nachricht m zu verschlüsseln, muß m zunächst in eine Zahlendarstellung überführt werden (z.B. ASCII-Code) und gegebenenfalls in geeignete Blöcke zerlegt werden, damit $m < n$ erfüllt bleibt. Mit dem öffentlichen Schlüssel $\langle e,n \rangle$ und

$$c = m^e \bmod n$$

erhält man den chiffrierten Text c. Wird $m > n$ gewählt, so können zwei verschiedene Nachrichten verschlüsselt auf den selben Wert abgebildet werden und die Entschlüsselung wäre nicht mehr eindeutig.

Nur wer den passenden privaten Schlüssel $\langle d,n \rangle$ kennt, kann jetzt die verschlüsselte Nachricht c wieder dechiffrieren.

$$m = c^d \bmod n$$

Die Nachricht m erscheint nun wieder im Klartext. Verschlüsselt man aber m mit dem privaten Schlüssel, so erhält man eine Signatur s:

$$s = m^d \bmod n$$

Jeder kann nun mit dem dazugehörigen öffentlichen Schlüssel diese Signatur überprüfen:

$$m = s^e \bmod n$$

5.1.5 Kommunikationsbeispiel

Am anschaulichsten läßt sich der Sicherheitsaspekt der Vertraulichkeit (und auch die folgenden Sicherheitsaspekte) an einer Beispielkommunikation darstellen. Die Hauptakteure sind, wie in den Beipielen der Kryptographie üblich, *Alice* und *Bob* als Kommunikationsteilnehmer und der böse *Mallet*, der die Kommunikation zwischen den beiden infiltrieren will.

Alice beabsichtigt also eine vertrauliche Nachricht an Bob zu schicken, die Mallet auf keinen Fall lesen soll. Dazu will sie aus Gründen der schnellen Verarbeitung die geheime Botschaft symmetrisch verschlüsseln. Damit Bob jedoch die Nachricht auch entschlüsseln kann, wird der gemeinsam benötigte Schlüssel mit einem asymmetrischen Verfahren vorher ausgetauscht:

> Alice will Bob eine vertrauliche Nachricht schicken und meldet sich hierfür bei Bob an:
> A→ B „Hallo, hier ist Alice!“
>
> Bob antwortet und schickt Alice seinen öffentlichen Schlüssel:
> B→ A „Hier ist Bob!“, bobs-public-key
>
> Alice kann nun den geheimen Schlüssel mit Bob's öffentlichem Schlüssel verschlüsseln:
> A→ B {secret-key} bobs-public-key
>
> Nur Bob kann diese Nachricht mit seinem privaten Schlüssel entschlüsseln und erfährt nun den geheimen Schlüssel für die Übertragung. Jetzt kann Alice ihre vertrauliche Nachricht an Bob senden, verschlüsselt mit dem geheimen Schlüssel:
> A→ B {„Ich liebe Dich!“} secret-key

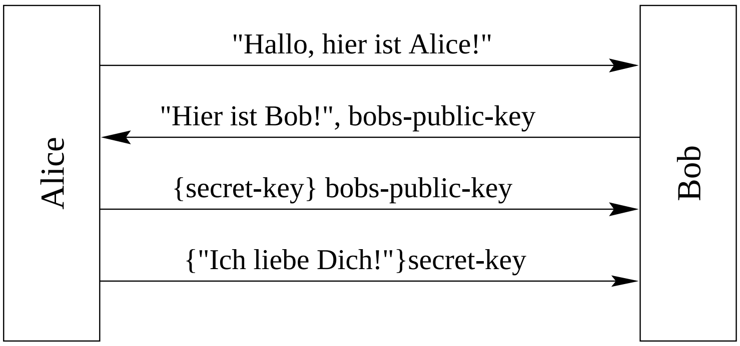

Abbildung 2: Alice und Bob tauschen eine vertrauliche Nachricht aus

5.2 Sicherheitsaspekt: Authentifikation

Während der normalen Mensch-Mensch-Kommuniktion sehen sich oder zumindest hören sich die Kommunikationspartner, um den anderen zu identifizieren. In einer Netzwerkumgebung entfallen in der Regel diese Authentifikationsmöglichkeiten, die Kommunikation findet meist über räumlich getrennte Terminals statt und ist nun nicht mehr auf die zwischenmenschliche Interaktion beschränkt, auch Maschinen kommunizieren jetzt untereinander.

5.2.1 Kommunikationsbeispiel 1

Das Problem der Authentifikation läßt sich wieder am einfachsten an einem Kommunikationsbeispiel darstellen: Alice will Bob wieder eine Nachricht übermitteln. Wie kann sich Alice nun wirklich sicher sein, daß sie auch mit Bob kommuniziert? Leicht könnte der böse Mallet die Nachrichten von Alice abfangen und sich als Bob ausgeben und so die geheime Nachricht lesen:

> Alice will sich erneut bei Bob anmelden, jedoch fängt Mallet die Nachricht ab:
> A→ M „Hallo, hier ist Alice!“
>
> Mallet gibt nun vor Bob zu sein und schickt Alice seinen öffentlichen Schlüssel:
> M→ A „Hier ist Bob!“, mallets-public-key
>
> Alice verschlüsselt jetzt den geheimen Schlüssel mit Mallet's öffentlichen Schlüssel:
> A→ M {secret-key} mallets-public-key
>
> Mallet entschlüsselt den geheimen Schlüssel und kann nun auch die geheime Nachricht von Alice lesen, die immer noch im Glauben ist, mit Bob zu kommunizieren:
> A→ M {„Ich liebe Dich!“} secret-key

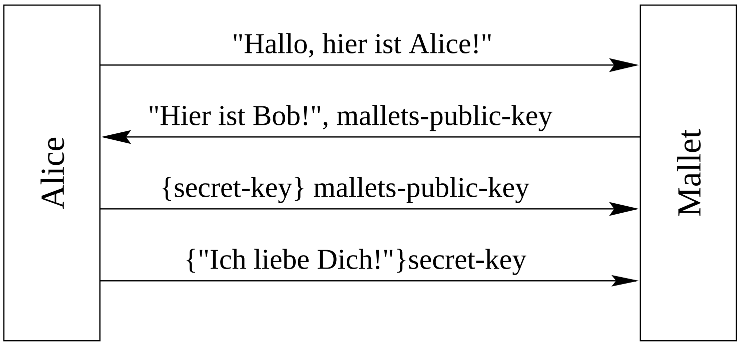

Abbildung 3: Mallet fängt die Daten von Alice ab und gibt sich als Bob aus

5.2.2 Beispiel: X.509 Zertifikate

Um den Kommunikationspartner eindeutig identifizieren zu können, sollte deshalb die asymmetrische Kryptographie mit dem *ISO authentication framework* kombiniert werden. Dieser *framework* ist auch als *X.509 Protokoll* bekannt und wird in [3] beschrieben, während [14] die Anwendung des X.509 Standards beschreibt. Mit X.509 werden Zertifikate definiert, mit denen der Name eines Benutzers an seinen öffentlichen Schlüssel gebunden wird. X.509 Zertifikate in ASN.1-Format, einer Datenrepräsentationssprache [2], haben im wesentlichen folgende Gestalt:

```
certificate ::= SIGNED SEQUENCE (
   signature    AlgorithmIdentifier,
   issuer       Name,
   validity     Validity ::= SEQUENCE (
      notBefore    UTCTime,
      notAfter     UTCTime )
   subject      Name,
   subjectPublicKeyInfo SubjectPublicKeyInfo :: SEQUENCE (
      algorithm          AlgorithmIdentifier,
      subjectPublicKey   BIT STRING ))
```

Hieraus erkennt man, daß zunächst ein Zertifikat aus einer *signierten Sequenz* besteht, die wiederum verschiedene Daten enthält. Auf das Signieren komme ich im weiteren Text noch zu sprechen, zuerst möchte ich die Bedeutung der verschieden Daten erläutern:

- *signature:* ein Wert, der die Identifikationsnummer des Algorithmus' angibt, mit dem die Signatur berechnet wurde.

- *issuer:* der Name der Instanz, die das Zertifikat signiert hat.
- *validity:* der Zeitraum, in dem das Zertifikat gültig ist.
- *subject:* der Name, für den das Zertifikat ausgestellt wurde.
- *subjectPublicKeyInfo:* in *algorithm* steht das gleiche wie in *signature*, ist also redundant, und in *subjectPublicKey* ist der öffentliche Schlüssel von *subject* enthalten.

Damit nicht jeder (Mallet eingeschlossen) seine eigenen Zertifikate erstellen kann, müssen sie mit dem privaten Schlüssel einer sog. *Certificate Authority*, kurz *CA*, signiert sein (siehe zum Signieren auch Kapitel 5.1.3). Diese CA muß eine Instanz sein, der alle Kommunikationsteilnehmer (hier Alice und Bob) vertrauen können. In unserem Beispiel wird also eine CA, der Alice vertraut, Bobs Zertifikat signieren und somit Bobs Namen an seinen öffentlichen Schlüssel binden. Alice erwartet nun nicht mehr im Kommunikationsbeispiel von Abbildung 3 Bobs öffentlichen Schlüssel, sondern sein Zertifikat. Indem sie den öffentlichen Schlüssel der CA auf das Zertifikat anwendet, kann sie überprüfen, ob dieses Zertifikat auch wirklich von der CA signiert wurde. Nachdem sie die Vertrauenswürdigkeit des Zertifikats geprüft hat, wird sie im *subject*-Feld des Zertifikats nachschauen, ob es auch wirklich Bobs Zertifikat ist, und kann dann schließlich Bobs öffentlichen Schlüssel zur weiteren Verwendung entnehmen.

5.2.3 Kommunikationsbeispiel 2

Mit der Benutzung von Zertifikaten sieht unsere Beispielskommunikation jetzt wie folgt aus:

> Auch diesmal fängt Mallet die Nachrichten von Alice ab:
> A$\rightarrow$ M „Hallo, hier ist Alice!“
>
> Will sich Mallet immer noch als Bob ausgeben, muß er, wie in Abbildung 4 zu sehen, Bobs Zertifikat zu Alice senden. Würde er statt dessen immer noch seinen öffentlichen Schlüssel oder sogar sein Zertifikat schicken, würde Alice sofort mißtrauisch werden und die Kommunikation abbrechen.
> M$\rightarrow$ A „Hier ist Bob!“, bobs-certificate
>
> Deswegen überprüft Alice das empfangene Zertifikat und

entnimmt daraus den öffentlichen Schlüssel von Bob und verschlüsselt damit den geheimen Schlüssel, den Mallet nun nicht mehr entschlüsseln kann.
A→ M {secret-key} bobs-public-key

Da Mallet den geheimen Schlüssel nicht entschlüsseln konnte, kann er auch nicht den Inhalt der geheimen Nachricht erfahren:
A→ M {„Ich liebe Dich!“} secret-key

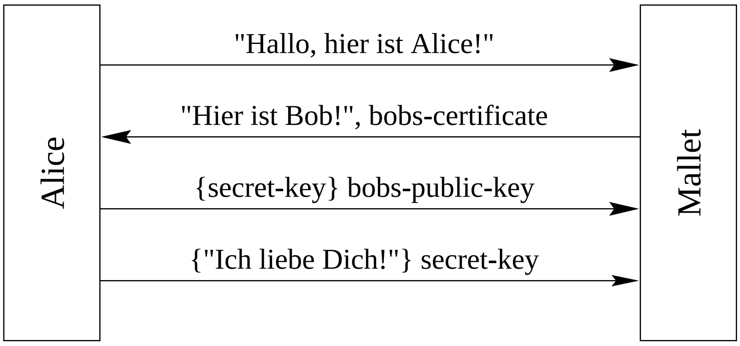

Abbildung 4: Authentifizierung mit Zertifikaten

Übertragen auf Dienste in einem Netzwerk können Zertifikate nicht nur der Authentifizierung sondern auch der Autorisierung dienen, indem einfach nur die Benutzer ein signiertes Zertifikat erhalten, die für einen speziellen Dienst autorisiert sind.

Zur sicheren Kommunikation im Internet stehen inzwischen mehrere kommerzielle, aber auch nichtkommerzielle *Certificate Authorities* zur Verfügung, die unter Vorlage eines Identitätsnachweises ein gültiges Zertifikat erstellen. Anbieter von Diensten im Netz können nun entscheiden, wie weit sie welcher CA vertrauen und welche Zertifikate sie zulassen.

5.3 Sicherheitsaspekt: Datenintegrität

Um Manipulationen an den übertragenen Daten festzustellen, wird von der Nachricht ein sogenannter *Message Authentication Code*, kurz MAC, gebildet. Berechnet wird dieser MAC durch eine Einwegfunktion H, die von einer beliebig langen Nachricht m einen Hash-Wert h von immer gleicher Länge zurückliefert.

Diese Einwegfunktion soll folgende Eigenschaften erfüllen:

- bei gegebenem m soll sich h leicht berechnen lassen
- bei gegebenem h soll sich m schwer berechnen lassen
- bei gegebenem m soll es schwer sein, ein anderes m' zu finden, sodaß $H(m) = H(m')$ ist

Eine Einwegfunktion ist also eine eindeutige (injektive) Funktion, deren Umkehrung sich nur sehr schwer berechnen läßt. Am besten kann man eine solche Funktion mit einem Telefonbuch vergleichen: anhand des Namens einer Person ist es leicht, die dazugehörige Telefonnummer zu finden. Im umgekehrten Fall wird der Aufwand jedoch unverhältnismäßig hoch.

Will man also eine Kommunikation vor Datenmanipulation schützen, berechnet der Absender den MAC der Nachricht und schickt beides auf den Weg. Der Empfänger wendet nun ebenfalls die gleiche Einwegfunktion auf die Nachricht an. Bei unverfälschter Nachricht stimmen beide MAC's überein.

5.3.1 Beispiel: MD5

Als Beispiel für eine Einwegfunktion möchte ich den weitverbreiteten MD5 Algorithmus kurz vorstellen, der in [17] beschrieben wird und die oben genannten Forderungen an eine Hash-Funktion gut erfüllt.

Die Nachricht wird in 512 Bit Blöcken bearbeitet. Wenn nötig, muß das Ende der Nachricht mit Füllbits aufgestockt werden, um sie auf ein Vielfaches von 512 zu bringen. Vom ersten 512 Bits Block wird unter Hinzunahme einer Konstanten in vier Phasen der Hash-Wert berechnet. Für jeden weiteren Block erfolgt die Berechnung analog, nur wird die Konstante durch den Hash-Wert des vorhergehenden Blocks ersetzt. Die angesprochenen vier Phasen führen im wesentlichen verschiedene logische Bitoperationen auf dem

Nachrichtenblock und der Konstanten, bzw. dem Hash-Wert des Vorgängerblocks, aus. Das Ergebnis ist schließlich ein *Message Authentication Code* für die gesamte Nachricht.

