Projektarbeit SNA05

Verify X.509 Zertifikate

X.509
Trust-Chain
Unterstützung
Für Linux
IPSec Stack

Dokumentation V1.0

Marco Bertossa	**Studenten**
Andreas Schleiss	
Dr. Andreas Steffen	**Betreuer**
ZHW Winterthur	**Ort**
21. Mai bis 6. Juli 2001	**Datum**

1 Zusammenfassung

Virtual Private Network (VPN) ist eine Möglichkeit, einzelne Hosts und Subnetze über öffentliche oder halb – öffentliche Kommunikationskanäle zu verbinden. Um die Sicherheit der Daten zu gewährleisten, wird ein sogenannter "sicherer IP Tunnel" aufgebaut, durch den die Daten verschlüsselt von einem Ende zum anderen gesendet werden.

Eine Implementation eines solchen VPN basiert auf dem IPSec (IP Security) Protokoll. Unter Linux wird ein OpenSource IPSec Patch eingesetzt, welcher eine sichere Verbindung aufbaut und mit Hilfe von X509 Zertifikaten (ein weiterer Zusatz) den IPSec Client authentisiert und dessen Public Key daraus extrahiert.

Das Problem dieser aktuellen Version war, dass die Authentifizierung des IPSec Clients durch die gesendeten Zertifikate aufgrund des Vorhandenseins von lokal auf dem Security Gateway gespeicherten Zertifikate durchgeführt wurde. Die Verbindung wurde hergestellt, auch wenn das gesendete Zertifikat aus irgendeinem Grund ungültig, jedoch auf dem Security Gateway gespeichert war. Auch wurde die Administration der Zertifikate bei einer grösseren Anzahl unübersichtlich, da der Administrator diese von Hand in das durchsuchte Verzeichnis hinein kopieren musste; diese Methode ist allerdings zu fehleranfällig.

Wir haben den Zusatz der Authentisierung durch x509 Zertifikate erweitert, so dass beim Verbindungsaufbau die gesendeten Zertifikate auf deren Gültigkeit geprüft werden können. Dies geschieht indem die Signaturen der Zertifikate entlang der Vertrauenskette verfolgt werden und zusätzlich geprüft wird, ob die Zertifikate nicht gesperrt oder nicht mehr gültig sind. Die Folge davon ist, dass lokal auf dem Security Gateway nur noch die CA Zertifikate und die CRLs (Certificate Revokation List) gespeichert werden müssen. Beim Starten des Pluto-Dämon (der eigentlichen Gateway Software) werden die vorhandenen Zertifikate in den Speicher geladen und können auf diese Weise bei neuen Verbindungen einfach und schnell zur Verifizierung hinzugezogen werden.

Winterthur, 6. Juli 2001

Marco Bertossa Andreas Schleiss

2 Einleitung

2.1 Verifizierung von x509 Zertifikaten

Beim Aufbau eines Virtual Private Network können x509- Zertifikate zur Authentifizierung und Autorisierung verwendet werden. Um die Integrität der Zertifikate zu gewährleisten, müssen diese auf Gültigkeit aufgrund des Ablaufdatums und von Veränderungen innerhalb des Zertifikates geprüft werden. Unsere Dateien `verify_x509.c` und `verify_x509.h` stellen zusätzliche Funktionen für diese Aufgabe zur Verfügung.

2.2 Gültigkeit einer Anfrage

Wir überprüfen nun eine Anfrage, indem wir das zum Initiator der Anfrage gehörende Zertifikat laden, das Ablaufdatum des Zertifikates als gültig erklären, um dann damit die Signatur des Initiator zu verifizieren. Ebenfalls werden die CA Zertifikate entlang der Trust-Chain bis zur Root-Certification-Authority auf deren Gültigkeiten überprüft.

Der Aussteller des Zertifikates, eine Certificate-Authority, besitzt zusätzlich eine CRL-Datei, in der gesperrte x509-Zertifikate aufgelistet sind. Während der Verifizierung eines Zertifikates wird ebenfalls geschaut, ob das empfangene und die geladenen Zertifikat nicht gesperrt sind.

2.3 Handhabung unserer Dokumentation

Die Zusammenfassung und die Einleitung sollen den Einstieg in die vorliegende Arbeit erleichtern und das Interesse des Lesers fördern. Der Ausgangspunkt stellt die Aufgabenstellung dar und das Inhaltsverzeichnis führt die verschieden Kapitel übersichtlich auf.

Die Kapitel 5 und 6 versuchen den Hintergrund darzustellen und unsere Arbeit in der Softwarelandschaft einzubetten.

Ein besonderes Augenmerk soll dem Kapitel 7 gegeben werden, da dort unsere Ideen erläutert, die Ziele definiert und die Umsetzung dokumentiert ist.

Für zukünftige Anwendung bzw. Erweiterungen ist der Ausblick gedacht. Im Anhang befinden sich dann die üblichen Dokumente wie das CD-Inhaltsverzeichnis, der Zeitplan und der Quellennachweis.

3 Aufgabenstellung

Kommunikationssysteme (KSy)
Projektarbeiten SS 2001 - PA2 Sna05
X.509 Trust Chain Unterstützung für den Linux IPSec Stack
Studierende:

- Marco Betrossa, IT3a
- Andreas Schleiss, IT3b

Termine:

- Ausgabe: Montag, 21.05.2001 13:30 - 14:30 im E523
- Abgabe: Freitag, 6.07.2001

Beschreibung:
Im Rahmen einer IT-Projektarbeit haben ZHW Studenten letztes Jahr für den Linux IPSec Stack einen Patch geschrieben, der die Verwendung von X.509 Zertifikaten unterstützt. Dadurch kann ein Linux Security Gateway mit den meisten Windows VPN Clients auf der Basis einer Public Key Authentisierung eine sichere Verbindung aufbauen und ermöglicht so kostengünstige Virtual Private Network Lösungen.

Der X.509 Patch hat weltweit sehr grossen Anklang gefunden und wurde schon gegen 1000 mal von der Web-Site http://www.strongsec.com/freeswan/ runtergeladen. Zu den Anwendern gehört z. Bsp. die Sibirische Bank in Nowosibirsk ;-)

In der momentanen Realisierung müssen alle X.509 Zertifikate der IPSec Clients lokal auf dem Linux Security Gateway gespeichert sein. Bei wenigen Verbindungen ist dies kein Problem. Bei Tausenden von Benutzern, z.Bsp. alle Studierenden, Dozierenden und Angestellten der ZHW, ist dieses Verfahren nur schwer skalierbar.

Die Projektarbeit soll folgenden Ansatz realisieren:

Praktisch alle Clients übermitteln ihr Zertifikat im Rahmen des Verbindungsaufbaus. Es soll nun die Gültigkeit der empfangenen Client Zertifikate überprüft werden, in dem die Unterschriften auf den X.509 Zertifikaten die Vertrauenskette hinauf bis zum Root CA Zertifikat verfolgt werden. Dadurch müssen lokal nur noch die CA Zertifikate und eine Certificate Revocation List mit den Serienummern der gesperrten Zertifikate gespeichert werden. Für die Realisierung dieser Funktionalität soll die GNU Multiprecision (GMP) Library verwendet werden.

Aufgaben:

- Einarbeiten in die Themen "IPSec Protokoll" und "X.509 Zertifikate".
- Studium der relevanten Stellen des FreeS/WAN Pluto Source Codes.
- Realisierung der X.509 Trust Chain Unterstützung:
    - Laden von Root und Intermediate CA Zertifikaten aus dem Verzeichnis "/etc/ipsec.d/cacerts".
    - Ueberprüfung der zeitlichen Gültigkeit der CA Zertifikate und der empfangenen
      Host Zertifikate.
    - Ueberprüfung der digitalen Unterschriften entlang der X.509 Trust Chain.
    - Laden von Certificate Revocation Lists aus dem Verzeichnis "/etc/ipsec.d/crls".
    - Ueberprüfung der CRLs auf Seriennummern von gesperrten Host Zertifikaten.
    - Nach erfolgreicher Überprüfung des empfangenen Host Zertifikates, Extraktion des RSA Public Key und Einfügen in die Public Key Liste von Pluto.
    - Unterstützung des ID Typs "DER_ASN1_DN" über den X.509 "Subject" Eintrag
    - Unterstützung der ID Typen "IPV4_ADDR" , "FQDN" und "USER_FQDN" über die X.509v3 Extension "SubjectAltName".
- Einhalten des FreeS/WAN "Coding Styles", alle Kommentare auf Englisch.
- Keine Verwendung von OpenSSL Source Code, sowie von Source Code Teilen, die in den USA oder durch US-Bürger erstellt wurde !!!

Infrastruktur / Tools:

- Raum: E523
- Rechner: 2 PCs
- SW-Tools: SuSE Linux, GNU C, GMP Library

Literatur / Links:

- Linux - FreeS/WAN Projekt
  http://www.freeswan.org
- X.509 Patch für Linux - FreeS/WAN
  http://www.strongsec.com/freeswan
- GMP - GNU Multi-Precision Library
  http://swox.com/gmp/
- OpenSSL Projekt
  http://www.openssl.org
- OpenSSL Handbuch
  http://www.pca.dfn.de/dfnpca/certify/ssl/handbuch/
- ZHW Projektarbeit "Linux - IPSec mit PGP und X.509 Zertifikaten"
  http://www.strongsec.com/zhw/PA/Sna03_2000.pdf
- ZHW Diplomarbeit zum Thema IPSec
  http://www.zhwin.ch/~sna/research/VPN/DA99/DA99_gaertner_uenal.pdf
- Relevante RFCs zum Thema IPSec
  http://www.zhwin.ch/~sna/research/VPN/ipsec.htm
- IETF RFC 2459
  Internet X.509 Public Key Infrastructure Certificate and CRL Profile
- ITU-T X.509
  The Directory: Authentication Framework

Winterthur, 21. März 2001

Dr. Andreas Steffen

4 Inhaltsvezeichnis

5 Kommunikationsweg

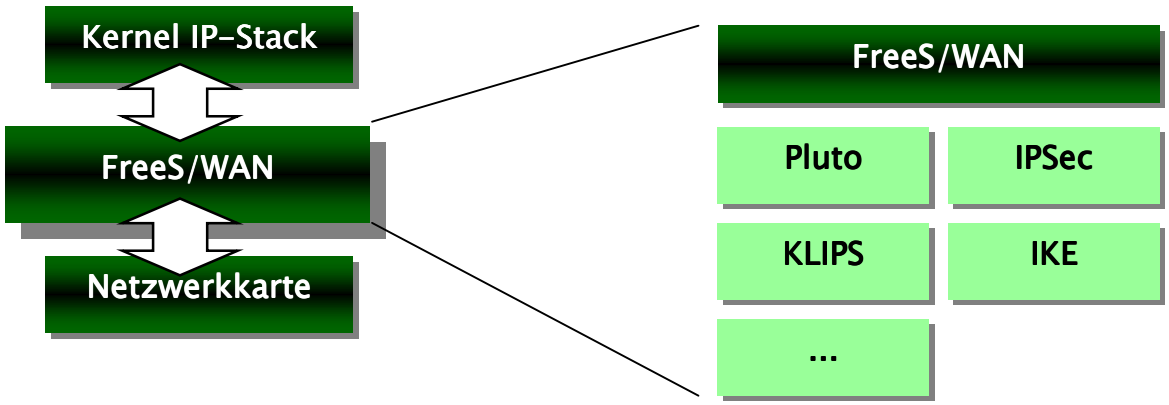
5.1 VPN-Theorie

Für eine sichere Verbindung zwischen lokalen Netzen (Intranets) kann ein virtueller privater Tunnel durch das Internet gelegt werden. Dieser virtuelle private Tunnel (VPN) gewährleistet eine abhörsichere und verschlüsselte Verbindung in den unsicheren Netzen.

Vor einigen Jahren genügten noch Telefone, bzw. Modems als Verbindungseinheit, damit war die Authorität der beteiligten Personen und die Sicherheit einer Verbindung gewährleistet. Durch das wachsende Datenvolumen wurde mit der Zeit der Kanal durch eine Telefonleitung zu schmal. Und wenn zur Kapazitätssteigerung mehrere Leitungen zusammengefasst werden oder gar eine SDH-Leitung gemietet wird, so ist dies um ein Vielfaches teurer als eine vergleichbare Verbindungskapazität durch das Internet. Aus diesem Grund versucht man durch eine geeignete Authorisierung und Verschlüsselung das Internet für solche VPN's zu nutzen. Es existieren verschiedene VPN-Protokolle, unsere Projektarbeit beschränkt sich auf die FreeS/WAN IPSec Implementierung.

5.2 FreeS/WAN

FreeS/WAN ist eine Softwarelösung, welche unter Linux den Aufbau eines VPN's durch das Internet ermöglicht. Sie besteht unter anderem aus den Elementen Pluto, IPSec, KLIPS und IKE, die in den nächsten Abschnitten näher erläutert werden sollen.



5.2.1 PLUTO

Der Pluto ist ein Dämon, welcher die VPN's Kanäle verwaltet. Es können bekanntlich mehrere verschiedene VPN's von demselben Host aufgebaut werden. Pluto stellt nun die Werkzeuge zum Aufbau und Unterhalt eines solchen Tunnels zur Verfügung.

5.2.2 IPSec

Das Ziel von IPSec [PA 00] ist eine sichere Kommunikation über das Internet zu gewährleisten. Das beinhaltet folgende Punkte:

- Integrität der Daten sicherstellen
- Authentifikation der Daten unterstützen
- Vertraulichkeit der Daten und der Verbindung gewährleisten
- Zugriffskontrolle auf Benutzerebene ermöglichen

Damit die Vertraulichkeit der Verbindung - dazu gehört die Anonymität des Senders und Empfängers - gewährleistet werden kann, muss IPSec schon beim Network Layer in den Protokollstack eingefügt werden. Würden die Daten erst weiter oben im Protokollstack von IPSec bearbeitet, könnte jeder Router, der an der Verbindung beteiligt ist, den Sender und den Empfänger einer Nachricht feststellen.
Der Network Layer im Internet Protokollstack wird in den meisten Fällen durch das Internet Protokoll IPv4 realisiert. IPSec hat also die Aufgabe, IPv4 um ein Sicherheitsprotokoll zu erweitern. Beim Internet Protokoll IPv6 wäre das nicht mehr notwendig, weil darin bereits ein Sicherheitsprotokoll integriert ist.
Bis zum Transport Layer hinunter ist IPSec völlig transparent. Als Sicherheitsprotokoll des Network Layers bietet es Schutz für alle Daten und Applikationen in den darüber liegenden Schichten, ohne dass die Applikationen angepasst werden müssen.