Das Konzept des Arbeitsablaufs von MD5 soll folgende Abbildung, entnommen aus [12], verdeutlichen:

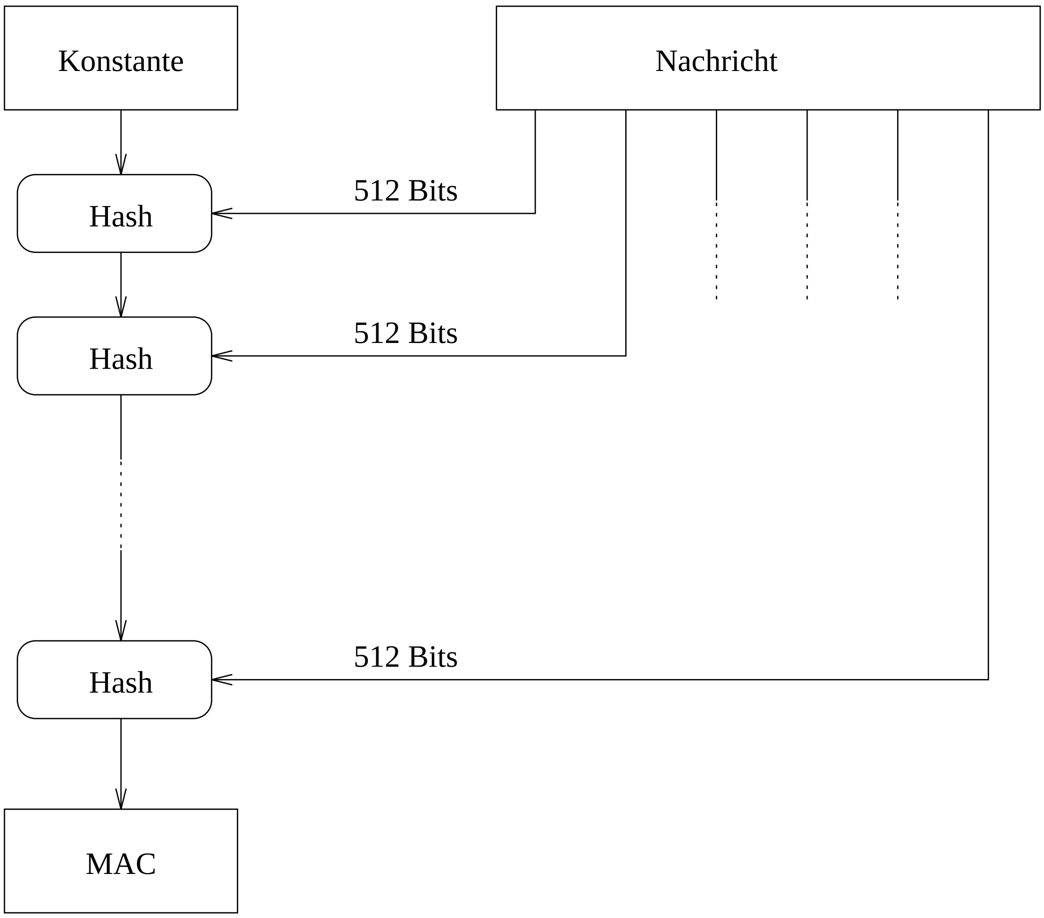

Abbildung 5: MD5 Arbeitsablauf

5.3.2 Kommunikationsbeispiel

Zurück zu unserem Kommunikationsbeispiel: In den Kapiteln 5.1 und 5.2 haben wir jetzt sichergestellt, daß Mallet weder der Kommunikation lauschen, noch sich als Bob ausgeben kann. Das einzige, was ihm bleibt, ist die unbemerkte Manipulation der geheimen Daten. Aus diesem Grund bildet Alice von der Nachricht und dem geheimen Schlüssel einen MAC:

$$\text{MAC} = \text{MD5}(\text{„Ich liebe Dich!“}, \text{secret-key})$$

und schickt diesen zusätzlich zu Bob:

Alice meldet sich bei Bob an:
A→ B „Hallo, hier ist Alice!“

Bob antwortet und schickt Alice sein Zertifikat:
B→ A „Hier ist Bob!“, bobs-certificate

Alice überprüft nun das Zertifikat und entnimmt daraus Bobs öffentlichen Schlüssel, mit dem sie den geheimen Schlüssel verschlüsseln kann:
A→ B {secret-key} bobs-public-key

Alice berechnet, wie oben beschrieben, den MAC und verschlüsselt ihn gemeinsam mit der geheimen Botschaft und schickt diese Nachricht zu Bob:
A→ B {„Ich liebe Dich!“, MAC} secret-key

Bob berechnet nun den MAC wie Alice. Stimmt dieser mit dem empfangenen MAC überein, so weiß Bob, daß die Nachricht nicht beschädigt wurde. Nachdem nun auch die Datenintegrität gewährleistet wurde, erhalten wir jetzt eine Kommunikation, die alle drei Sicherheitsaspekte erfüllen kann. Abbildung 6 stellt diese sichere Kommunikation zwischen Bob und Alice nochmals anschaulich dar.

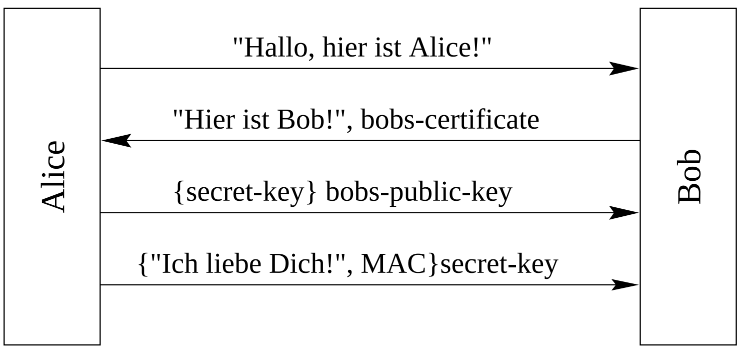

Abbildung 6: Die vertrauliche Nachricht wird mit einem MAC „geschützt“

5.4 Zusammenfassung

Da die drei geschilderten Sicherheitsaspekte den zentralen Punkt innerhalb eines Sicherheitsprotokolls bilden, will ich die wichtigsten Ergebnisse dieses Kapitels kurz zusammenfassen:

- *Vertraulichkeit*
  Die vertraulichen Daten werden mit schnellen symmetrischen Kryptoverfahren, wie DES, verschlüsselt. Der dafür gemeinsam benötigte Schlüssel wird vorher mit einem asymmetrischen Verfahren, z.B. RSA, ausgetauscht.
  Ein RSA-Schlüssel besteht aus einem öffentlichen, für jedermann frei zugänglichen Schlüssel und aus einem privaten, geheimen Teil. Daten, wie der gemeinsam benötigte, geheime DES-Schlüssel, werden mit dem öffentlichen Schlüssel des Empfängers verschlüsselt. Jetzt kann nur der Empfänger mit seinem privaten Schlüssel die Nachricht entschlüsseln.
  Wendet eine Person ihren privaten Schlüssel auf digitale Daten an, so werden diese signiert und jeder kann dies mit ihrem öffentlichen Schlüssel, angewendet auf die signierten Daten, überprüfen.

- *Authentifikation*
  Zur Authentifikation einer Instanz dienen Zertifikate, die den öffentlichen Schlüssel an ihren Namen binden. Zur Herstellung dieser Bindung werden öffentlicher Schlüssel und Name von einer *Certificate Authority* (CA), der alle Beteiligten der Kommunikation vertrauen, signiert.
  Will sich eine Instanz authentifizieren, so schickt sie dem Kommunikationspartner ihr Zertifikat, der daraus Namen und öffentlichen Schlüssel dieser Instanz erfährt. Nun kann der Partner mit ihrem öffentlichen Schlüssel Daten verschlüsseln, die nur diese Instanz mit ihrem privaten Schlüssel entschlüsseln kann.

- *Datenintegrität*
  Um die Datenintegrität überprüfen zu können, gibt es Hash Funktionen wie MD5, die von den Daten einen *Message Authentication Code* (MAC), eine Art Fingerabdruck, berechnen. Diese Hash Funktion ist nur sehr schwer umkehrbar und liefert, unabhängig von der Nachricht, einen MAC von immer gleicher Länge.
  Der Empfänger einer Nachricht bekommt nun zusätzlich auch ihren MAC, den der Sender berechnet hat. Jetzt berechnet der Empfänger selber den MAC und vergleicht beide. Stimmen sie überein, so weiß er, daß die Nachricht nicht beschädigt oder manipuliert wurde.

6 Was ist CODINE?

CODINE ist ein Job-Management-System und steht für **Co**mputing in **Di**stributed **N**etworked **E**nvironments. Es ermöglicht, in einem Netzwerk von verschiedenen UNIX-Workstations die anfallende Rechenlast so zu verteilen, daß die Rechenleistung im Netz möglichst optimal ausgenutzt wird. Dazu werden anfallende Aufträge mittels eines Shell Scripts, das dann auch als Job bezeichnet wird, an das CODINE-System übermittelt. CODINE sorgt nun dafür, daß der Job auf der oder den passenden Maschinen mit der geringsten Auslastung ausgeführt wird. Ziel von CODINE ist es also, die Produktivität der vorhandenen Maschinen zu erhöhen, die im Normalfall die meiste Zeit über ungenutzt bleiben. Weiterführende Informationen zum Konzept und Gebrauch von CODINE können aus dem Handbuch [4] entnommen werden.

6.1 Komponenten von CODINE

Ein komplettes CODINE-System besteht aus mehreren Komponenten, die im Hintergrund als Dämonen auf teilweise verschiedenen Workstations laufen:

- *qmaster*
  Der *qmaster* ist das Herzstück eines CODINE-Systems. Bei ihm laufen alle Informationen zusammen, die benötigt werden, um das ganze System zu steuern.

- *scheduler*
  Der *scheduler* hat die Aufgabe, für jeden Job zu entscheiden, auf welcher Maschine er ausgeführt werden soll. Alle für diese Entscheidung nötigen Informationen erhält er vom *qmaster*. Sinnvollerweise läuft der *scheduler* auf der gleichen Workstation wie der *qmaster*.

- *execd*
  Der *execution daemon* ist zuständig für die Ausführung der Jobs. Hierfür sind Queues auf seinem Host eingerichtet, die die anfallenden Jobs zunächst aufnehmen können. Außerdem teilt er dem *qmaster* in regelmäßigen Abständen die Auslastung des Host und den Zustand seiner Queues mit.

- *shepherd*
  Der *shepherd*, aufgerufen vom *execd*, führt nun endgültig einen Job aus und überwacht diesen bis zur seiner Beendigung.

- *commd*
  Der *communication daemon* stellt eine deadlock-freie, auf Wunsch asynchrone Kommunikation zwischen verschiedenen Hosts innerhalb des CODINE-Systems her. Mehr zum *commd* findet sich in diesem Kapitel unter 6.4.

6.2 Hosts und Benutzergruppen von CODINE

CODINE unterscheidet zwischen verschiedenen Hosts, auf denen nur bestimmte Aktionen ausgeführt werden können, bzw. auf denen bestimmte CODINE-Dämonen laufen:

- *Master Host*
  Auf diesem Host laufen der *qmaster* und meist auch der *scheduler*.

- *Shadow Host*
  Dieser Host übernimmt die Aufgabe des *Master Host*, falls dieser ausfällt.

- *Execution Host*
  Dies ist ein Host, auf dem Jobs von CODINE ausgeführt werden können. Hierfür muß auf ihm der *execd* und mindestens eine Queue eingerichtet sein.

- *Administration Host*
  Auf dem *Administration Host* können alle CODINE-Kommandos ausgeführt und das CODINE-System selbst verwaltet werden. So können hier beispielsweise Queues auf verschiedenen Maschinen hinzugefügt oder gelöscht werden.

- *Submit Host*
  Auf diesen Hosts können Jobs mittels eines Shell Scripts an das CODINE-System übermittelt werden.

Natürlich ist es im CODINE-System auch möglich, daß ein Host mehrere dieser zuvor genannten Aufgaben erfüllt und daß es mehrere Hosts einer Art im Netz geben kann, mit Ausnahme des *Master Host*, da innerhalb eines Systems nur ein *qmaster* aktiv sein kann.

Zusätzlich zu den Hosts unterscheidet CODINE zwischen vier verschiedenen Benutzergruppen, denen unterschiedliche Rechte innerhalb des CODINE-Systems eingeräumt werden:

- *Manager*
  Manager haben die volle Kontrolle über das gesamte CODINE-System.
- *Operator*
  Ein *Operator* hat die gleichen Rechte wie ein *Manager*, außer daß er keine Queues hinzufügen, löschen oder modifizieren darf.
- *Owner*
  Ein *Owner* einer Queue hat das Recht, diese Queue zu sperren oder auch frei zu geben.
- *User*
  Ein *User* hat das Recht, Jobs an das CODINE-System zu übermitteln und diese zu überwachen und zu löschen.

6.3 Arbeitsweise des CODINE-Systems

Am besten läßt sich die Arbeitsweise von Codine erläutern, indem man einen Job von seiner Übermittlung bis zu seiner Ausführung verfolgt (siehe auch Abbildung 7, auf die im Text mit $\boxed{\text{i}}$ Bezug genommen wird):

6.3.1 Job an *qmaster* übermitteln

Zunächst muß der *User* für seine Anweisungen ein Shell Script schreiben. Will er beispielsweise das Programm `program.c` mit einem C-Compiler übersetzen, um das Ergebnis dann auszuführen, könnte das Script `test.sh` folgendermaßen aussehen:

```
echo Dies ist ein Test
cd /home/bieder/SOURCEN/
cc program.c -o testprogram
testprogram
echo Test beendet
```

Hat sich der Benutzer auf einem *Submit Host* eingeloggt, kann er nun von hier aus mit dem folgenden CODINE Kommando das Shell Script an das CODINE-System übermitteln:

```
qsub test.sh
```

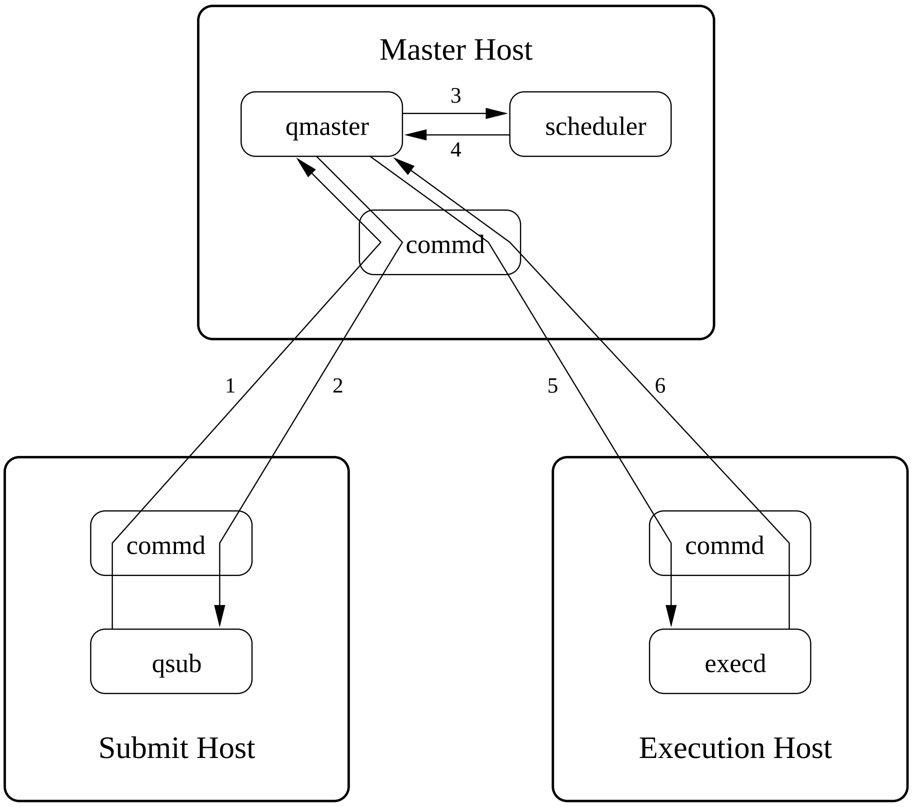

Abbildung 7: Ein Job wird an das CODINE System übermittelt

Das Programm `qsub` sendet das Script mit der Zieladresse des *qmaster* an den *commd*, der im Hintergrund auf dem gleichen Host wie `qsub` läuft. Dieser *commd* reicht das Script an den *commd* vom *Master Host* weiter. Von ihm erhält der *qmaster* schließlich den Job $\boxed{1}$. Auf umgekehrtem Weg bestätigt der *qmaster* den Erhalt des Scripts $\boxed{2}$, `qsub` hat somit seine Aufgabe erfüllt und wird beendet.

6.3.2 Job zuteilen *(schedule)*

Der *qmaster* gibt dem *scheduler* alle Job-Informationen, die Auslastung der *Execution Hosts* und den Statuts ihrer Queues weiter $\boxed{3}$. Der *scheduler* entscheidet anhand der Daten, auf welcher Maschine der Job ausgeführt werden soll und teilt diese Entscheidung dem *qmaster* mit $\boxed{4}$.