5.2.3 KLIPS

Der KLIPS (Kernel Level IP Security) erweitert den Linux-Kernel mit den IPSec-Modulen. Es fügt dem IP-Stack die speziell für die sichere und verschlüsselte Übertragung der Daten benötigten Zwischenschichten hinzu. Solche Sicherheitsschichten sind unter IPv4 nicht implementiert, deshalb müssen sie zusätzlich installiert werden. Danach ist es möglich, mit dem Internet Key Exchange (IKE) Protokoll eine Security Association (SA) aufzubauen.

5.2.4 Security Association (SA)

Die SA ist ein Vertrag zwischen den Endpunkten und beinhaltet folgende Abmachungen:

- Authentifikationsmechanismus
- Verschlüsselungsalgortihmus
- Hashalgorithmus
- Diverse Schlüssel für die Authentifikation und das Ver- und Entschlüsseln der Daten
- Gültigkeitsdauer der Schlüssel
- Zeit bis die SA erneuert werden muss

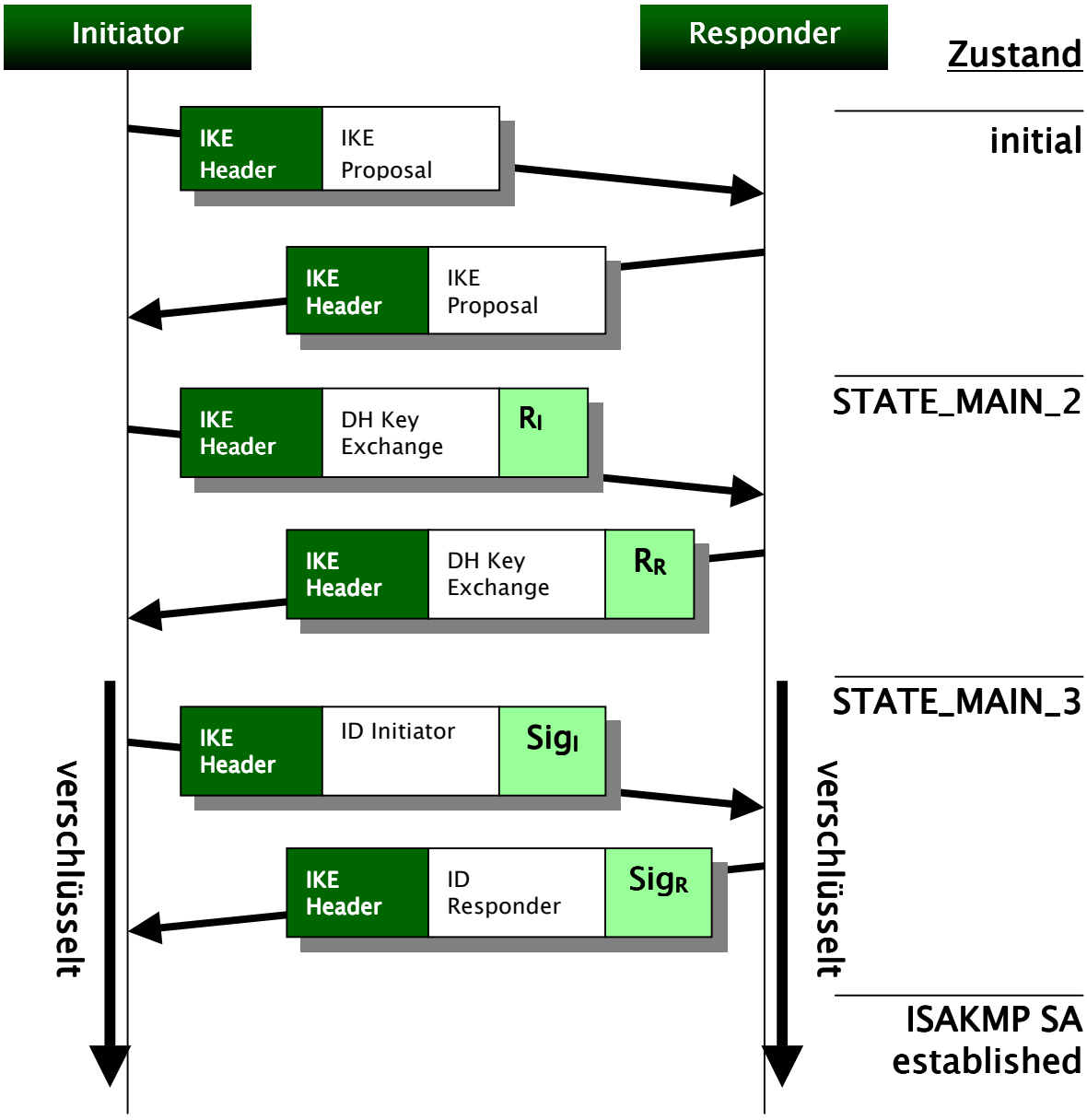
Diese Parameter werden jeweils bei jeder neuen Verbindung wieder ausgehandelt. Um die Sicherheit zu erhöhen können nach einer ebenfalls abgemachten Zeit die Parameter erneuert werden.

5.2.5 IKE-Verbindungsaufbau

Das Internet Key Exchange Protokoll [RFC 2409] beschreibt den gesamten Ablauf zur Erstellung einer IPSec SA (Security Authority). Dies beinhaltet unter anderem den Schlüsselaustausch, das Aushandeln und Verteilen der benötigten Parameter und Mechanismen. Der Verbindungsaufbau findet in 2 Phasen statt, die Phase 1 ist der Main Mode, die Phase 2 der Quick Mode.

5.2.6 IKE Phase 1- Main Mode

Um einen geschützten Tunnel durchs Internet aufzubauen braucht es geeignete Verschlüsselungsmethoden. Im Main Mode wird deshalb der Diffie-Hellman (DH) Key-Exchange Algorithmus eingesetzt. Dadurch beitzen im Anschluss an den Main Mode beide Kommunikationspartner einen Secret Key, für die weitere Kommunikation.



Die oben beschriebenen Mechanismen zur Sicherung des Parameteraustauschs stellt selbst eine SA dar, die im IKE Protokoll Internet Security Association and Key Management Protocol (ISAKMP) SA genannt wird. Der hier in der Grafik beschriebene Vorgang bis die ISAKMP SA steht, wird IKE Phase 1 - Main Mode genannt:

Beim Initialisieren einigen sich die Kommunikationspartner auf eine SA-Konfiguration. Ist dieses erfolgreich verlaufen, wird mit Hilfe des Diffie-Hellman Algorithmus ein Secret Key erzeugt. Dieser Secret Key ist nach dem Erreichen des Zustandes 3 (STATE_MAIN_3) beiden bekannt und damit kann der gesamte weitere Datenaustausch verschlüsselt stattfinden. Im letzten Teil des Main Mode generiert der Initiant und der Responder mit dem generierten Secret Key als Initialwert ein Hash über die Datenpakete. Beide signieren diesen Hash mit Ihrem Private Key (→ Sig_x) und senden diese dem anderen. Beide senden nun ihre ID und diese Signatur durch das Netz, welche die Gegenstelle einfach verifizieren kann. Optional kann auch ein X.509 Zertifikat mitgesendet werden welches den Public Key dem Initiator zuordnet und ihm auf diese Weise die benötigte Glaubwürdigkeit verleiht. Dieses wird aber erst durch unsere Arbeit komfortabel ausgewertet.