6.3.3 Job an *execd* übermitteln

Der *qmaster* übermittelt nun den Job „seinem“ *commd* mitsamt der Adresse des vom *scheduler* erwählten *Execution Host*. Dieser reicht den Job weiter an den *commd* auf dem Zielhost und wird dort dem *execd* zugeteilt $\boxed{5}$. Dieser startet den *shepherd*, der dann die eigentliche Ausführung des Jobs übernimmt. Nach Beendigung des Jobs teilt der *execd* dem *qmaster* über die entsprechenden *commds* mit, daß der Job abgearbeitet wurde $\boxed{6}$.

6.4 Kommunikation von CODINE

Der Kommunikation innerhalb des CODINE-Systems gilt das Hauptaugenmerk dieser Arbeit. Zum Entwurf und zur Implementierung eines Sicherheitsprotokolls muß sie im weiteren genauer untersucht werden.

Im Zentrum der Kommunikation steht der *qmaster*. Er und sein dazugehöriger *commd* sind an jeder Kommunikation innerhalb von CODINE beteiligt. Dies legt nahe, den *qmaster* als eine zentrale Steuereinheit für ein Sicherheitsprotokoll einzurichten. Er bekommt dann zusätzlich die Aufgabe zugeteilt, alle vorhandenen Verbindungen mit ihren sicherheitsrelevanten Daten zu verwalten.

Kennzeichnend für die CODINE-Kommunikation ist das eigenständige Kommunikationssystem, das durch die *commlib* und den *commd* repräsentiert wird. Die *commlib* bildet dabei die Schnittstelle zum *commd*, der dann die eigentliche Kommunikation zwischen den verschiedenen CODINE-Komponenten auf unterschiedlichen Hosts abwickelt. Diese Vorgehensweise bietet verschiedene Vorzüge:

- Die Kommunikation zwischen Kommunikationspartnern kann asynchron erfolgen. Die Teilnehmer übergeben die Nachricht dem *commd* und sind nun nicht mehr bis zur Beendigung der Kommunikation blockiert.

- Deadlocks, die auftreten, wenn zwei CODINE-Komponenten (z.B. qmaster/execd) miteinander kommunizieren wollen, werden nun umgangen.

- Die Kommunikation findet auf einer abstrakten Ebene statt, d.h. es kommen keine Socket-Filedeskriptoren und keine I/O-Systemaufrufe in den Anwendungsprogrammen vor. Dies eröffnet die Möglichkeit, die Kommunikation auf andere Transportschichten aufzusetzen, ohne den CODINE-Kerncode ändern zu müssen.

- Die Einbindung von Komprimier- oder auch Security-Verfahren kann aus Anwendersicht völlig transparent erfolgen.
- Die Problematik des *Host Name Resolvings*, also die Adressierung eines Host im Netzwerk, wird vollständig in das Kommunikationssystem verlegt.

Ein solches Kommunikationssystem und seine besonderen Merkmale müssen beim Entwurf eines Sicherheitsprotokolls natürlich berücksichtigt und integriert werden. Mehr zum Thema *commd* und *commlib* findet sich auch im Artikel [5].

7 Warum Sicherheit für CODINE?

Nachdem nun dargestellt wurde, was eine sichere Kommunikation ausmacht und wie CODINE arbeitet, kann ich jetzt in diesem Abschnitt die Motivation zur eigentlichen Aufgabenstellung beschreiben, warum es notwendig ist, ein Sicherheitsprotokoll für das Job-Management-System CODINE zu entwerfen und zu implementieren:

Das Konzept des Software-Produkts CODINE ist auf eine Anwendung im großen, kommerziellen Rahmen ausgerichtet, in dem man eher erwarten darf, daß vertrauliche und wertvolle Daten zur Übertragung kommen. Man könnte sich Industriespionage vorstellen, bei der jemand anhand der übermittelten CODINE-Jobs feststellen kann, an welchen Projekten eine Firma zur Zeit arbeitet[2]. Methoden zur Verschlüsselung verhindern ein Abhören der Kommunikation und machen sie vertraulich.

Weiterhin eignet sich CODINE sehr gut für einen Anbieter von Rechenleistung. Seine Kunden können ihre Jobs mit Shell Scripts an das CODINE-System übermitteln. Allerdings muß sich dieser Anbieter für eine korrekte Abrechnung sicher sein, daß die Jobs auch von dem Kunden kommen, für den er sich ausgibt. Das bedeutet, daß der Benutzer von CODINE authentifiziert werden muß. Mit Zertifikaten ist eine Authentifizierung sogar im *Wide Area Net* denkbar[3].

Zu der Authentifizierung des Kunden muß nun auch sichergestellt werden, daß niemand in der Lage ist, durch wiederholtes Versenden des gleichen Jobs das Kundenkonto unberechtigt zu belasten. Ein einfacher Zähler, den beide Kommunikationsteilnehmer nach jeder Nachricht inkrementieren, kombiniert mit Verschlüsselungs- und/oder Hash-Algorithmen, kann dies verhindern.

Zuletzt kann ein *Message Authentication Code* von jeder Nachricht dem CODINE-System helfen, schnell und sicher beschädigte oder manipulierte Nachrichten festzustellen, um so die Datenintegrität zu überprüfen.

Bislang gab es innerhalb der CODINE-Kommunikation keine Vertraulichkeit - alle Daten wurden unverschlüsselt übertragen. Auch wurde die Datenintegrität nicht überprüft, beschädigte oder manipulierte Daten wurden nicht erkannt. Authentifiziert wurde der Benutzer, indem der Client seine *effective user ID* (EUID) und seine *effective group ID* (EGID) bestimmt und

[2]Beispielsweise hat der Kunde BMW Interesse an einem „sicheren“ CODINE bekundet.

[3]Es gibt ein EG-Projekt $E = mc^2$, ein Teilprojekt von *TEN-IBC*, das sich mit der Verteilung von Rechenleistung im WAN-Bereich beschäftigt.

sie dem *qmaster* mitgeteilt hat, der dann überprüfen konnte, ob diese ID vorhanden ist. *Access-Listen* haben gewährleistet, daß nur bestimmte Benutzer bestimmte Dienste in Anspruch nehmen können. Wer sich also erfolgreich ins System einloggen konnte, galt als authentifiziert, unter der Annahme, daß alle Hosts im CODINE-Cluster sicher sind. Dazu werden die entsprechenden Hosts beim CODINE-System eingetragen.

Durch die fehlende Vertraulichkeit innerhalb der Kommunikation ist eine solche Benutzer-Authentifikation allerdings mangelhaft. Die Nachrichten könnten abgefangen und die ID durch jede beliebige ersetzt werden. So wird es einem Eindringling durch Benutzung der *root*-ID ermöglicht, die volle Kontrolle über das gesamte System erlangen.

Das CODINE-System selbst wurde „authentifiziert“, dadurch daß es ausschließlich über reservierte Ports (kleiner 1024) kommuniziert, die nur unter *root* eingerichtet werden können.

Aber auch diese Art der Authentifikation bietet keine vollständige Sicherheit und kann nicht verhindern, daß sich jemand mit einem Rechner Zugang zum Netz verschafft, um sich dort beispielsweise als ein *qmaster* auszugeben.

Wie in diesem Kapitel aufgezeigt, besteht durchaus ein Bedarf nach mehr Sicherheit für CODINE. Da jedoch die bisherige Implementation die Sicherheitsaspekte der Vertraulichkeit und der Datenintegrität gar nicht und den der Authentifikation nicht ausreichend berücksichtige, muß ein Sicherheitsprotokoll entworfen und implementiert werden, das diese Aufgaben erfüllt.

8 Das SSL-Protokoll

SSL steht für **S**ecure **S**ocket **L**ayer und aufmerksam auf dieses Sicherheitsprotokoll hat mich Andy Schwierskott von Genias Software GmbH gemacht. In einem Artikel der Zeitschrift c't (siehe [18]) wurde SSL im Zusammenhang mit der Sicherheit im Internet erwähnt. Ziel des SSL-Protokolls ist, daß nach seiner Abarbeitung ein Kanal zur Verfügung steht, auf dem die Kommunikationsteilnehmer sicher Daten austauschen können, unabhängig davon, ob diese Kommunikation im Internet oder nur in einem lokalen Netzwerk statt findet. Da sich SSL als sehr vielversprechend im Hinblick auf meine Diplomarbeit zeigte, habe ich im WWW nach weiteren Hinweisen gesucht und bin schließlich auf die Spezifikation und auf eine Implementation von SSL gestoßen, die ich in den folgenden Abschnitten beschreiben möchte.

8.1 Die Konzeption des SSL-Protokolls

Das SSL-Protokoll in Version 3 wird in [10] spezifiziert. Es stellt ein Sicherheitsprotokoll dar, daß Vertraulichkeit innerhalb eines Computernetzes bietet. Es erlaubt Client/Server-Applikationen eine sichere Kommunikation und bietet die drei schon bekannten Sicherheitsaspekte:

- *Vertraulichkeit:* Nach einem *Handshake* wird ein geheimer Schlüssel zwischen beiden Kommunikationspartnern ausgetauscht, mit dem dann die Nachrichten symmetrisch verschlüsselt werden können (z.B. mit DES).
- *Authentifikation:* Während einer SSL-Kommunikation muß sich der Server immer authentifizieren, beim Client erfolgt dies optional. Zertifikate und asymmetrische Kryptographie (z.B. RSA) helfen, dieses Problem zu lösen.
- *Datenintegrität:* Von jeder Nachricht wird der **M**essage **A**uthentication **C**ode (MAC) gebildet, der beurteilen hilft, ob eine Nachricht beschädigt oder manipuliert wurde. Der MAC wird durch eine Hashfunktion (z.B. MD5) berechnet.

Das Protokoll ist so ausgelegt, daß es applikationsunabhängig einsetzbar ist und von „höheren" Protokollen, wie z.B. HTTP, FTP, TELNET etc., transparent genutzt werden kann. SSL benötigt lediglich ein Transportprotokoll (z.B. TCP) zur Datenübertragung. Bevor jedoch die eigentlichen Daten sicher über das Netz laufen können, muß ein Handshake-Protokoll abgearbeitet werden, dessen Ziel es ist, die Kommunikationspartner zu authentifizieren, die Algorithmen zur Verwirklichung der drei Sicherheitsaspekte festzulegen und einen gemeinsamen geheimen Schlüssel zu vereinbaren. Das Handshake-Protokoll besteht aus folgenden Phasen, die in Abbildung 8 zusammengefaßt werden:

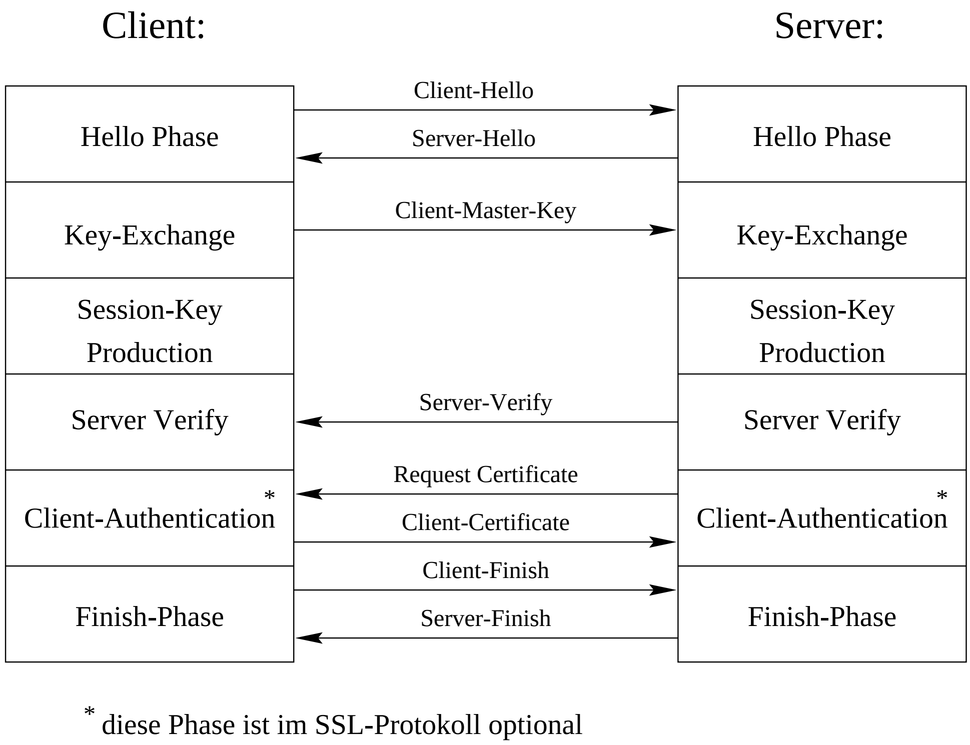

Abbildung 8: Ablauf des SSL-Protokolls

1. **Hello Phase**
   In dieser Phase werden die Algorithmen zum Schlüsselaustausch, zur Verschlüsselung und zur MAC-Berechnung vereinbart.

   - Client-Hello
     Der Client schickt dem Server eine Liste der Algorithmen, die er unterstützt, sowie eine *challenge*, zufällig generierte Bits, die später benötigt werden, um die Authentizität des Servers zu überprüfen.

   - Server-Hello
     Der Server sucht sich aus der Liste der Algorithmen, diejenigen

aus, die er für diese Verbindung unterstützen will, und teilt die Wahl dem Client mit. Zudem schickt er dem Client sein Zertifikat und eine *Connection-ID*, ebenfalls zufällige Bits, die diese Verbindung eindeutig identifiziert.

2. **Key Exchange**
   Der Client generiert den sog. *Master Key*, der dem Server mitgeteilt werden muß.

   - Client-Master-Key
     Der Client extrahiert aus dem Zertifkat vom Server seinen öffentlichen Schlüssel. Der *Master Key* wird nun vom Client zufällig generiert und mit dem öffentlichen Schlüssel vom Server verschlüsselt. Auf diese Weise kann nur der Server mit seinem privaten Schlüssel den *Master Key* entschlüsseln.

3. **Session Key Production**
   Der *Session Key*, der zur eigentlichen symmetrischen Verschlüsselung verwendet wird, wird von beiden Seiten generiert, indem der Hashwert aus *Master Key*, *Connection-ID* und *challenge* mit Hilfe der vereinbarten Hashfunktion berechnet wird. Ab jetzt erfolgt alle weitere Kommunikation zwischen Client und Server verschlüsselt mit dem *Session Key*.

4. **Server Verify**
   Der Server wird vom Client authentifiziert

   - Server-Verify
     Der Server verschickt die *challenge* verschlüsselt mit dem zuvor generierten *Session Key*. Der Client entschlüsselt die Nachricht und vergleicht sie mit der ursprüglichen *challenge*. Stimmen sie überein, so weiß der Client, daß der Server den *Session Key* richtig abgeleitet hat. Das wiederum bedeutet, daß der Server den *Master Key* korrekt dechiffriert haben muß. Dies konnte er allerdings nur, wenn das Zertifkat mit dem darin enthaltenen öffentlichen Schlüssel, das er dem Client geschickt hat, korrekt war. Somit ist der Server authentifiziert.

5. **Client Verify**
   Optional kann der Server vom Client ebenfalls verlangen, sich zu authentifizieren.

- Request-Certificate
  Der Server generiert zufällige *Request*-Bits und schickt sie zum Client.
- Client-Certificate
  Der Client verschlüsselt diese *Request*-Bits mit seinem privaten Schlüssel und schickt sie mit seinem Zertifikat zum Server. Der extrahiert aus dem Zertifikat den öffentlichen Schlüssel vom Client und entschlüsselt damit die empfangenen Request-Bits. Stimmen sie mit den ursprünglichen überein, so ist auch der Client authentifiziert.

6. **Finish Phase**
   Beide Seiten bestätigen, daß sie bereit zur eigentlichen Datenübertragung sind.

8.2 Die Implementierung des SSL-Protokolls

Durch die Implementierung des SSL-Protokolls in den weit verbreiteten *Netscape Navigator*, einen WWW-Browser, ist SSL an die Öffentlichkeit gedrungen und hat gute Chancen, den Status eines Standards zu erlangen. Auch der *Internet Explorer* von Microsoft, der durch eine aggressive Vermarktungsstrategie immer mehr an Marktanteilen gewinnt, unterstützt SSL.

Für den nichtkommerziellen Gebrauch stellt Netscape eine Referenz-Implementation, *SSLref*, frei zur Verfügung, allerdings nur für U.S.-Bürger, wegen der strengen Exportvorschriften der Regierung für Kryptographiesoftware. (Aus diesem Grund wird auch die Schlüssellänge der Export-Version des Netscape Browsers auf 40 Bit beschränkt, eine Schlüssellänge, bei der man auf das Verschlüsseln eigentlich verzichten kann, da jedem Studenten schon die nötige Rechenleistung zur Verfügung steht, um einen solchen Schlüssel in wenigen Stunden durch Ausprobieren aller möglichen Schlüsselkombinationen (*brute force*) zu knacken.)

Um jedoch SSL mit dem vollen Sicherheitsgrad für den Rest der Welt zur Verfügung zu stellen, hat der Australier Eric Young für das Protokoll eine eigene Implementation, *SSLeay*, geschaffen, frei für den kommerziellen und nicht-kommerziellen Gebrauch (siehe dazu auch [23],[24] und [25]).

Im wesentlichen läßt sich diese Implementierung in drei Bereiche einteilen:

1. *libssl.a*
   Diese Bibliothek stellt die eigentliche Implementierung des Netscape

SSL-Protokolls dar. Mit wenigen Änderungen im Source-Code kann man mit Hilfe dieser Bibliothek eine Client-/Server-Anwendung um die Sicherheitsfunktionen von SSL erweitern.

2. *libcrypto.a*
   Bei dieser Bibliothek handelt es sich um eine Sammlung verschiedener Kryptographie-Funktionen, auf die das SSL-Protokoll zurückgreift. Die Bibliothek enthält im einzelnen Funktionen zur

   - symmetrischen Verschlüsselung (DES, IDEA, RC4)
   - asymmetrischen Verschlüsselung (RSA)
   - Hash-Berechnung (MD2, MD5)
   - Zertifikat-Handhabung (X.509)

3. *Applikationen*
   In einem Applikations-Verzeichnis stehen verschiedene Programme zur Generierung und Handhabung von Zertifikaten und RSA-Schlüsseln zur Verfügung.

Zu *SSLeay* ist noch anzumerken, daß sich diese Ausführung des SSL-Protokolls zur Zeit dieser Arbeit noch in der Entwicklungsphase befindet, die allerdings soweit fortgeschritten ist, daß *SSLeay* bereits angewendet werden kann, wie dies beispielsweise beim *Apache-SSL Webserver* [13], bei *SSL-MZtelnet* [15], sowie bei *SSLtelnet* und *SSLftp* [26] geschehen ist. Durch die weltweite Verbreitung im Internet geht zudem die weitere Entwicklung zügig voran, sodaß demnächst eine vollständige, in ihrer Sicherheit umfassend überprüfte Implementation des SSL-Protokolls erwartet werden kann.

8.3 Die Integration des SSL-Protokolls in CODINE

Die Implementierung des SSL-Protokolls bietet eine komfortable Möglichkeit, bestehende Software mit Sicherheitsfunktionen zu erweitern (siehe [24]). Nur wenige Zeilen an C-Code sind zu ändern bzw. hinzuzufügen. Leider ist das SSL-Protokoll rein Socket-orientiert, wie der Name *Secure Socket Layer* erahnen läßt. Die Kommunikation zwischen den CODINE-Komponenten spielt sich jedoch auf einer abstrakteren Ebene ab, wie in Kapitel 6.4 beschrieben.

Theoretisch besteht natürlich die Möglichkeit, das SSL-Protokoll direkt in den *commd* zu integrieren, weil er letztendlich die Kommunikation auch über Sockets herstellt. Dies muß aber nicht unbedingt für die Zukunft gelten und eine Änderung hier würde auch eine Veränderung der in CODINE verwendeten Sicherheitsmechanismen erfordern. Deshalb ist es besser, das Prinzip der Modularität zu wahren und die Security-Ebene zwischen Anwendungsprogramm und *commlib* anzusiedeln. An dieser Stelle stehen jedoch, wie oben beschrieben, keine Sockets zur Verfügung und die Implementierung des SSL-Protokolls ist so nicht anwendbar.

Ein weiterer Nachteil, der gegen das SSL-Protokoll in seiner ursprünglichen Form spricht, ist der hohe Kommunikations-Overhead, der beim SSL-Handshake-Protokoll stattfindet. Legt man ein paar Freiheitsgrade, wie die verwendeten Verschlüsselungsalgorithmen, fest, so läßt sich der nötige Handshake drastisch reduzieren und somit die Performance in den ohnehin stark ausgelasteten Netzen steigern.

Aus diesen Gründen habe ich für CODINE auf den direkten Einsatz des SSL-Protokolls verzichtet und habe ein neues Protokoll, angelehnt an SSL, entworfen, das ich im folgenden Kapitel vorstellen will.

9 Die Konzeption von CSP

Wie im verangehenden Kapitel 8.3 erläutert, eignet sich das SSL-Protokoll nicht unbedingt für einen Einsatz im CODINE System. Daher habe ich für CODINE ein maßgeschneidertes, an SSL angelehntes Protokoll entworfen und implementiert, das möglichst effizient und dennoch sicher arbeitet und das die Kommunikationsmechnismen von CODINE, nämlich den *commd* inklusive der *commlib*, unterstützt. Außerdem zeichnet es sich durch einen wesentlich kürzeren Handshake aus, der sogar unter Umständen ganz wegfallen kann. Das neue Sicherheitsprotokoll, das in diesem Kapitel vorgestellt wird, soll natürlich auch die drei Sicherheitsaspekte Vertraulichkeit, Authentifikation und Datenintegrität von SSL unterstützen.

Kennzeichnend für das SSL-Protokoll ist, daß es viele Freiheitsgrade zuläßt, die erst während einem langen Handshake-Protokoll zwischen Client und Server festgelegt werden. In weltweiten Netzwerken, bei denen eine Kommunikation zwischen den verschiedensten Anwendungen und beliebigen Staaten mit ihren spezifischen Kryptographiebestimmungen denkbar ist, ist ein solches Vorgehen sinnvoll, wenn nicht sogar zwingend, denn gewisse Verschlüsselungsalgorithmen sind nicht in allen Ländern verfügbar oder unterliegen anderen Lizenz- und Rechtsbestimmungen.

Die Kommunikation bei CODINE ist allerdings nur auf das eigentliche System beschränkt, sodaß die Freiheitsgrade für ein *CODINE Security Protocol*, kurz *CSP*, leicht wie folgt festgelegt werden können:

- *symmetrische Verschlüsselung:* DES im CBC-Modus
- *Schlüsselaustausch:* RSA
- *Hashwert-Berechnung:* MD5
- *Client Authentifikation:* immer

Auf diese Art und Weise läßt sich die Anzahl der Nachrichten während des Handshakes gegenüber dem SSL-Protokoll von acht auf zwei reduzieren, was einen spürbaren Gewinn an Ausführungszeit und Netzentlastung mit sich bringt.

9.1 Erklärung einiger Begriffe

Bevor ich den Ablauf des CSP erläutere, möchte ich einige Begriffe erklären, die im weiteren Verlauf der Beschreibung Verwendung finden:

Server Wenn ich im weiteren Verlauf der Ausarbeitung den Terminus *Server* verwende, meine ich im Zusammenhang von CODINE den *qmaster*, die zentrale Steuereinheit des Systems.

Client Unter *Clients* verstehe ich im folgenden alle Komponenten, die mit dem *qmaster* über das Netz hinweg kommunizieren wollen, beispielsweise *execd, qsub,* etc.

Sequencenumber Nach jeder erfolgten Kommunikation zwischen Client und Server inkrementieren beide Seiten eine sog. *Sequencenumber.* Sie verhindert, daß Nachrichten mehrfach unberechtigt versendet werden können. Genau genommen besteht die *Sequencenumber* aus zwei Zählern, einer für empfangene, der andere für gesendete Nachrichten, wobei folgendes bei einer ungestörten Kommunikation immer gilt:

Receive-Sequencenumber(Client) = Send-Sequencenumber(Server)
Receive-Sequencenumber(Server) = Send-Sequencenumber(Client)

Connection ID Jeder Client muß mit dem Server zuerst eine Verbindung aufbauen, unabhängig davon, wie oft sie später miteinander kommunizieren und ob die Kommunikation verschlüsselt wird oder nicht. Jede Verbindung zwischen einem Client und dem Server erhält eine eindeutige *Connection ID*, damit der Server später wieder diese Verbindung anhand der Nummer referenzieren kann.

Connection List Der Server, als zentrale Steuereinheit, verwaltet eine *Connection List*, die die Daten jeder Verbindung speichert. Dazu gehört die Connection ID, die Zieladresse des Clients (host, commprog, id), das „Verfallsdatum“ für den Listeneintrag, die beiden Sequencenumbers und das Schlüsselmaterial, das aus acht *unsigned longs* besteht. Will ein Client mit dem Server kommunizieren, dient als Zugriffsindex auf die *Connection List* die vom Client angegebene Connection ID. So kann der Server alle verbindungsorientierten Daten aus der Liste entnehmen, ohne daß sich der Client erneut anmelden muß. Findet die Kommunikation in der anderen Richtung statt, also vom Server zum Client, referenziert der Server die benötigen Daten anhand der Zieladresse des Clients, die ebenfalls wie die Connection ID eindeutig ist. Auf diese Art und Weise wird dem Server eine asynchrone Kommunikation zwischen verschiedenen Clients ermöglicht.

9.2 Ablauf von CSP

Die Abbildung 9 zeigt den Ablauf von CSP schematisiert. Der Text der linken Spalte bezeichnet die einzelnen Phasen, die während der Abarbeitung des Protokolls durchlaufen werden. Unter den Spalten *Client* und *Server* stehen die einzelnen Aktionen, die auf der entsprechenden Seite ausgeführt werden. Pfeile signalisieren eine Nachrichtenübertragung und deren Richtung, wobei der Text den Inhalt der Nachricht näher bestimmt. Die gestrichelten Linien in der Server-Spalte deuten an, daß der Server zu diesem Zeitpunkt des Protokolls noch mit beliebig anderen Clients kommunizieren kann.

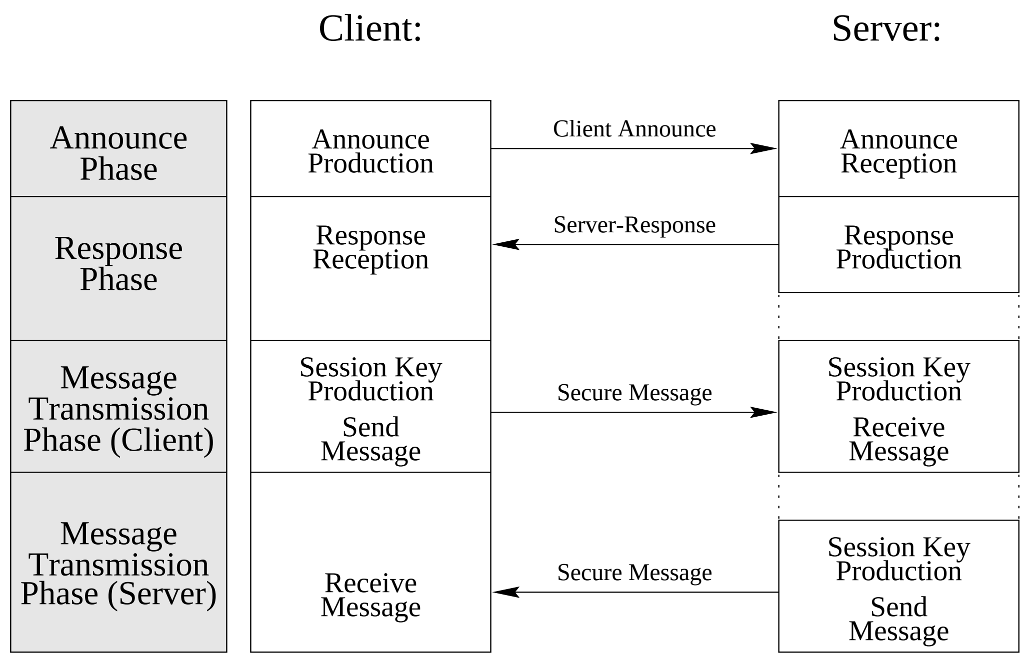

Abbildung 9: Das *CODINE Security Protocol* CSP

Dadurch, daß das Protokoll sehr wenige Phasen durchläuft und dennoch den gleichen Sicherheitsstandard wie das SSL-Protokoll bietet, werden in den einzelnen Phasen von CSP weit mehr Aktionen durchgeführt als bei SSL:

1. **Announce Phase**
   Jeder Client, der mit dem Server kommunizieren will, muß sich zuvor beim Server anmelden (siehe Abbildung 10). Zu diesem Zweck schickt der Client dem Server eine Nachricht, die das Zertifikat des Clients, die Länge des Zertifikats und eine *Challenge* für den Server enthält. Diese

Challenge besteht aus Zufallsbits, die vom Client generiert werden und wird benötigt, um den Server später zu authentifizieren.

Abbildung 10: Client Announce

Der Server überprüft das empfangene Zertifikat auf seine Gültigkeit, wobei ein Zertifikat als gültig gilt, wenn es von der richtigen CA signiert wurde und das Gültigkeitsdatum noch nicht abgelaufen ist. Danach kann er aus dem Zertifikat den öffentlichen Schlüssel des Clients extrahieren. Tritt in dieser Phase beim Server ein Fehler auf, wird das Protokoll mit einer Fehlermeldung zum Client abgebrochen.

2. **Response Phase**
Der Server antwortet auf die Anmeldung vom Client. Dazu muß er zunächst das Schlüsselmaterial, aus dem später der symmetrische Schlüssel abgeleitet wird, zufällig generieren. Damit nur der Client das Schlüsselmaterial erfährt, verschlüsselt der Server es mit dem öffentlichen Schlüssel vom Client. Desweiteren signiert der Server mit seinem privaten Schlüssel den Hashwert der Challenge, um dem Client seine Authentizität zu beweisen. Würde der Server anstatt des Hashwertes die Challenge selbst verschlüsseln, so würde er etwas signieren, was er nicht kennt. Zudem wäre der Client jetzt auch in der Lage, aus der verschlüsselten Challenge den privaten Schlüssel vom Server abzuleiten, was unter keinen Umständen geschehen darf (siehe dazu auch [20] Seite 286).

   Nun kann der Server dem Client eine Antwort schicken, die das Zertifikat des Servers, die Länge des Zertifikats, die verschlüsselte Challenge, die Länge des Verschlüsselungsergebnisses, das verschlüsselte Schlüsselmaterial, die Länge dieser Verschlüsselung und die nächste freie Connection ID enthält. Hierfür besitzt der Server einen Connection-ID-Zähler, der nach jeder Anmeldung inkrementiert wird. Die letzte Aktion des Servers während der *Response Phase* ist das Eintragen der Daten dieser Verbindung in die Connection List. Abbildung 11 listet nochmals die übertragenen Daten im einzelnen auf.

   Der Client überprüft nach dem Empfang der *Response* ebenfalls das

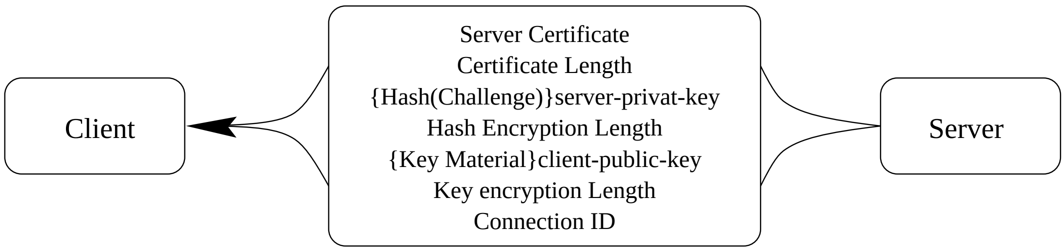

Abbildung 11: Server Response

Zertifikat vom Server und liest aus ihm seinen öffentlichen Schlüssel. Nun entschlüsselt der Client den verschlüsselten Hashwert von der Challenge, bildet auch den Hashwert von der ursprünglichen, von ihm generierten Challenge und vergleicht beide Werte miteinander. Stimmen sie überein, dann hat sich der Server authentifiziert, da nur er seinen persönlichen, privaten Schlüssel kennt (siehe dazu auch 5.1.3, Signieren). Danach entschlüsselt der Client mit seinem privaten Schlüssel das vom Server generierte Schlüsselmaterial und der Client ist jetzt bereit, seine eigentliche Nachricht sicher zu senden.