Dabei muss noch erwähnt werden, dass das Initiator-Zertifikat in jedem Schritt des Main Mode übertragen werden kann, dies aber meistens im letzten Schritt der Fall ist.

5.2.7 IKE Phase 2 – Quick Mode

Nach dem Abschluss der Phase 1 (Main Mode) ist die Authentizität der Hosts und die Vertraulichkeit der Verbindung bereits gewährleistet. Aus diesem Grund ist die Phase 2 nicht mehr so aufwendig - der Quick Mode baut auf der zuvor erstellten ISAKMP SA auf. Erst der Quick Mode setzt die eigentliche IPSec SA auf und handelt die verschiedenen Parameter für den AH, die ESP [PA 00], den Authentifikations- und Verschlüsselungsmechanismus etc. aus. Der Quick Mode kann während einer IPSec Verbindung wiederholt durchgeführt werden, um die Erneuerung der Schlüssel für die ISAKMP SA und die IPSec SA zu erzwingen. Damit die neuen Schlüssel in keiner Relation zu den vorher verwendeten Schlüsseln stehen, kann auch beim Quick Mode der Diffie-Hellman Key-Exchange Algorithmus angewandt werden, um neue Schlüssel zu erzeugen (Perfect Forward Secrecy).

5.2.8 Installation

Im folgenden soll die Installation und die Konfiguration grob aufgezeigt werden. Ein genauerer Beschrieb kann den vorausgegangenen Arbeiten [DA 99] / [PA 00] oder über verschiedene Links [WW ST] entnommen werden:

Die Linux-Distribution von SuSE (ab Version 6.3) enthält bereits das FreeS/WAN-Paket und im mitgelieferten Kernel ist IPSec bereits aktiviert, dadurch muss der Kernel (KLIPS) nicht neu kompiliert werden.

Nun sollte die aktuellste Version von FreeS/WAN hinuntergeladen und ins Verzeichnis `/usr/local/src/freeswan-1.9` entpackt werden. In unserem Falle wurde die Version 1.9.0 installiert.

Danach muss der Kernel neu kompiliert werden, damit die neue Version in den Kernel aufgenommen wird. Dies geschieht mit dem Befehl `make xgo` im oben erwähnten Verzeichnis.

Wichtig für die Hosts im Subnet ist, dass das IP-Forwarding auf dem Security-Gateway eingeschaltet ist. Dieser Eintrag kann in der Datei `/usr/etc/rc.config` oder mit Yast gemacht werden (`IP_FORWARD="yes"`).

Damit FreeS/WAN auch X.509-Zertifikate unterstützt, muss der x509patch-X.X-freeswan-1.9.tar Patch installiert werden [WW ST].

5.2.9 Konfiguration

5.2.9.1 ipsec.conf

Die Datei `/etc/ipsec.conf` [WW FD] enthält die Konfiguration für den FreeS/WAN Security-Gateway.

In einem ersten Teil, dem `config setup`, werden Basiseinstellung gesetzt. Hier wird bestimmt, welche Interfaces für die Kommunikation verwendet werden, ob Informationen gelogged werden sollen und wie sich Pluto verhalten soll.

```
# basic configuration
config setup
# THIS SETTING MUST BE CORRECT or almost nothing will work;
# %defaultroute is okay for most simple cases.
interfaces=%defaultroute
klipsdebug=none
plutodebug=all
plutoload=%search
plutostart=%search
uniqueids=yes
```

Beschreibung zu den hier gesetzten Variablen:

- interfaces:

Beschreibt die Zuordnung der physikalischen zu den virtuellen (gesicherten) IPSec-Interfaces. Im Normalfall genügt hier schon der Eintrag `%defaultroute`, wobei das konfigurierte Standardinterface verwendet wird. Wird das Interface explizit angegeben, so lautet der Eintrag z.B. `interfaces="ipsec0=eth1"`. Sollen mehrere Interfaces angegeben werden, so ist zu beachten, dass diese durch ein Leerzeichen voneinander getrennt sind.

- klipsdebug:
  Soll der KLIPS Debugging Output gelogged werden, so wird hier `all` angegeben. Andernfalls kann man diese Variable auf none setzen.

- plutodebug:
  Hier kann der Pluto Debugging Output mit `all` gelogged werden. Soll dies nicht geschehen, so bleibt der Eintrag leer oder wird mit none gekennzeichnet. Für Testzwecke wird diese Variable auf `all` für komplettes Logging gesetzt.

- plutoload:
  Hier kann dem Pluto mitgeteilt werden, welche Verbindungen er zur Startzeit laden soll. Wird die Variable, wie in diesem Falle gezeigt, auf `%search` gesetzt, so werden alle connections mit den Einträgen `auto=add` und `auto=start` geladen.

- plutostart:
  Wenn hier `plutostart="%search"` eingetragen wird, startet Pluto automatisch die Verbindungen, welche mit `auto="start"` gesetzt wurden.

Im zweiten Teil , dem `conn %default`, werden Werte gesetzt die in jedem conn Eintrag die gleichen sind. Falls in einem conn Eintrag ein schon in `%default` definierter Parameter eingetragen ist, wird automatisch der conn-Eintrag benützt.

```
conn %default
     keyingtries=3
     authby=rsasig
     right=160.85.131.57
     rightid=@ksy005.zhwin.ch
     rightrsasigkey=%cert
```

In einem dritten Teil werden die verbindungsspezifischen Vereinbarungen getroffen. Dabei kann für jede einzelne Verbindung eine eigene Vereinbarung konfiguriert werden.

```
# connections

conn server
      left=160.85.20.100
      leftnexthop=160.85.20.1
      leftid=@sna-gw.strongsec.com
      leftrsasigkey=%cert
      right=160.85.131.57
      rightnexthop=160.85.130.30
      rightid=@ksy005.zhwin.ch
      rightrsasigkey=%cert
      auto=add

conn ksy006
      left=160.85.131.58
      leftid=@ksy006.zhwin.ch
      leftrsasigkey=%cert
      auto=add

conn schleand
      left=160.85.131.58
      leftcert=ksy006Cert.pem
      right=160.85.131.57
      rightcert=ksy005Cert.pem
      auto=add

conn swiss-server
      left=160.85.20.100
      leftnexthop=160.85.20.1
      leftid=@sna-gw.strongsec.com
      leftrsasigkey=%cert
      auto=add
```

- authby:
  Dies ist die Authentifikationsart der beiden Kommunikationspartner. Mögliche Einträge sind `secret` für Pre-shared Secrets oder `rsasig` für digitale RSA-Unterschriften.