3. **Message Transmission Phase (Client)**
Diese Phase kann nach der Anmeldung beliebig oft[4] wiederholt werden und dient primär der Nachrichtenübertragung. Allerdings wird hier noch nebenbei der Client authentifiziert. Dazu bildet er von der zu sendenden Nachricht, dem Schlüsselmaterial und der Send-Sequencenumber einen MAC:

```
MAC = MD5(Message, Key Material, Send-Sequencenumber)
```

Dann leitet er vom Schlüsselmaterial den Session Key ab und verschlüsselt damit Nachricht und MAC und schickt beides mit der Connection ID zum Server, nachdem er seine Send-Sequencenumber um eins erhöht hat.

Der Server sucht jetzt anhand der empfangenen Connection ID in der Connection List nach dem entsprechenden Schlüsselmaterial zu dieser Verbindung. Von dem Schlüsselmaterial kann er ebenfalls den Session Key ableiten, mit dem er den MAC und die Nachricht korrekt entschlüsselt. Nun bildet der Server den gleichen MAC von der empfangenen Nachricht, von seiner Receive-Sequencenumber und von seinem

[4]solange die Verbindungsdaten gültig sind. Danach muß eine erneute Anmeldung seitens des Clients folgen.

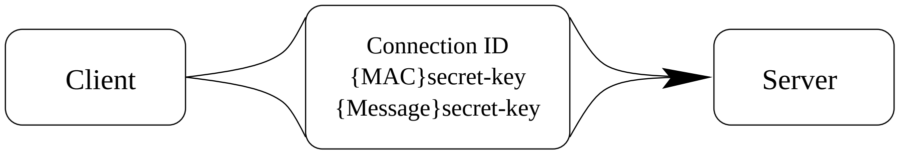

Abbildung 12: Secure Message (Client → Server)

Schlüsselmaterial. Stimmen die beiden MACs nicht überein, kann das verschiedene Gründe haben:

- *unterschiedliches Schlüsselmaterial bei Client und Server*
  In diesem Fall muß der Server davon ausgehen, daß der Client das Schlüsselmaterial nicht korrekt mit seinem öffentlichen Schlüssel entschlüsseln konnte. Somit ist der Client nicht authentifiziert.
- *die Nachricht wurde mutwillig manipuliert*
  Jemand hat sich Zugang zum Netz verschafft und hat Daten der Nachricht verändert.
- *die Nachricht wurde beim Transport beschädigt*
  Bei der Übertragung durch das Netz sind Daten der Nachricht unbeabsichtigt verändert worden. Die ursprüngliche Nachricht ist nicht mehr erhalten.
- *Verlust einer vorangehenden Nachricht dieser Verbindung*
  Ist eine Nachricht verloren gegangen, stimmen die Sequencenumbers auf beiden Seiten nicht mehr überein. Eine solche Störung der Kommunikation kann sowohl mutwillig, als auch unbeabsichtigt erfolgen.

Egal welche Gründe nun vorliegen, der Server muß diese Nachricht unbearbeitet ablehnen und der Client muß das Protokoll erneut, von der Anmeldung her, abarbeiten. Stimmen jedoch die MACs überein, kann der Server die Nachricht auswerten. Dann erhöht auch der Server die Receive-Sequencenumber und schreibt sie mit der Zieladresse des Clients an entsprechender Stelle in die Connection List zurück.

4. **Message Transmission Phase (Server)**
   Wie der Client, kann umgekehrt auch der Server sichere Nachrichten an verschiedene Clients schicken, die sich zuvor bei ihm angemeldet haben. Dazu sucht der Server anhand der Zieladresse (host, commprog, id) die verbindungsorientierten Daten aus der Connection List, bildet

aus dem Schlüsselmaterial den geheimen Schlüssel und den MAC und verschlüsselt anschließend beides.

Wie man in Abbildung 13 sieht, versendet auch der Server eine Connection ID, obwohl der Client keine Connection List besitzt, da er nur mit einem Server kommuniziert. Dies erfolgt dennoch, um die gleichen *send_message*- und *receive_message*-Funktionen bei Server und Client verwenden zu können.

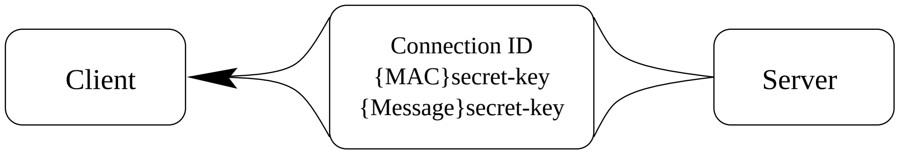

Abbildung 13: Secure Message (Server → Client)

Nachdem nun der Server die Nachricht an seinen *commd* weitergeleitet hat, inkrementiert er seine Send-Sequencenumber und schreibt den aktuellen Wert in die Connection List zurück.

Der Client entschlüsselt nun Nachricht und MAC und bildet, wie gehabt, seinerseits einen MAC und vergleicht beide. Danach inkrementiert er seine Receive-Sequencenumber und beendet diese Phase.

10 Die Implementierung von CSP

Die Implementierung von CSP wird von einer Reihe von Faktoren bestimmt, die ich im folgenden Abschnitt beschreiben werde. Zuletzt möchte ich die einzelnen Bestandteile von CSP erläutern, wie sie aufgebaut und wie sie zu benutzen sind.

10.1 Beeinflussende Faktoren

Hauptfaktor der Gestaltung der Ausführung von CSP war natürlich der Aufbau von CODINE, in den sich CSP nahtlos einfügen sollte. Daher war es eine Zielsetzung dieser Diplomarbeit, die Methoden zur Sicherung der Kommunikation von CODINE in Form einer *Library* zu realisieren, um den bestehenden CODINE-Code nur um wenige Bibliotheksaufrufe und Ergänzungen erweitern zu müssen. Zudem sollten die Vorgänge der Sicherheitsmechanismen weitgehend transparent für den Anwender des CODINE-Systems ablaufen, was ebenfalls die Struktur der Implementierung weitgehend beeinflußte.

Das Einsatzgebiet von CODINE in heterogenen UNIX-Netzwerken bestimmte auch die Programmierwerkzeuge dieser Arbeit: Als Hardware kamen bei der Implementierung Workstations verschiedener Hersteller mit ihren verschiedenen UNIX-Derivaten zum Einsatz[5]. Und wegen der weiten Verbreitung von C-Compilern auf diesen Plattformen wurde, wie für CODINE, die Programmiersprache C gewählt.

Ein weiterer wichtiger Faktor, der die Implementierung von CSP beeinflußte, war die Anlehnung von CSP an SSL, was nahe legt, die verwendeten Kryptographie-Algorithmen von *SSLeay* auch für CSP zu verwenden. Dies bringt zum einen eine Ersparnis von viel Programmierarbeit mit sich, die wohl auch den Rahmen einer Diplomarbeit gesprengt hätte, und zum anderen ein Plus an Sicherheit, weil die Algorithmen aus der Implementierung von SSL Monate, zum Teil Jahre, im Internet getestet, korrigiert und optimiert wurden.

10.2 Die Bestandteile von CSP

In diesem Abschnitt will ich einen Überblick der Bestandteile von CSP vermitteln, ihren Aufbau, ihre Funktion und ihre Arbeitsweise. Dies soll vor

[5]DEC mit DEC OSF, HP mit HP UX, IBM mit AIX, PC Klone mit Linux, Silicon Graphics mit IRIX und SUN mit SunOS

allem dem Programmierer und Anwender helfen, dieses Protokoll richtig zu gebrauchen. Detailliertere Informationen über Funktionsweise und Schnittstelle der einzelnen Funktionen können aus dem Quell-Code im Anhang entnommen werden.

Im wesentlichen setzt sich die Implementierung von CSP aus den folgenden vier Teilen zusammen:

1. *sec_local.h*
   Der Kern diese Header Datei bildet die Definition und Deklaration der *gsd*-Struktur, *gsd* für *global secure data*, die die globalen Variablen, auf die CSP zugreift, beinhaltet. Die Entsprechung hierfür in *SSLeay*, der Implementation von SSL, ist die sogenannte *SSL*-Struktur, die alle verbindungsbezogenen Daten aufnimmt. Zudem enthält *sec_local.h* noch viele wichtige Makros und weitere Definitionen.

2. *libsec.a*
   Bei dieser Bibliothek handelt es sich um die Implementierung von CSP und stellt einige Funktionen zur Sicherung der Kommunikation von CODINE zur Verfügung.

3. *libcrypto.a*
   Diese Bibliothek, kurz vorgestellt in Kapitel 8.2, ist ein Bestandteil von *SSLeay* und wurde aus der Version 0.4.5d entnommen. Sie bildet eine Sammlung vieler Kryptographie-Methoden, auf die CSP zurückgreift.

4. *Applikationen*
   Wie SSL hat auch CSP Applikationen, um RSA-Schlüssel und Zertifikate zu generieren und zu handhaben. Jedoch sind diese Applikationen direkt auf die Bedürfnisse von CODINE zugeschnitten und wesentlich einfacher zu bedienen als die jeweilige Entsprechung von *SSLeay*.

10.2.1 Die Header Files

Die meisten *Header*-Dateien aus der Implementierung von CSP enthalten nur Funktionsprototypen für ANSI-C-Compiler. Ihre Namen sind gleichlautend zu den entsprechenden C-Programmen und bedürfen keiner weiteren Erläuterung. Allerdings gibt es noch einige Header Files, deren Funktion darüber hinausgeht und deren Aufgaben im weiteren genauer beschrieben werden.

`crypto.h`

Dieses Header File inkludiert verschiedene Header Files aus *SSLeay*, um dessen Funktionen aus der `libcrypto.a` nutzen zu können. Nicht alle dort aufgeführten Funktionsdeklarationen entsprechen dem ANSI-C-Standard und es kommt zu entsprechenden *Warnings* beim Übersetzen, die allerdings ignoriert werden können (Stand *SSLeay* Version 0.4.5d).

`sec_lib.h`

`sec_lib.h` muß immer dann eingebunden werden, wenn Funktionen aus der CSP-Bibliothek benutzt werden sollen. Dies erfolgt durch:

```
#ifdef SECURE
#include "sec_lib.h"
#endif
```

`sec_subst.h`

Dieses Header File ersetzt alle Vorkommen von `send_message` und `receive_message` aus der `libcomm.a` durch die Entsprechungen `sec_send_message` und `sec_receive_message` aus der `libsec.a`.

`apilib/cod_secL.h`

Die Struktur der *Connection List*, einer CULL-Liste, die die sicherheitsrelevanten Daten aller Verbindungen speichert, wird in diesem Header definiert.

`sec_local.h`

`sec_local.h` bildet den zentralen Punkt der Implementierung von CSP. Sie enthält wichtige Definitionen und Deklarationen von globalen Variablen, die CSP benötigt. Nur Funktionen der `libsec.a` Bibliothek sollte der Zugriff auf diese Variablen gewährt werden, um mit `libsec.a` eine klar definierte Schnittstelle zwischen der CODINE-Anwendung und den Sicherheitsmechanismen von CSP zu erhalten.

```
#ifndef __SEC_LOCAL_H
#define __SEC_LOCAL_H

#ifndef __CRYPTO_H
#include "crypto.h"
#endif
```

Die Funktionsprototypen der *libcrypto.a* werden eingebunden.

```
#ifndef __CULL_H
#include "../../cull/src/cull.h"
#endif
```

Definitionen der *CULL* (**C**ommon **U**sable **L**ist **L**ibrary), beschrieben in [1], werden eingebunden, auf der die *Connection List* beruht.

```
#include "cod_all_listsL.h"     /* connection list      */
```

Die *Connection List* u.a. wird bekannt gemacht.

```
#define CHALL_LEN       16
```

Definition der Länge der Challenge, die der Client zur Anmeldung generiert, um den Server zu authentifizieren.

```
#define ValidHours      12      /* expiry of connection */
```

Definition der Anzahl der Stunden, für die die Daten einer Verbindung in der *Connection List* gültig sind.

```
#define ClearHours      1
```

Nach *ClearHours* Stunden wird die *Connection List* von alten Verbindungen gereinigt.

```
#define KeyPath         "KEY"
#define CertPath        "CERT"
#define CodSecPath      "COD_SECURE"
```

Pfad-Definitionen für RSA-Schlüssel und Zertifikate.

```
#define CaKey           "ca_key.pem"
#define CaText          "ca_cert.txt"
#define CaTextDefault   "ca_def.txt"
#define CaCert          "ca_cert.pem"

#define RsaKey          "rsakey.pem"
#define CertText        "cert.txt"
#define CertTextDefault "default.txt"
#define CertReq         "request.pem"
#define Cert            "cert.pem"

#define ReconnectFile   "reconnect.dat"
```

Datei-Namen für Schlüssel, Zertifikate etc. werden definiert. Genauere Informationen über Verzeichnisse und Pfade befinden sich im Abschnitt 10.2.4 auf Seite 61.

```
#define DES_192_EDE3_CBC_WITH_MD5      0
#define NULL_WITH_MD5                  1  /* not implemented yet  */
#define RC4_128_WITH_MD5               2  /* not implemented yet  */
#define IDEA_128_CBC_WITH_MD5          3  /* not implemented yet  */
#define DES_64_CBC_WITH_MD5            4
```

Definition der Kombinationen von verschiedenen Verschlüsselungsmechanismen.

```
#define CIPHER_MODE                    DES_192_EDE3_CBC_WITH_MD5
```

Die verwendeten Kryptomechanismen werden definiert. In diesem Fall ist es DES-Verschlüsselung mit einem 192 Bit-Schlüssel im EDE / CBC Modus (-**e**ncrypted -**d**ecrypted -**e**ncrypted / **c**ipher-**b**lock -**c**haining) und eine Hashwertberechnung mit dem MD5 Algorithmus.

```
#define INC32(a)        (((a) == 0xffffffffL)? 0:(a)+1)
```

Dieses Makro inkrementiert eine 32 Bit-Zahl a, bzw. setzt sie auf 0, falls alle Bits dieser Zahl gesetzt sind.

```
typedef struct path_str
{
      char    *ca_key_file;
      char    *ca_text_file;
      char    *ca_textdefault_file;
      char    *ca_cert_file;
      char    *key_file;
      char    *text_file;
      char    *textdefault_file;
      char    *req_file;
      char    *cert_file;
      char    *reconnect_file;
} FILES;
```

Deklaration einer Struktur, die die Pfadinformationen der Schlüssel- und Zertifikatdateien aufnimmt.

```
typedef struct gsd_str
{
      int     crypt_space;
```

Nun beginnt die Global-Secure-Data (gsd) Struktur. Die erste Variable hält die Anzahl der Bits fest, die zusätzlich zu einer verschlüsselten Nachricht als Information gebraucht werden.

```
      int     block_len;
```

Die Anzahl der Bits, die der symmetrische Verschlüsselungsalgorithmus an einem Stück chiffriert. Der verschlüsselte Text ist immer ein Vielfaches dieser Zahl, bei DES beträgt die Blocklänge 8 Bits.

```
      u_char  *key_mat;
      int     key_mat_len;
      char    *keys;
      int     keys_len;
```

Diese Variablen speichern der Reihe nach das Rohmaterial für den symmetrischen Schlüssel, die Länge dieses Rohmaterials, den fertig generierten Schlüssel und seine Länge in Bits.

```
RSA     *rsa;
```

Diese Struktur enthält den öffentlichen und privaten, also asymmetrischen RSA-Schlüssel.

```
X509    *x509;
int     x509_len;
```

Die erste Variable, eine Struktur, definiert in der *libcrypto.a* wie auch RSA, speichert das Zertifikat, die zweite seine Länge.

```
void    (*crypt_init)(u_char *key_material);
```

Ein Zeiger auf eine Funktion, die aus dem Schlüsselmaterial den eigentlichen symmetrischen Schlüssel generiert.

```
u_long  (*crypt)(u_long len, u_char *from, u_char *to,
                 int encrypt);
```

Ein Zeiger auf eine Funktion, die einen Puffer *from* der Länge *len* symmetrisch chiffriert und nach *to* ablegt. Der Wert von *encrypt* entscheidet, ob ver- oder entschlüsselt werden soll.

```
u_long  connid;
```

Die *Connection ID*, der aktuellen Verbindung.

```
u_long  connid_counter;
```

Der Server braucht eine Variable, die die nächste freie *Connection ID* bereit hält.

```
int     iscodsys;
```

Eine Variable, die anzeigt, ob das Programm ein CODINE-Daemon ist oder nicht. Zu den CODINE-Daemons zählen der *qmaster*, der *schedd* und der *execd*.

```
int connect;
```

Hat diese Variable den Wert '1', dann ist eine Anmeldung beim *qmaster* bereits erfolgt oder die verbindungsorientierten Daten konnten aus dem *Reconnect-File* gelesen werden.

```
        lList   *conn_list;
```

Für den Server ein Zeiger auf seine *Connection List*, die alle zuvor behandelten Verbindungen speichert.

```
        u_long  seq_send;
```

Ein Zähler für alle gesendeten Nachrichten der aktuellen Verbindung.

```
        u_long  seq_receive;
```

Ein Zähler für alle empfangenen Nachrichten.

```
        char    *refresh_time;
```

Zeit bis die Verbindung ihre Gültigkeit verliert.

```
        FILES   *files;
  } GlobalSecureData;
```

Mit *files* wird die *gsd* abgeschlossen und enthält Dateinamen für Schlüssel und Zertifikat.

```
  typedef struct des_ede3_cbc_state_st
  {
        des_key_schedule        k1;
        des_key_schedule        k2;
        des_key_schedule        k3;
        des_cblock              iv;
  } DES_EDE3_CBC_STATE;
```

Die Schlüsselstruktur von DES im EDE-CBC-Modus.

```
typedef struct des_cbc_state_st
{
      des_key_schedule        k1;
      des_cblock              iv;
} DES_CBC_STATE;
```

Die Schlüsselstruktur von DES im CBC-Modus.

```
#define c4TOl(c,l)     (l =((unsigned long)(*((c)++)))<<24, \
                        l|=((unsigned long)(*((c)++)))<<16, \
                        l|=((unsigned long)(*((c)++)))<< 8, \
                        l|=((unsigned long)(*((c)++))))

#define lTO4c(l,c)     (*((c)++)=(unsigned char)(((l)>>24)&0xff), \
                        *((c)++)=(unsigned char)(((l)>>16)&0xff), \
                        *((c)++)=(unsigned char)(((l)>> 8)&0xff), \
                        *((c)++)=(unsigned char)(((l)    )&0xff))
```

Zwei Makros, die die Bits von vier *Characters* in die ersten 32 Bit eines *Long* umwandeln und umgekehrt und werden nur benötigt, um das Schlüsselmaterial aus der *Connection List*, einer CULL Liste, zu lesen bzw. zu schreiben.

```
#ifdef SEC_MAIN
GlobalSecureData              gsd;
#else
extern GlobalSecureData       gsd;
#endif /* SEC_MAIN      */
```

gsd wird in SEC_MAIN definiert, andernfalls deklariert.

10.2.2 Die Bibliothek *libsec.a*

Die *libsec.a* bildet die Schnittstelle zwischen der *global secure data* und der CODINE-Anwendung. Diese Bibliothek ist so gestaltet, daß das CODINE-System leicht um die Sicherheitsmechanismen erweitert werden kann, ohne den Source Code umfangreich ändern zu müssen. Solche Änderungen sollten aber wie folgt in die Präprozessor-Anweisungen eingebettet sein:

```
#ifdef SECURE
...
#endif
```

Um jedoch *libsec.a* zu nutzen, muß zuvor die Header Datei `sec_lib.h`, die die Funktionsprototypen zur Verfügung stellt, eingebunden werden. Im Folgenden werden die wichtigsten Funktionen aus dieser Bibliothek aufgeführt und erläutert:

`int sec_init(char *progname)`

Diese Funktion initialisiert die Werte der *gsd* Struktur und muß immer beim Beginn von Client und Server aufgerufen werden. Zur Zeit erfolgt dies in der `do_enroll`-Funktion.

progname : Programmname des aufrufenden Programms.

$$return\_value = \begin{cases} 0 & : \text{ ok} \\ -1 & : \text{ Fehler} \end{cases}$$

Neben der Initialisierung verschiedener Werte werden folgende Aktionen ausgeführt:

- Zufallszahlengenerator initialisieren
- Pfade für Dateien setzen
- eigenen Schlüssel und eigenes Zertifikat von Datei lesen

`int sec_send_message(int synchron,char *tocomproc,int toid,char *tohost,int tag,char *buffer,int buflen,u_long mid)`

Wird CODINE mit CSP genutzt, müssen alle Vorkommen im Code von `send_message` durch diese Funktion ersetzt werden. Sie bildet eine Schale um die alte Send-Funktion aus der `libcomm.a`, die über die gleiche Schnittstelle verfügt. In ihr werden neben dem *send* verschiedene sicherheitsrelevante Aktionen durchgeführt.

synchron	: Übertragungsmodus
tocomproc	: Zielprogramm
toid	: ID der Verbindung
tohost	: Zielhost

tag	: Nachrichtentyp
buffer	: enthält die zu sendende Nachricht
buflen	: Länge dieser Nachricht
mid	: ID für die asynchrone Übertragung

$$return\_value = \begin{cases} 0 & : \text{ ok} \\ -1 & : \text{ Fehler} \end{cases}$$

Zunächst wird überprüft, ob Verbindungsdaten vorhanden und noch aktuell sind. Ist dies nicht der Fall, muß sich ein Client mit `sec_announce` beim *qmaster* (erneut) anmelden:

Die Anmeldung umfaßt die Vereinbarung eines gemeinsamen geheimen symmetrischen Schlüssels und einer Connection ID. Diese Funktion deckt die *Announce Phase* von CSP auf der Seite vom Client ab. (siehe dazu auch Abbildung 9).

- Challenge generieren
- Anmeldung an Server verschicken
- Antwort von Server empfangen
- Zertifikat vom Server überprüfen und ihn authentifizieren
- symmetrischen Schlüssel vom Schlüsselmaterial generieren

Stellt die Funktion `sec_set_encrypt` anhand des *Tags* fest, daß die Nachricht nicht verschlüsselt werden muß, so sendet der Server wie gewohnt die Nachricht. Der Client hingegen hängt noch zusätzlich die Connection ID am Ende an, damit der Server den entsprechenden Eintrag in der Connection List findet, um die Zieladresse des Clients zu aktualisieren. Ansonsten werden die Daten mit `sec_encrypt` verschlüsselt:

- es wird ein MAC gebildet
- MAC und Nachricht werden mit dem symmetrischen Schlüssel verschlüsselt
- die Nachricht wird um die Connection-ID ergänzt
- die Send-Sequencenumber wird inkrementiert

Zuletzt werden der verschlüsselte MAC und die verschlüsselte Nachricht mit der Connection-ID an die Zieladresse versendet.

```
int sec_receive_message(char *fromcommproc, u_short
*fromid,char *fromhost,int *tag,char **buffer,u_long
*buflen,int synchron)
```

Analog zum *Send-Message* stellt diese Funktion die Substitution zur `receive_message`-Routine aus der `libcomm.a` dar. Mit dieser Funktion können empfangene, verschlüsselte Nachrichten wieder entschlüsselt und Anmeldungen der Clients verarbeitet werden.

fromcomproc	: Sender-Programm
fromid	: ID der Verbindung
fromhost	: Sender-Host
tag	: Nachrichtentyp
buffer	: enthält die empfangene Nachricht
buflen	: Länge dieser Nachricht
synchron	: Übertragungsmodus

$$return\_value = \begin{cases} 0 & : \text{ ok} \\ -1 & : \text{ Fehler} \end{cases}$$

Zunächst wird die Nachricht empfangen und anhand des *Tags* wird festgestellt, ob es sich um eine Anmeldung handelt oder ob die Nachricht verschlüsselt ist. Anmeldungen kann nur der Server empfangen. Mit `sec_respond_announce` wird in CSP die *Response Phase* abgedeckt:

- Zertifikat vom Client überprüfen
- Schlüsselmaterial erstellen
- Antwort an Client schicken
- Sequencenumbers initialisieren und sie mit dem Schlüsselmaterial unter der Connection ID in die Connection List eintragen
- Connection ID-Zähler inkrementieren

Ist die Nachricht verschlüsselt, muß sie mit `sec_decrypt` entschlüsselt werden. Der Ablauf unterscheidet sich ein bißchen, abhängig davon, ob ein Client oder der Server die Funktion aufruft:

- Der Server muß anhand der übermittelten Connection ID das dazugehörige Schlüsselmaterial und die Sequencenumbers aus der Connection List entnehmen und aus dem Schlüsselmaterial den symmetrischen Schlüssel generieren. Der Client kann all diese Daten aus der *gsd*-Struktur entnehmen.
- empfangener MAC und die Nachricht werden entschlüsselt
- MAC überprüfen
- Receive-Sequencenumber inkrementieren
- der Server muß zuletzt die neue Sequencenumber und die Adresse des Clients in die Connection List eintragen

`int sec_write_data2hd(void)`

Ein Client kann mit Hilfe dieser Routine alle sicherheitsrelevanten Daten, die zuvor mit dem Server vereinbart wurden, auf Festplatte verschlüsselt speichern. Diese Funktion wird aufgerufen, wenn der Client sich mit `leave()` beendet.

$$return\_value = \begin{cases} 0 & : \text{ ok} \\ -1 & : \text{ Fehler} \end{cases}$$

Arbeitsschritte:

- sicherheitsrelevante Daten, wie Connection ID, Sequencenumbers und Schlüsselmaterial werden mit dem öffentlichen RSA-Schlüssel vom Client verschlüsselt, sodaß nur er sie mit seinem privaten Schlüssel wieder entschlüsseln kann
- verschlüsselte Daten auf Festplatte in das *Reconnect File* schreiben

`int sec_reconnect(void)`

Diese Funktion, vom Client während der Initialisierung aufgerufen, liest die Daten aus dem *Reconnect File*, sodaß keine Anmeldung beim Server mehr erforderlich ist. Der Client kann nun sofort eine verschlüsselte Nachricht senden. Eine Anmeldung muß nur dann erfolgen, wenn der Aufruf dieser Funktion fehlschlägt oder wenn der Server mit den Daten aus dem *Reconnect File* nichts anfangen kann.

$$return\_value = \begin{cases} 0 & : \text{ ok} \\ -1 & : \text{ Fehler} \end{cases}$$

Ablauf:

- Connection ID, Sequencenumber und Schlüsselmaterial aus *Reconnect File* lesen
- Daten mit dem privaten RSA-Schlüssel entschlüsseln
- aus dem Schlüsselmaterial symmetrischen Schlüssel ableiten

10.2.3 Die SSL-Bibliothek *libcrypto.a*

Diese Bibliothek ist direkt aus der SSL-Protokoll-Implementation *SSLeay* ([23] bis [25]) entnommen worden. Auf eine weitere Schnittstellenbeschreibung möchte ich im weiteren verzichten und auf die Dokumentation der Autoren verweisen. Im wesentlichen geben aber schon die Funktionsnamen der enthaltenen Routinen ausreichend Aufschluß über ihre Aufgabe. Bis auf Ausnahme der DES-Verschlüsselungs-Funktionen erkennt man die Bibliotheksaufrufe daran, daß sie mit Großbuchstaben beginnen, z.B. `RSA_public_decrypt`.

10.2.4 Pfade und Dateien

CSP benötigt eine Vielzahl an Pfaden und Dateien, die in diesem Abschnitt genauer erläutert werden. Prinzipiell muß man zwischen Pfaden von Benutzern des CODINE-Systems und denen vom CODINE-System selbst unterscheiden. Beide haben dann wiederum je einen Pfad für die Zertifikate und einen für die RSA-Schlüssel.

Die Benutzer-Verzeichnisse

Die ersten beiden Pfade sind für den Benutzer des CODINE-Systems. Diese Pfade setzen sich aus dem *Home*-Verzeichnis des Benutzers, aus `/COD_SECURE/` und dem *Cell*-Verzeichnis von CODINE zusammen. Sollen mehrere *qmaster* in einem Netz nebeneinander arbeiten, so braucht jeder *qmaster* ein eigenes *Cell*-Verzeichnis. Standardmäßig wird der Inhalt der *Shell*-Umgebungsvariablen `COD_CELL` verwendet. Ist diese Variable nicht gesetzt, wird die Präprozessor Definition von `DEFAULT_CELL` als *Cell*-Verzeichnis benutzt. An dieses Grundverzeichnis wird entweder `/CERT/` oder `/KEY/` angehängt und man erhält den Zertifikat- und den Schlüsselpfad für den Benutzer, die beispielsweise folgende Gestalt für den Benutzer *bieder* haben können:

`/home/bieder/COD_SECURE/default/CERT/` bzw.
`/home/bieder/COD_SECURE/default/KEY/`

Das Benutzer-Zertifikat-Verzeichnis

Auf das Verzeichnis `/home/bieder/COD_SECURE/default/CERT/` sollte der Benutzer alle Rechte haben, hingegen andere nur Lese-Rechte besitzen dürfen. In diesem Verzeichnis sind mehrere Dateien abgelgt, die folgende Bedeutung haben:

`cert.txt` das bei der Zertifikatgenerierung ensteht, enthält das Zertifikat im Text-Format. Diese Datei wird später nicht mehr benötigt.

`default.txt` enthält die Standardwerte für `cert.txt`. Werden diese Werte bei der Zertifikatgenerierung nicht oder nur teilweise benutzt, werden die aktuellen Werte als Standardwerte wieder abgespeichert.

`request.pem` ist der selbstsignierte Zertifikat-*Request* im PEM-Format (beschrieben in [14]), der von der CA signiert werden muß, um ein vollständiges Zertifikat zu werden.

`cert.pem` ist schließlich das vollständige, von der CA signierte, Zertifikat im PEM-Format.

Das Benutzer-Schlüssel-Verzeichnis

Auf das Schlüsselverzeichnis `/home/bieder/COD_SECURE/default/KEY/` sollte niemand sonst als der Benutzer Zugriffsrechte haben. In diesem Verzeichnis liegen nur zwei Dateien:

`rsakey.pem` enthält den RSA-Schlüssel des Benutzers und zwar sowohl den öffentlichen, als auch den privaten Teil. Zur Sicherheit ist der Schlüssel mit einem Paßwort DES-verschlüsselt.

`reconnect.dat` beinhaltet alle sicherheitsrelevanten Daten, die der Client braucht, um erneut Verbindung mit den Server aufzunehmen, ohne sich vorher bei ihm anmelden zu müssen. Diese Daten sind mit dem öffentlichen Schlüssel vom Benutzer verschlüsselt worden, sodaß niemand anderes Zugriff auf diese Daten hat, auch wenn sie gerade über das Netzwerk übertragen werden.