- left/right:
  Wir definieren hier eine gültige IP-Adresse des linken bzw. rechten Hosts oder Security-Gateways.

- leftnexthop/rightnexthop:
  Hier wird die Gateway IP Adresse für den rechten bzw. den linken Kommunikations-
  partner zum öffentlichen Netzwerk angegeben.

- leftid/rightid:

Dieser Eintrag legt die Identifikation für die Authentifizierung fest. Der Kommunikation – spartner wird anhand seiner IP-Adresse Identifiziert. Es kann anstelle der IP-Adresse auch ein gültiger Domainname mit einem führenden "@" eingetragen werden.

- leftrsasigkey/rightrsasigkey:
  Dieser Eintrag beinhaltet den Public Key für die RSA Signatur Authentifikation. Der X.509 – Patch ermöglicht die Eingabe %cert, womit der Pluto den Public Key direkt aus dem vom Client gesendeten Zertifikat heraus liesst.

- leftcert/rightcert:
  Hier wird explizit das Zertifikat angegeben, welches vom Pluto aus dem Verzeichnis /etc/ipsec.d geladen werden muss, um den Public Key herauszulesen. Bei diesem Verfahren wurde das Zertifikat noch nicht auf Gültigkeit überpüft.

- auto:
  Wir setzen die Variable `auto` auf `add` und teilen damit Pluto mit, dass er zur Startzeit diese Verbindung starten soll. Dies bedingt, dass die Variable `plutoload` auf `%search` gesetzt wurde, damit Pluto beim Start nach den Verbindungen mit diesem Eintrag sucht und diese startet.

Sämtliche Aktionen von FreeS/WAN können geloggt werden. Dazu müssen in der Datei `ipsec.conf` die Variablen `plutodebug` bzw. `klipsdebug` entsprechend gesetzt werden (siehe oben). Die Logdatei kann mit folgendem Befehl in eine Datei geschrieben werden:

```
ipsec barf > barf.txt
```

5.2.9.2 ipsec.secrets

Die Datei `/etc/ipsec.secrets` [WW FD] enthält die Konfigurationsparameter von FreeS/WAN, die geheim gehalten werden müssen. Bei der Authentisierung mit X.509 Zertifikaten wird hier der Private Key gespeichert.

```
: RSA {
      Modulus:
            0x9DBB9E1E4199C372CAD8819BC15B23AE716CA6135D27634E513472E
            60B4011447C907E0BF34E34422A410CD8F70D3A5C40C62C2FF459A589
            502D05B8B194B7CFC1B9D73703E95A452F9E0C9B4B471E7111C1DD47D
            AFD8435425DCBC57BD5A1F9C0A3CE42D63BD4D4D2D74523F629D32E49
            B91594187629B07833B47EB0E417A

      PublicExponent: 0x010001

      PrivateExponent:
```

```
        0x2B31C4D42E4CC85C3836600FE23C6E220847A3972BEC6C62771470D
        E947820026C46396E565BB52DE55A5905556A56F00A80FA7ACB647D53
        8414403BDACD8A2439A392B9A45545906494075DC81E04BB1E2E3E847
        80D6D03865905984FA4AD8253B2BD3F3D96F02DF25F79C363D7144C0B
        EE14D1DB4BE6DFAEAF4B6BC726B5C5
    Prime1:
        0xCEC5F6AD68B56D5E2C74AF5D2BD677C5A7D74EC93EF53EB0618D8D7
        2CDDA0FC7B4C902BC2498B05781F9398B224AB1A781241A44DD8FC54B
        EBE48473808A8E4B

    Prime2:
        0xC348D4619F70C72DA59168839C7803787DE1881BB8C6B655E2FCF2C
        65F928A9B5B29AD4CF0BD5965AC14BF2485D6F0FBCA29D8BF87D839B1
        0DAC00E299577BCF

    Exponent1:
        0x5A38DEEDC366869634E7A52D0E57C263285D362F719D2901654F928
        CA96DA0BD0F11367449B3A61E48C42A6B9F46E045AA7FCA468A36956C
        0F3DE2DD53152209

    Exponent2:
        0x7EAF396FD7321F54426B5124C815A712BC7ED9803A5F569BE2F3FB3
        EAB73724D945736208AB01B5574CBC4B019CF3567F25F31481705336F
        E391A8C5577AC44B
    Coefficient:
        0x921CD457451407210EEA6B58575E8136A419839B182B6B8BC958F16
        F41B147D06163FC1963B91BAA13FEFA2C95CB188ADC0FE7746589863A
        087852A84335FC45
}
```

Weil diese Datei geheime Informationen enthält, sollte sie nur durch den FreeS/WAN Administrator zugänglich sein und für alle anderen lese- und schreibgeschützt bleiben.

5.2.9.3 /etc/x509cert.der

In diesem File wird das vom Host verwendete Zertifikat gespeichert, welches bei einem Verbindungsaufbau dem Kommunikationspartner gesendet wird. Wie an der Endung zu erkennen ist, muss das Zertifikat im DER Format vorliegen [6.1]

6 Zertifikate

Wie in 5.2.5 IKE - Verbindungsaufbau beschrieben, sendet der Client bei einem Verbindungsaufbau ein X.509 Zertifikat, welches den Public Key diesem Client zuordnet und dieser Zuordnung die benötigte Glaubwürdigkeit verleiht.

Beim zweiten Zertifikat, das verwendet wird, handelt es sich um die sogenannte Certificate Revocation List (CRL). Dieses gehört zu einer bestimmten CA (Certification Authority) und enthält eine Liste der gesperrten Zertifikate dieser CA.

Im folgenden wird der Aufbau dieser beiden Zertifikate und deren wichtigster Felder kurz erläutert.

6.1 X.509 Zertifikat

Um die benötigte Glaubwürdigkeit eines X.509 Zertifikates zu erreichen, wird dieses von einer "Certification Authority" (CA) ausgestellt und unterschrieben.
Die CA bildet einen Hash über die gesamten Daten und verschlüsselt den Hash mit ihrem Secret Key. Der verschlüsselte Hash bildet die Signatur (Unterschrift) der CA und wird dem Zertifikat hinzugefügt. Die Signatur bestätigt die Richtigkeit der Daten.
Um die Gültigkeit eines Zertifikats zu überprüfen, muss die Signatur mit dem Public Key der CA entschlüsselt und mit dem Hash verglichen werden.
Eine CA kann ihr Zertifikat ebenfalls unterschreiben lassen. Dadurch entsteht eine Zertifizierungskette (Trust-Chain), an dessen oberster Stelle sich die Root CA befindet. Die Root CA unterschreibt ihr Zertifikat selbst. Jeder muss dieser Root CA vertrauen.