Die CODINE-Verzeichnisse

Neben den Pfaden und Dateien des Benutzers gibt es Entsprechungen für das CODINE-System. Diese Pfade setzen sich aus dem CODINE-*Root*-Verzeichnis und dem *Cell*-Verzeichnis zusammen. Das *Root*-Verzeichnis wird aus der *Shell*-Umgebungsvariablen `CODINE_ROOT` entnommen. Ist diese nicht gesetzt, so wird die Präprozessor-Definition von `ROOT_DIR` benutzt. Wie beim Benutzer-Verzeichnis gilt auch hier das gleiche für das *Cell*-Verzeichnis an dem ebenfalls `/CERT/` bzw. `/KEY/` angehängt wird, um das Zertifikat- bzw. das Schlüssel-Verzeichnis für CODINE zu erlangen. Diese beiden Verzeichnisse könnten folgende Gestalt haben:

`/usr/CODINE/default/CERT/` bzw.
`/usr/CODINE/default/KEY/`

Das CODINE-Zertifikat-Verzeichnis

Für die Zugriffsrechte dieses Verzeichnisses gilt das gleiche wie für das Zertifikat-Verzeichnis des Benutzers. Auch enthält es die gleichen Dateien, die nun allerdings nicht für den Benutzer, sondern für das CODINE-System, wie den *qmaster* oder die *execd*, bestimmt sind. Dies bedeutet also, daß sich das ganze System ein Zertifikat und einen RSA-Schlüssel teilt. Neben den bekannten Dateien enthält dieses Verzeichnis zudem noch welche für die CA:

`ca_cert.txt` wie `cert.txt`, nur diesmal für die CA.

`ca_def.txt` Funktion wie `default.txt`.

`ca_cert.pem` ist das Zertifikat der CA.

`ca_cert.pem.srl` enthält den Wert der nächsten freien *Serialnumber*. Jedes Zertifikat, das die CA signiert, bekommt von ihr eine eindeutige *Serialnumber* zugewiesen.

Das CODINE-Schlüssel-Verzeichnis

Wie beim Benutzer-Schlüssel-Verzeichnis, sollte auch auf dieses Verzeichnis niemand anderes Zugriff haben. Es enthält ebenfalls einen RSA-Schlüssel und ein *Reconnect*-File und zudem die Schlüsseldatei der CA:

`ca_key.pem` ist wie `rsakey.pem`, enthält jedoch den Schlüssel der CA.

10.2.5 Die Applikationen

Mit *SSLeay* stehen verschiedene Werkzeuge zur Verfügung, um Zertifikate und RSA-Schlüssel zu generieren und zu handhaben. Allerdings sind diese Werkzeuge so angelegt, daß sie über Parametersteuerung universell einsetzbar und dadurch sehr umständlich zu gebrauchen sind. Aus diesem Grund habe ich aus den alten *Tools* neue abgeleitet, die wesentlich einfacher zu bedienen und direkt auf CODINE abgestimmt sind. Es gibt vier verschiedene Hilfsprogramme:

- *sec_ca*
  Mit diesem Programm kann man den RSA-Schlüssel und das Zertifikat der *Certificate Authority*, kurz CA, erstellen.
- *sec_req*
  Dieses Programm erstellt für einen Benutzer seinen RSA-Schlüssel und ein *Certificate Request*.
- *sec_sign*
  Mit diesem Tool signiert die CA ein *Certificate Request*.

- *sec_verify*
  Schließlich kann man mit diesem Werkzeug ein Zertifikat auf seine Gültigkeit überprüfen.

`sec_ca`

Mit diesem Programm kann der CODINE-System-Administrator eine CA erstellen, die später alle Zertifikate signiert, die im System verwendet werden sollen. Zu dieser CA gehört ein RSA-Schlüssel, der in einem Verzeichnis aufbewahrt werden sollte, in dem niemand sonst Schreib- oder Leserecht hat. Sicherheitshalber wird der RSA-Schlüssel nochmals mit einem Paßwort DES-verschlüsselt, sodaß der Schlüssel auch sicher über das Netzwerk transportiert werden kann. Außerdem braucht eine CA auch ein Zertifikat, das mit dem eigenen RSA-Schlüssel signiert wird und das in einem Verzeichnis liegt, das alle Benutzer des CODINE-Systems lesen können, in das aber niemand sonst schreiben darf.

Als Parameter können optional verwendet werden:

- `-keylen` *argument*
  Die RSA-Schlüssellänge kann frei angegeben werden, sollte allerdings nicht unter dem Grundwert von 512 Bits liegen.[6]

- `-days` *argument*
  Mit diesem Parameter kann man festlegen, wie lange das Zertifikat gültig ist. Der Grundwert beträgt 365 Tage.

- `-cell` *argument*
  Läuft ein CODINE-System unter einer anderen *Cell*, werden Zertifikat und Schlüssel in das entsprechende *Cell*-Verzeichnis abgelegt.

Im wesentlichen werden von `sec_ca` folgende Aufgaben erfüllt:

- Generierung des RSA-Schlüssels
- Schlüssel mit einem Paßwort DES-verschlüsseln
- Generierung des Zertifikats
- dieses Zertifikat mit dem generierten Schlüssel signieren
- Zertifikat und Schlüssel in die entsprechenden Verzeichnisse ablegen

[6]Je größer der Wert gewählt wird, um so länger dauert die Schlüsselgenerierung und später das Verschlüsseln selbst, aber um so sicherer wird der Schlüssel.

`sec_req`

Jeder CODINE Benutzer muß mit `sec_req` zunächst einen eigenen RSA-Schlüssel und ein *Certificate Request* erstellen, bevor er das System nutzen kann. Der Schlüssel wird im Schlüssel-Verzeichnis, für das nur der Benutzer Rechte hat, unter seinem *Home*-Verzeichnis abgelegt; der Request hingegen sollte zumindest für die CA lesbar sein.

Es gibt folgende optionale Parameter:

- `-keylen` *argument*
  Die RSA-Schlüssellänge kann frei angegeben werden, sollte allerdings nicht unter dem Grundwert von 512 Bits liegen.
- `-codsys`
  Wird dieser Parameter gesetzt, so wird ein *Certificate Request* für die CODINE Komponenten (*qmaster, execd*, etc.) generiert. Der Programmablauf bleibt der gleiche, nur werden Schlüssel und Zertifikat diesmal in die CODINE-System-Verzeichnisse abgelegt.
- `-cell` *argument*
  Läuft ein CODINE-System unter einer anderen *Cell*, werden Zertifikat und Schlüssel in das entsprechende *Cell*-Verzeichnis abgelegt.

Der Ablauf ist ähnlich wie bei `sec_ca`:

- Generierung des RSA-Schlüssels
- Schlüssel mit einem Paßwort DES-verschlüsseln
- Generierung der Zertifikatsanforderung
- diese Anforderung mit dem generierten Schlüssel signieren
- Zertificate Request und Schlüssel in die entsprechenden Verzeichnisse ablegen

`sec_sign`

Mit diesem Programm kann der CODINE Administrator mit dem privaten Schlüssel der CA *Certificate Requests* von Benutzern oder vom CODINE-System signieren. Das Ergebnis ist ein Zertifikat, das nun innerhalb von CODINE genutzt werden kann.

Ein Parameter ist zwingend:

- `-in` *argument*
  Das *Input File*, der *Certificate Request* muß spezifiziert werden und zwar mit vollem Pfadnamen. Das signierte Zertifikat wird dann im gleichen Verzeichnis abgelegt.

Dazu kommen wieder optionale Parameter:

- `-days` *argument*
  Mit diesem Parameter kann man festlegen, wie lange das Zertifikat gültig ist. Der Grundwert für Benutzer-Zertifikate beträgt 180 Tage.
- `-cell` *argument*
  Läuft ein CODINE System unter einer anderen *Cell*, wird das Zertifikat in das entsprechende *Cell*-Verzeichnis abgelegt.

Während der Abarbeitung geschieht folgendes:

- *Request* einlesen
- RSA-Schlüssel der CA einlesen und mit Paßwort entschlüsseln
- *Request* signieren, d.h. mit dem privaten Schlüssel der CA „verschlüsseln“
- entstandenes Zertifikat abspeichern

`sec_verify`

Dieses Hilfsprogramm überprüft ein Zertifikat auf seine Gültigkeit. Ein Zertifikat ist gültig, wenn es die beiden folgenden Bedingungen erfüllt:

- Das aktuelle Datum muß zwischen den beiden Werten `notBefore` und `notAfter` des Zertifikats liegen. Ist dies nicht der Fall, ist das Zertifikat noch nicht gültig bzw. schon abgelaufen.
- Das Zertifikat muß von der CA signiert worden sein. Dies gilt auch für das Zertifikat der CA, das selbstsigniert sein muß.

Ein Zertifikat ist selbstsigniert, wenn es mit dem privaten Schlüssel signiert wurde, dessen zugehöriger öffentlicher Schlüssel im Zertifikat selbst enthalten ist.

Als Parameter sind folgende zulässig:

- `-ca`
  Wird dieser Parameter gesetzt, wird dem Programm mitgeteilt, daß ein Zertifikat einer CA überprüft werden soll. Für ein solches Zertifikat ist es ausreichend, daß es selbstsigniert ist.

- `-cell` *argument*
  Läuft ein CODINE System unter einer anderen *Cell*, werden Zertifikat und Schlüssel in dem entsprechende *Cell*-Verzeichnis gesucht.

- `-in` *argument*
  Das zu prüfende Zertifikat wird im *Input File* spezifiziert.

11 Die Integration von CSP in CODINE

In diesem Kapitel möchte ich beschreiben, wie CSP in CODINE einzusetzen ist, welche Auswirkungen CSP auf den Code von CODINE hat und wie es das Gesamtsystem beeinflußt. Während für den CODINE-Entwickler alle drei Aspekte von Interesse sein könnten, ist es für den Anwender wohl nicht von Belang, inwieweit sich die CODINE-Sourcen verändert haben.

11.1 Einsatz von CSP in CODINE

In diesem Abschnitt möchte ich beschreiben, welche Vorkehrungen getroffen werden müssen, um CSP in CODINE verwenden zu können.

11.1.1 Übersetzen des Codes

Um CODINE mit dem erweiterten Sicherheitskonzept von CSP nutzen zu können, muß zuerst das SSL-Paket im Verzeichnis `sec/ssl/` übersetzt werden. Dabei wird die Bibliothek `libcrypto.a` gebildet, die in das entsprechende Binär-Verzeichnis zu kopieren und mit `ranlib -t libcrypto.a` zu aktualisieren ist.

Wird nun der AIMK mit der Option `-secure` aufgerufen, wird CODINE und der Code von CSP übersetzt.

11.1.2 Generierung von Zertifikaten und Schlüsseln

Bevor die verschiedenen Komponenten von CODINE gestartet werden können, müssen zuvor Zertifikate und RSA-Schlüssel generiert werden. Zu diesem Zweck stehen verschiedene Hilfsprogramme zur Verfügung, die im vorangegangenen Kapitel beschrieben wurden.

Zuerst muß der Systemadministrator von CODINE eine *Certificate Authority* mit Hilfe von `sec_ca` bilden. Mit diesem Tool werden Zertifikat und RSA-Schlüssel der CA gebildet.

Als nächstes ist für das System selbst ein Zertifikat und ein RSA-Schlüssel zu generieren. Dazu muß der Administrator `sec_req` mit der Option `-codsys` aufrufen und erhält einen *Certificate Request* und den RSA-Schlüssel. Das endgültige Zertifkat entsteht, wenn er den *Request* mit `sec_sign` signiert.

Alle Komponenten von CODINE, die zum System gehören (`cod_qmaster, cod_execd, cod_schedd`), teilen sich sowohl Zertifikat als auch Schlüssel.

Aber auch jeder Benutzer von CODINE benötigt Zertifikat und Schlüssel. Mit `sec_req` kann er sich ebenfalls seinen RSA-Schlüssel und ein *Certificate Request* erstellen, der von dem Administrator mit `sec_sign` mit dem Schlüssel der CA signiert werden muß. Das Ergebnis ist ein gültiges Zertifikat, das der Benutzer nun für CODINE einsetzen kann.

11.1.3 Eingabe des Paßwortes

Nachdem CSP vom Compiler übersetzt wurde und alle nötigen Zertifikate B und Schlüssel generiert wurden, kann man beginnen, die einzelnen CODINE-Komponenten zu starten. Dabei wird unabhängig vom Aufrufer nach einem Paßwort gefragt, mit dem sein RSA-Schlüssel verschlüsselt wurde, damit dieser Schlüssel von Festplatte eingelesen werden kann.

Nachdem diese „Hürden" genommen wurden, ist das CODINE-System wie gewohnt zu benutzen und CSP arbeitet ab diesem Zeitpunkt völlig transparent im Hintergrund.

11.2 Änderungen im CODINE-Code

CSP wurde möglichst gekapselt in Form der Bibliothek `libsec.a` implementiert. Jedoch ließ es sich nicht vollständig vermeiden, daß Änderungen im CODINE-Quellcode vorgenommen werden mußten und seien es nur Einfügungen von Aufrufen der Bibliotheksroutinen.

In diesem Abschnitt möchte ich auf diese Änderungen zu sprechen kommen und darstellen, wie und an welchen Stellen die Funktionen der Bibliothek einzusetzen sind. Grundvoraussetzung ist jedoch das Einbinden der Header-Datei `sec_lib.h`, in der die Prototypen der Bibliotheksaufrufe enthalten sind:

```
#ifdef SECURE
#include "sec_lib.h"
#endif
```

Für Code, der sich auf CSP bezieht, sollte mit einer Präprozessor-Anweisung des C-Compilers überprüft werden, ob die Bibliotheken für CSP überhaupt mitübersetzt und gebunden wurden:

```
#ifdef SECURE
     ... Anweisungen von CSP ...
#endif
```

11.2.1 AIMK

Der AIMK, *Architecture Independet MaKe*, für CODINE wurde für CSP um eine Option erweitert:

```
-secure
```

Wird der AIMK mit diesem Parameter aufgerufen, wird `SECURE` mit der Compileroption `-DSECURE` definiert und der von CSP abhängige Code wird mitübersetzt.

11.2.2 *Makefile*

Das *Makefile* `src/Makefile` wurde nur um wenige Erweiterungen ergänzt, sodaß unter anderem die beiden Bibliotheken `libsec.a` und `libcrypto.a` zu allen CODINE-Komponenten gebunden werden. Aus gleichem Grund wurden auch `schedd/Makefile` und `history/Makefile` verändert.

11.2.3 Einführung neuer *Message Tags*

Im CODINE-Kommunikationssystem gibt es verschiedene *Message Tags*[7], die die Art einer Nachricht beschreiben und die in der Header-Datei `apilib/cod_api_intern.h` definiert werden. Um die Sicherheitsmechanismen von CSP nutzen zu können, ist die Liste der Nachrichtentypen um drei weitere *Message Tags* erweitert worden:

- TAG_SEC_ANNOUNCE
  Bei Nachrichten von diesem Nachrichtentyp handelt es sich um eine Anmeldung beim *qmaster*. Diese werden bei dem Bibliotheksaufruf `sec_announce_connection` der `libsec.a` Bibliothek generiert und versendet.

[7] über die Message Tags wird auch das Verschlüsselungsverhalten gesteuert, siehe dazu auch die Funktion `sec_set_encrypt` aus der `libsec.a`

- TAG_SEC_RESPOND
  Nachrichten mit diesem *Message Tag* sind Antworten auf Anmeldungen vom *qmaster* und entstehen beim Aufruf von `sec_respond_announce`, ebenfalls aus der Bibliothek `libsec.a`.
- TAG_SEC_ERROR
  Treten Fehler im Zusammenhang mit CSP auf, werden Nachrichten von diesem Nachrichtentyp an den entsprechenden Kommunikationspartner versendet.

11.2.4 Erweiterung der *Error Return Codes* der *commlib*

Die *commlib* unterstützt eine Reihe von *Error Return Codes*, die den Fehler einer Kommunikation näher beschreiben. Die Struktur dieser Codes ist um drei zusätzliche Einträge erweitert worden:

- SEC_ANNOUNCE_FAILED
  Diese Fehlermeldung tritt auf, wenn die Anmeldung während dem Ablauf von CSP nicht korrekt durchgeführt werden konnte.
- SEC_SEND_FAILED
  Dieser *Error Code* zeigt an, daß eine Nachricht vor dem Versand nicht richtig vorbereitet werden konnte in Hinblick auf CSP. Beispielsweise konnte die Nachricht nicht verschlüsselt werden, oder es schlug fehl, die Connection ID am Ende des Nachrichten-Puffers zu schreiben.
- SEC_RECEIVE_FAILED
  Eine Nachricht wurde zwar korrekt empfangen, konnte aber von CSP nicht ausgewertet werden.

Betroffen von dieser Erweiterung sind die Dateien `commd/commlib.c` und `commd/commlib.h`.

11.2.5 Zusätzliche *Exit*-Funktion

Bislang wurde immer eine *Exit*-Funktion von einer CODINE-Komponente aufgerufen, wenn sich diese beendet hat. Zu dieser Funktion kommt nun eine weitere hinzu, die die sicherheitsrelevanten Daten auf Festplatte sichert. Der *qmaster* schreibt seine Connection List, alle anderen Komponenten ein Reconnect File, damit bei einer folgenden Kommunikation keine erneute Anmeldung erfolgen muß. Für diese Änderung mußten die Dateien `utilib/cod_exit.c` und `utilib/cod_utility.h` erweitert werden.

11.2.6 Realisierung der *Connection List*

Um die Connection List beim *qmaster* verwirklichen zu können, mußten die Dateien `apilib/cod_boundaries.h` und `apilib/cod_all_listsL.h` erweitert und die Datei `apilib/secL.h` neu eingeführt werden.

11.2.7 Änderung im Code der Komponenten

Bevor CODINE-Komponenten miteinander kommunizieren können, müssen sie ihre sicherheitsrelevanten Daten vor der ersten Kommunikation initialisieren, sich aber auch beim *commd* anmelden. Daher wird die Initialisierungsroutine in der Funktion `do_enroll` von `apilib/cod_any_request.c` aufgerufen, wo auch die Anmeldung stattfindet. Aus diesem Grund sind keinerlei Änderungen im Code der Komponenten notwendig, um sicher kommunizieren zu können, wenn sie die `do_enroll`-Funktion nutzen. Eine Ausnahme bildet der *qmaster*:

11.2.8 Änderung im Code vom *qmaster*

Im Code vom *qmaster* muß der Tag TAG_SEC_ANNOUNCE bekannt gemacht werden:

```
     switch (tag) {

     /* ======================================== */
#ifdef SECURE
     case TAG_SEC_ANNOUNCE:    /* completly handled in libsec  */
        clear_packbuffer(&pb);
        break;
#endif
```

Die Anmeldung wird dann komplett in der `libsec.a` abgehandelt.

11.2.9 Substitution der Kommunikationsaufrufe

Alle Aufrufe von `send_message` und `receive_message` im Code müssen durch die jeweiligen Entsprechungen `sec_send_message` bzw. `sec_receive_message` der *seclib* substituiert werden. Dies geschieht am einfachsten durch das Einbinden der folgenden *Header*-Datei:

```
#ifdef SECURE
    include "sec_subst.h"
#endif
```

In diesem Zusammenhang möchte ich darauf hinweisen, daß bei der Angabe der Zieladresse des Kommunikationspartners beim Nachrichtenversenden keine *Wildcards* erlaubt sind. Die Adresse für eine sichere Kommunikation muß immer eindeutig sein!

11.3 Mehraufwand durch den Einsatz von CSP

Der Administrator eines CODINE-Systems hat durch den Einsatz von CSP einen zusätzlichen Verwaltungsaufwand. Zunächst muß er für sein System eine *Certificates Authority* erstellen und muß damit einmalig das Zertifikat für jeden Benutzer von CODINE signieren. Nach Ablauf der Gültigkeit des jeweiligen Zertifikates muß diese Prozedur wiederholt werden, deren Durchführung jedoch durch die tools sec_ca und sec_sign erleichtert wird.

Neben dieser Zertifikatsverwaltung muß der Administrator beim Starten von CODINE Komponenten wie dem `qmaster` oder dem `execd` etc. zusätzlich ein Paßwort eingeben, damit der verschlüsselte RSA-Schlüssel korrekt eingelesen werden kann. Allerdings erfolgt dieser Vorgang nicht häufig, da die CODINE-Dämonen über einen längeren Zeitraum kontinuierlich laufen.

Auf der Benutzerseite sieht es ähnlich aus. Der Benutzer muß einmalig eine Zertifikatsanforderung stellen, was mit dem tool sec_req geschieht, und sie von der Certificate Authority signieren lassen, damit er ein gültiges Zertifikat erhält.

Wie der Administrator muß auch der Benutzer beim Aufruf von CODINE-Befehlen wie `qsub` oder `qstat` etc. jedesmal ein Paßwort eingeben. Dies ist notwendig, da sich ein Befehl nach seiner Abarbeitung selbst beendet und das Wissen über das Paßwort wieder verloren geht. Je nach dem, wie intensiv der Benutzer CODINE einsetzt, kommt dies pro Tag recht häufig vor.

Dementsprechend muß für jeden auszuführenden Befehl das Zertifikat und der RSA-Schlüssel vom Dateisystem eingelesen und letzterer wie erwähnt auch entschlüsselt werden, was auch wieder Zeit benötigt, die aber wegen der geringen Größe der beiden Dateien kaum ins Gewicht fällt.

Nachdem die zwei Dateien eingelesen wurden, muß sich ein CODINE-Kommando beim *qmaster* anmelden, d.h. es wird das *CODINE Security Protocol* abgearbeitet. Dazu wird je eine zirka 600 Bit lange Nachricht zwischen Client (Kommando) und Server (*qmaster*) ausgetauscht und kleine Sequencen (kleiner als 64 Bit) werden RSA-ver- bzw. entschlüsselt. Neben diesen Tätigkeiten finden noch wenige Vergleichs- und Generierungs-Operationen statt, die weniger rechenintensiv sind. Allerdings muß diese Anmeldeprozedur beim Server nur einmal pro Tag erfolgen, da der Client nach seiner Beendigung alle verbindungsbezogenen Daten in einer verschlüsselten Reconnect-Datei abspeichert. Beim Starten eines neuen Clientprogramms liest der Client die Reconnect-Datei ein und entschlüsselt sie - die Anmeldung kann somit für alle weiteren Verbindungen entfallen und spielt bei der Gesamtaufwandsabschätzung kaum eine Rolle.

Jetzt, nachdem die Anmeldung abgeschlossen ist bzw. das Reconnect-File gelesen wurde, können die Nachrichten verschlüsselt übertragen werden. Der Verschlüsselungsaufwand hängt direkt von der Schlüssellänge, von der Länge der Nachricht und dem Verschlüsselungsmodus ab. Wenn die Schlüssellänge groß gewählt wird, die Nachricht lang ist und wenn als Modus die Dreifachverschlüsselung verwendet wird, wird der Aufwand zur Verschlüsselung anwachsen. Generell sollte man überlegen, welche Nachrichten es wert sind, verschlüsselt zu werden. Über den *Message Tag* läßt sich das Verschlüsselungsverhalten steuern. In der Praxis macht sich die Verschlüsselung im Vergleich zu den relativ hohen Netzwerklaufzeiten jedoch nicht bemerkbar - die Kommunikation wird nicht spürbar beeinträchtigt.

Durch die Verwendung der Sicherheitsmechanismen wird der ausführbare Code und der Speicherplatzbedarf zur Laufzeit der verschiedenen CODINE-Komponenten nur gering vergrößert (ca. um 180 kByte). Nur der *qmaster* erhält zusätzlich eine dynamische Liste, die die Daten der Verbindungen der letzten zwölf Stunden enthält. Die Größe der Liste ist allerdings auf die Anzahl der Benutzer beschränkt, d.h. jeder Benutzer erhält in der Regel nur einen Eintrag in der Connection List. Zudem werden Verbindungen, die älter als zwölf Stunden sind, automatisch aus der Liste entfernt.

Die Größe der zu übertragenden Nachrichten wird durch die Verschlüsselung ebenfalls kaum länger (ca. 60 Bits), wobei sich das Verhältnis bei kleineren Nachrichten, wie ein 8-Bit langer Acknowledge, ungünstig verschiebt.

Abschließend läßt sich also sagen, daß sich die Performance von CODINE durch den Einsatz von CSP kaum spürbar verschlechtert. Einzig störend für den Benutzer ist es, bei jedem neuen CODINE-Kommando das Paßwort wiederholt eingeben zu müssen. Programme, wie der `qmon`, die nach der Ausführung eines Kommandos im Speicher verbleiben, können hier Abhilfe schaffen.

12 Zusammenfassung und Ausblick

Im Rahmen dieser Diplomarbeit ist es mir gelungen, ein Sicherheitsprotokoll zu entwerfen und zu implementieren, um die Kommunikation in der Anwendung CODINE sicher zu gestalten, wobei alle Anforderungen, wie Vertraulichkeit, Authentifikation und Datenintegrität, erfüllt werden. Kennzeichnend für dieses an SSL angelehnte Protokoll ist, daß es äußerst effizient arbeitet und so implementiert wurde, daß es nahezu völlig transparent sowohl für den Anwender, als auch für den Entwickler abläuft. Durch eine Realisierung in Form einer Bibliothek wurde ein klare Schnittstelle zu der bestehenden Anwendung definiert und es ist auch ohne Probleme möglich, die Implementation dieses Protokolls in anderen Anwendungen zu benutzen.

Selbstverständlich kann ich nicht dafür garantieren, daß die Konzeption von CSP frei von Sicherheitslücken ist und daß die Implementierung keine Fehler aufweist, die Sicherheitsmängel hervorrufen könnten. Sicherheitskonzepte sind immer nur so stark, wie ihre schwächste Stelle. Ihnen muß das Hauptaugenmerk gelten. Deswegen habe ich den Entwurf des Protokolls mit sehr viel Sorgfalt durchgeführt und habe viel Literatur und auch einen Experten[8] der Kryptographie zu rate gezogen.

Bei der Implementierung sollte man für die Zukunft noch einige Punkte berücksichtigen. Zum einen könnte die 64 Bit DES-Verschlüsselung (effektiv 56 Bit) des RSA-Schlüssels zu wenig sein. Ein 128 Bit-Schlüssel wäre auch noch über die nächsten Jahre hinaus sicher. Ein anderer Punkt betrifft die Initialisierung des Zufallszahlengenerators. Sie erfolgt im Moment nur über die Systemzeit, was zu wenig ist, wie man bei der Implementation des SSL-Protokolls von Netscape sehen konnte.

CSP verwendet Mechanismen der Kryptographie aus der freien SSL-Implementation *SSLeay*. Wie erwähnt ist dieses Projekt allerdings noch in der Aufbauphase und es ist sicherlich lohnend, die Entwicklung weiterer Versionen mitzuverfolgen, da sie sicherer, fehlerfreier und schneller sein könnten. Jedoch hat sich die Schnittstelle zur benötigten Library inzwischen verändert, sodaß auch kleinere Anpassungen in der Implementierung vom CSP nötig wären.

Neben Fragen der Sicherheit gibt es noch andere Punkte zu berücksichtigen, um CSP einsetzen zu können: Gehen Nachrichten während einer bestehenden Kommunikation verloren, aus welchen Gründen auch immer, so

[8] Holger Reif, siehe [18] und [19]

müssen alle weiteren Nachrichten wegen einem möglichen Angriff auf die Sicherheit abgelehnt werden - der andere Kommunikationspartner muß sich erneut anmelden. Dies kann man ihm aber nicht immer mitteilen, wenn er auf einen ganz bestimmten Nachrichtentyp wartet. Hierzu müßte das Kommunikationssystem von CODINE so erweitert werden, daß man nicht nur einen, sondern ein Menge von Nachrichtentypen beim Empfangen angeben kann.

An dieser Stelle möchte ich auch nicht verschweigen, daß das Austesten der Implementation von CSP aus Zeitmangel leider viel zu kurz ausgefallen ist. Getestet wurde nur auf SUNs unter SunOS und beschränkte sich auf einzelne Komponenten. Das Testen erfolgte unsystematisch und deckte kaum Randbedingungen ab.

Wie man sieht, sind noch einige Probleme zu lösen und verschiedene Aspekte zu betrachten, bis CSP in der Praxis eingesetzt werden kann. Allerdings konnte ich mit der vorliegenden Arbeit eine beachtliche Strecke in diese Richtung zurücklegen und den richtigen Weg bis zum endgültigen Ziel weisen, zu dem es nur noch wenige Schritte sein sollten.

Literatur

[1] Alefeld, Andre; Haas, Andreas; Ferstl, Fritz. *Common Usable List Library (CULL)*. Genias Software GmbH Neutraubling.

[2] CCITT Recommendation X.208. *Specification of Abstract Syntax Notation One (ASN.1)*. Blue Book, Melbourne 1988.

[3] CCITT Recommendation X.509. *The Directory - Authentication Framework*. Blue Book, Melbourne 1988.

[4] *CODINE Manual*. Genias Software GmbH Neutraubling.

[5] Engelhardt, Stefan. *Communication Daemon (commd) and Communication Library (commlib)*. Genias Software GmbH Neutraubling.

[6] Eriksson, Peter. *Peter's Ident Daemon*. ⟨http://ftp.lysator.liu.se/pub/ident/⟩

[7] Fischer, Matthew. *How to implement the Data Encryption Standard (DES)*. ⟨mfischer@heinous.isca.uiowa.edu⟩

[8] Fox, Dirk. *Schlüsseldienst - Private Kommunikation mit PEM und PGP*. c't 9/95.

[9] Hagemann, Hagen; Rieke, Andreas. *Datenschlösser - Grundlagen der Kryptologie*. c't 8/94.

[10] Hickman, Kipp E.B.; Elgamal, Taher. *The SSL Protocol (version 3 of specification)*. Netscape Communications Corp. June 1995. ⟨http://home.netscape.com/newsref/std/SSL.html⟩

[11] Johns, M. St.; *Identification Protocol*. RFC 1413, February 1993.

[12] Kaufman, Charlie; Perlman, Radia; Speciner, Mike. *Network Security - Private Communication in a Public World*. Prentice-Hall, New Jersey 1995.

[13] Laurie, Ben. *Apache-SSL Webserver*. ⟨http://arachnet.algroup.co.uk/Apache-SSL/⟩

[14] Linn, J.; Kent, S.; Balenson, D.; Kaliski, B. *Privacy Enhancement for Internet Electronic Mail, Part I-IV*. RFC 1421 - 1424, February 1993.

[15] Martin, Christoph. *SSL-MZtelnet.*
⟨http://www.uni-mainz.de/ martin/⟩

[16] NBS FIPS PUB 46. *Data Encryption Standard.* National Bureau of Standards, U.S. Department of Commerce, Jan. 1977.

[17] Rivest, R. *The MD5 Message-Digest Algorithm.* RFC 1321, April 1992.

[18] Reif, Holger. *Netz ohne Angst - Sicherheitsrisiken des Internets.* c't 9/95.

[19] Reif, Holger. *Homepage.*
⟨http://Remus.PrakInf.TU-Ilmenau.DE/Reif/⟩

[20] Schneier, Bruce. *Applied Cryptography.* John Wiley & Sons NY 1994. (gibt es meines Wissens jetzt in der 2. Auflage)

[21] Salomaa, Arto. *Public-Key Cryptography.* Springer-Verlag, 1990.

[22] Ylönen, Tatu. *International Cryptography Pages.*
⟨http://www.cs.hut.fi/crypto/⟩

[23] Young, Eric A.; Hudson, Tim J. *SSLeay FAQ.*
⟨http://www.psy.uq.oz.au/ ftp/Crypto/⟩

[24] Young, Eric A.; Hudson, Tim J. *SSLeay Programmer Reference.*
⟨http://www.psy.uq.oz.au/ ftp/Crypto/ssl.html⟩

[25] Young, Eric A.; Hudson, Tim J. *SSLeay Implementation.*
⟨ftp://ftp.psy.uq.oz.au/pub/Crypto/SSL⟩

[26] Young, Eric A.; Hudson, Tim J. *SSLeay Applications.*
⟨ftp://ftp.psy.uq.oz.au/pub/Crypto/SSLapps/⟩

A Source Code