Ein X.509-Zertifikat ist folgendermassen aufgebaut (ASN.1 codiert):

```
Certificate ::= SEQUENCE {
     tbsCertificate TBSCertificate,
     signatureAlgorithm AlgorithmIdentifier,
     signatureValue BIT STRING
}

 TBSCertificate ::= SEQUENCE {
     version [0] EXPLICIT Version DEFAULT v1,
     serialNumber CertificateSerialNumber,
     signature AlgorithmIdentifier,
     issuer Name,
     validity Validity,
     subject Name,
```

```
        subjectPublicKeyInfo SubjectPublicKeyInfo,
        issuerUniqueID [1] IMPLICIT UniqueIdentifier OPTIONAL,
        subjectUniqueID [2] IMPLICIT UniqueIdentifier OPTIONAL,
        extensions [3] EXPLICIT Extensions OPTIONAL
    }
```

Definition der einzelnen Felder:

```
Version ::= INTEGER { v1(0), v2(1), v3(2) }
CertificateSerialNumber ::= INTEGER
Validity ::= SEQUENCE { notBefore Time, notAfter Time }
Time ::= CHOICE { utcTime UTCTime, generalTime GeneralizedTime }
UniqueIdentifier ::= BIT STRING
SubjectPublicKeyInfo ::= SEQUENCE { algorithm AlgorithmIdentifier,
subjectPublicKey BIT STRING }
Extensions ::= SEQUENCE SIZE (1..MAX) OF Extension Extension ::=
  SEQUENCE
  {
    extnID OBJECT IDENTIFIER, critical BOOLEAN DEFAULT FALSE,
extnValue OCTET STRING
  }
```

Nach dieser ASN.1 Definition eines X.509 Zertifikates werden die Zertifikate im Pluto "geparsed" [7.3.2.5]. Im folgenden sind die wichtigsten Felder des X.509 Zertifikates, die benötigt werden, um die Gültigkeit zu überprüfen, aufgeführt:

- tbsCertificate:
  Der gesamte Inhalt des Zertifikates ausser dem Algorithmus, mit welchem dieses Feld verschlüsselt wurde, und der Signatur.
- signature, signatureAlgorithm:
  Geben den Algorithmus an, mit dem das Zertifikat signiert wurde, MD5 oder SHA1
- issuer:
  Aussteller des Zertifikats (eine CA)
- validity:
  Gültigkeitsbereich des Zertifikats (Datum von/bis)
- subject:
  Distinguished Name des Inhabers des Zertifikats

Damit ein Zertifikat signiert werden kann, muss es zuerst verschlüsselt werden. Dies geschieht entweder mit dem RSA MD5 [RFC 1321] oder dem SHA1 von NIST/NSA [WW SH]. Der erzeugte "Message Digest" oder "Fingerprint" wird nun von der CA mit deren Private Key signiert.

Die x509-Zertifikate können in zwei verschiedenen Dateiformaten abgespeichert und verwendet werden:

- ".pem"-Format (BASE 64)

- ".der" - Format (ASN.1 mit Distinct Encoding Rules (DER) binär codiert)

Mit dem folgenden Befehl können ".pem" Dateien in ".der"-Formate umgewadelt werden, welche benötigt werden, um eine Verbindung aufzubauen, ausser die Zertifikate werden explizit in der Datei /etc/ipsec.conf [5.2.9] angegeben:

```
openssl x509 -in [Name_des_Zertifikates].pem -outform der
-out [Name_des_Zertifikates].der
```

Um ein Zertifikat antzschauen kann ebenfalls openssl verwendet werden:

```
openssl x509 -in [Name_des_Zertifikates].pem  -noout -text
```

6.2 CRL Zertifikat

Die Gültigkeit eines Zertifikates hat eine bestimmte Dauer. Sollte aber ein Zertifikat innerhalb dieser Zeitspanne dennoch als ungültig erklärt werden, muss dies publik gemacht werden. Hierfür wird die Certificate Revokation List (CRL) benötigt. Dies ist eine von der ausstellenden CA signierte Datenstruktur, ähnlich einem Zertifikat, welche eine Liste aller ungültigen Zertifikate, welche von dieser CA ausgestellt wurden, beinhaltet.
Jedes ungültige Zertifikat wird anhand seiner Seriennummer in dieser Liste eingetragen.

Eine CRL Datenstruktur ist folgendermassen aufgebaut (ASN.1 codiert):

```
CertificateList ::= SEQUENCE {
      tbsCertList TBSCertList,
      signatureAlgorithm AlgorithmIdentifier,
      signatureValue BIT STRING
}

TBSCertList ::= SEQUENCE {
      version Version OPTIONAL,
      signature AlgorithmIdentifier,
      issuer Name,
      thisUpdate Time,
      nextUpdate Time OPTIONAL,
      revokedCertificates SEQUENCE OF SEQUENCE {
            userCertificate CertificateSerialNumber,
            revocationDate Time,
            crlEntryExtensions Extensions OPTIONAL
      } OPTIONAL,
      crlExtensions [0] EXPLICIT
      Extensions OPTIONAL
}
```

Einzelne Felder wurden bereits im Abschnitt 6.1.509 Zertifikat beschrieben.

- tbsCertList:
  Der gesamte Inhalt des Zertifikates ausser dem Algorithmus mit welchem dieses Feld verschlüsselt wurde und der Signatur.
- signature, signatureAlgorithm:
  Geben den Algorithmus an, mit dem das Zertifikat signiert wurde, MD5 oder SHA1
- issuer:
  Aussteller der CRL (eine CA)
- thisUpdate/nextUpdate:
  Ausstelldatum, beziehungsweise Erneuerungsdatum der CRL
- userCertificate:
  Serienummer eines ungültigen X.509 Zertifikates

Wie ein X.509 Zertifikat, wird eine CRL zuerst verschlüsselt (MD5 oder SHA1) und danach mit dem Private Key der CA signiert. Somit kann überprüft werden, ob es sich um die richtige, sowie um eine gültige CRL handelt.

CRL-Zertifikate können wie X.509 Zertifikate in zwei verschiedenen Dateiformaten abgespeichert werden, haben aber eine ".crl" Endung. Beim Herunterladen aus dem Internet kann es sein, dass diese Zertifikate im falschen Format (pem) vorliegen. Bevor ein CRL – File gebraucht wird, sollte überprüft werden, ob die Datei im „der" – Format und nicht im „pem – Format vorliegt.
Ein Base64 codiertes „pem" – Format sollte folgendermassen aussehen, um in ein „der"-Format gewandelt werden zu können:

```
---------BEGIN CRL CERTIFICATE -----
MIIDkzCCAnugAwIBAgIBEzANBgkqhkiG9w0BAQQFADBCMQswCQYDVQQGEwJDSDEX
MBUGA1UEChMOc3Ryb25nU2VjIEdtYkgxGjAYBgNVBAMTEXN0cm9uZ1NlYyBSb290
IENBMB4XDTAxMDUyNjAxNTY0NVoXDTAyMDUyNjAxNTY0NVowQjELMAkGA1UEBhMC

                              ...

82v8sdqBQ+HWh08kI3LaANBA0HGvZqsf+9RpDqt+5PRftQ5AQp44jwschf/eq/2U
rGKzSnHqZpsfxLsuKZMxU5JWa2NQzo7rRJYKOfv0+TtUZfqBdc9g72U9LorE8HfM
M45K9d+J/w==
---------END CRL CERTIFICATE -----
```

Um nun eine DER formatierte Datei zu erhalten muss diese mit der Extension ".pem" mit dem folgenden Befehl umgewandelt werden:

```
openssl x509 -in [Name_des_Zertifikates].pem -outform der -out
[Name_des_Zertifikates].crl
```

7 Verifizierung

7.1 Analyse

7.1.1 Ziel

Das Ziel dieser Arbeit (Erweiterung) ist die Verifizierung des IPSec Clients mit Hilfe von X.509 Zertifikaten zu vervollständigen, indem deren Gültigkeit überprüft wird, bevor der Public Key extrahiert und in die Public Key Liste von Pluto gehängt wird.

Die CA Zertifikate und die CRLs (Certificate Revokation List) sollten beim Start von Pluto in den Speicher geladen werden, damit eine einfache und effiziente Überprüfung stattfinden kann.
Ein Zertifikat wird zur Authentifizierung zugelassen, wenn

- die zeitliche Gültigkeit nicht abgelaufen ist
- die Signatur stimmt, bis hinauf zum Root CA Zertifikat des Ausstellers
- es nicht gesperrt ist

Falls eine der oben genannten Anforderungen nicht zutrifft, soll der Verbindungsaufbau abgebrochen und eine Fehlermeldung ins Logfile geschrieben werden.

Die zeitliche Gültigkeit soll mit der aktuellen Systemzeit und den Einträgen im Zertifikat [7.3.2.1] verglichen werden.

Die Signaturen der Zertifikate entlang der Vertrauenskette sollen jeweils mit dem Public Key des im Feld ISSUER (Aussteller) angegebenen CA Zertifikates entschlüsselt und mit dem berechneten Hash des zu überprüfenden Zertifikates verglichen werden.

Als letzter Schritt soll in der zum CA Zertifikat gehörenden CRL nach diesem Zertifikat gesucht werden. Falls dieses gefunden wird, ist es gesperrt und kann zur Verifizierung des IPSec Clients nicht benützt werden.

Da die CRL vom CA Zertifikat unterschrieben wurde, kann diese Signatur ebenfalls überprüft und somit festgestellt werden, ob es sich um die richtige, beziehungsweise eine gültige CRL handelt.

Die Reihenfolge der Überprüfung soll nach der Gewichtung der Einfachheit und Aussagekraft der einzelnen Schritte gewählt werden.

7.1.2 Lösungsvarianten

7.1.2.1 Einlesen der Zertifikate :

Um eine schnelle und effiziente Verifizierung der gesendeten Zertifikate zu gewährleisten, werden die CA-Zertifikate dynamisch im Speicher verwaltet. Beim starten von Pluto sollen die lokal gespeicherten Zertifikate in den Speicher gelesen werden.

Als Lösungsvariante wurde eine Liste gewählt. Der Fokus fiel auf die Komplexität dieser Liste, einfach oder doppelt verkettet zu sein. Wir entschieden uns für ersteres, da die Listenfunktionen minimal gehalten sind. Es werden höchstens Elemente in die Liste gehängt, gelöscht oder alle gezählt.

Die Listenelemente beinhalten einen next-Pointer und einen Daten-Pointer, welcher je nachdem ob es sich um ein CRL- oder ein x509-Zertifikat handelt auf eine Datenmenge im Speicher zeigt. Diese wird geparst und mit dem entsprechenden Typ (x509cert_t, bzw crl_t) dargestellt.

Jeder Zertifikatstyp besitzt sein eigenes typengerechtes Element, einen dazugehöriger Anker und typenspezifische Funktionen.

7.1.2.2 Überprüfung der Zertifikate

Die Reihenfolge der einzelnen Überprüfungen wurde folgendermassen festgelegt:

1. Überprüfung der Zeitlichen Gültigkeit
2. Berechnung des Hashes über das Zertifikat und Entschlüsselung der Signatur
3. Vergleich dieser Werte
4. Überprüfung der Signatur der CRL, falls vohanden
5. Suchen nach der Seien Nummer des Zertifikats in der CRL

Falls die Schritte 1,3,5, nicht erfüllt werden, beziehungsweise die Überprüfung fehl schlug, werden die folge Schritte nicht mehr ausgeführt.

Der 1. Schritt benötigt am wenigsten Zeit und falls das Zertifikat abgelaufen ist, muss auch die Signatur nicht mehr geprüft werden (welches auch der rechen-intensivste Schritt ist).

Nachdem die Signatur berechnet und verglichen wurde, stellt sich die Frage, was geschehen soll, wenn zu einem CA Zertifikat keine CRL vohanden ist, oder die Signatur dieser nicht übereinstimmt ? Soll die Verbindung nicht aufgebaut werden,

beziehungsweise diesem X.509 Zertifikat, dessen Signatur korrekt ist, nicht vertraut werden ?

Es wurde entschieden, dass eine Verbindung aufgebaut wird, wenn die Signatur eines Zertifikates stimmt, auch wenn das CRL Zertifikat nicht vorhanden, oder dessen Signatur fehlerhaft ist.

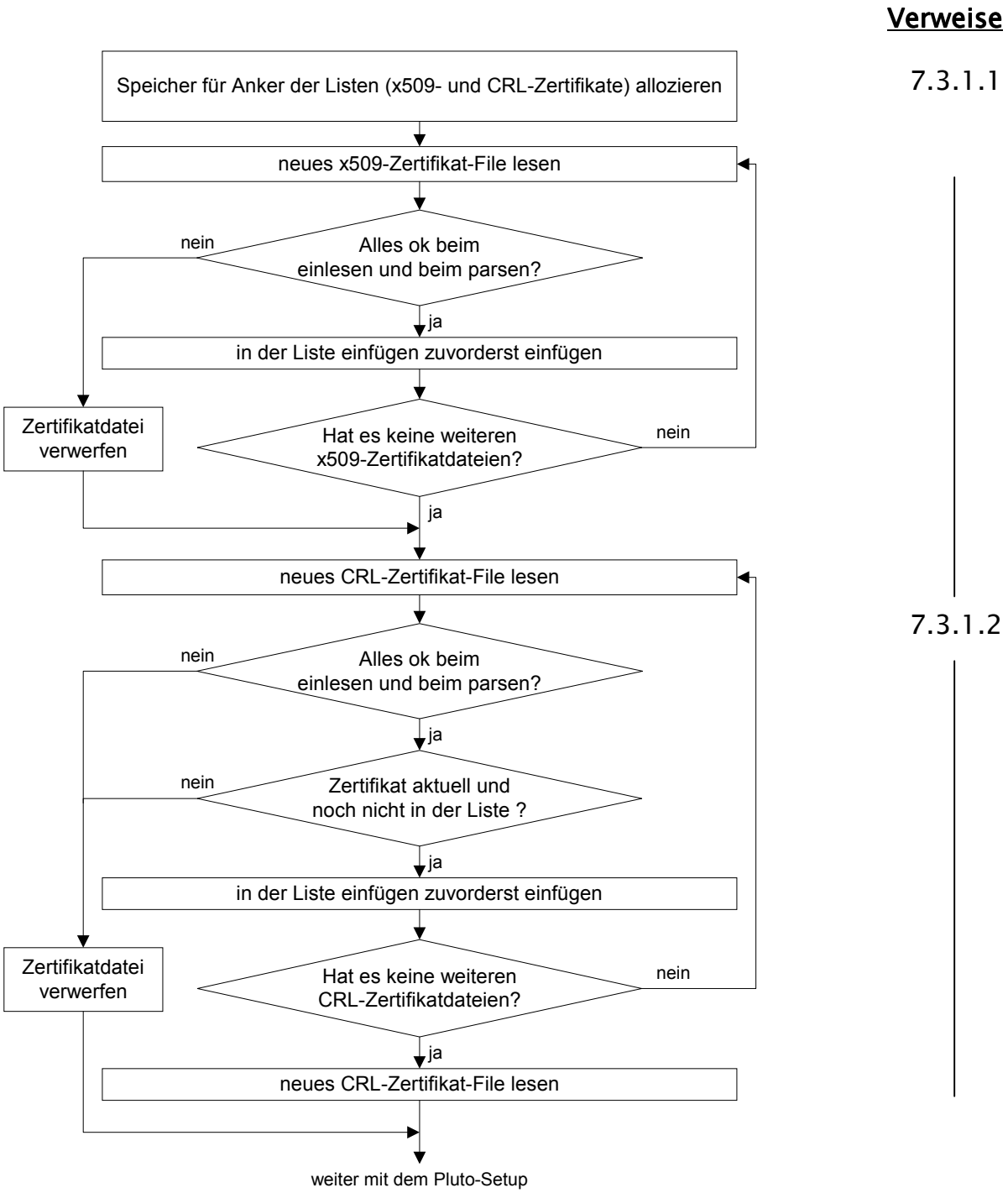
Der Grund für diese Entscheidung ist, dass eine CRL nur die aus einem bestimmten Grund gesperrten Zertifikate beinhaltet und nur dann einem Zertifikat nicht vertraut werden sollte, wenn es auch wirklich in einem CRL Zertifikat eingetragen ist.

7.2 Design

Diese Arbeit wurde in zwei Bereiche aufgeteilt:

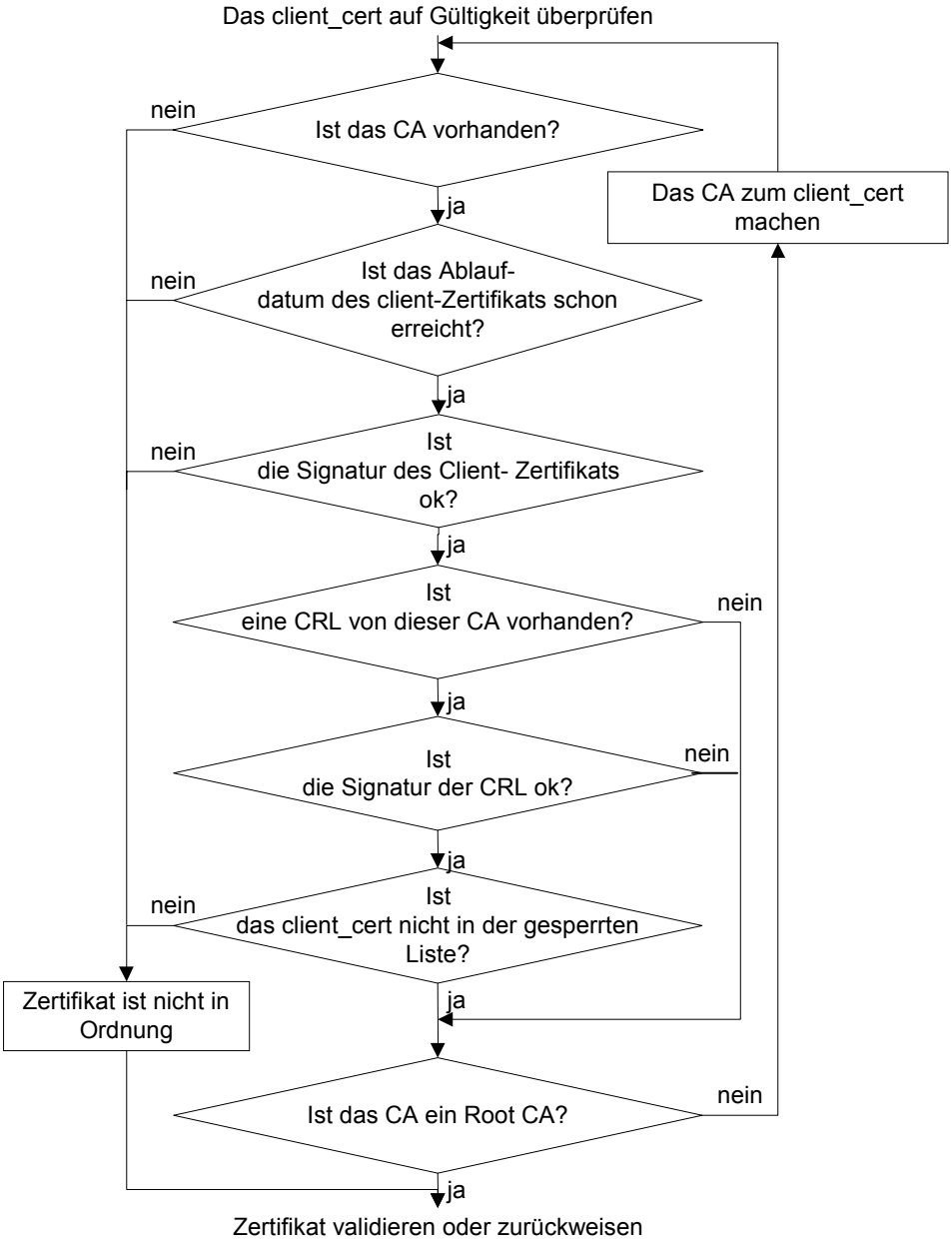
- Das Einlesen der CA Zertifikate und der CRLs. (verify – setup)
- Die eigentliche Verifizierung eines x509-Zertifikates mit Hilfe der Berechnung des Hashes und die Entschlüsselung der Signatur, sowie der Vergleich dieser beiden Werte. (verify – connection)

7.2.1 verify – setup



7.2.2 verify – x509 Cert

Verweise



7.3.2.1

7.3.2.3

7.3.2.5

Dem Projekt wurden zwei neue Files hinzugefügt, das `verify_x509.c` und das `verify_x509.h`. Das letztere bildet die äussere Schnittstelle zum restlichen Programm.

Um X.509 Zertifikate decodieren zu können, wurde eine "Parse" – Funktion zur Verfügung gestellt, welche in den Files `x509.c` und `x509.h` zu finden ist. Analog zu dieser, haben wir eine Funktion entwickelt, welche ein CRL Zertifikat decodiert. Um die schon gegebenen Strukturen, welche zum Parsen der X.509 Zertifikate benötigt werden, beizubehalten, wurde diese Funktion ebenfalls in das `x509.c` File geschrieben.

7.2.3 innere Schnittstellen

Zu Beginn werden alle CA-und CRL-Zertifikate in den Speicher geladen [7.3.1.1]. Die wichtigsten Funktionen für diese Aufgabe sind:

```
bool get_host_certs(void
bool putCert(FILE *cf,TElement **myanchor)
bool putCRL(FILE *cf,TElement2 **myanchor)
```

Um die beiden Bereiche miteinander zu verbinden, mussten zwei Funktionen definiert werden, mit denen das zum Überprüfen des Client Zertifikates benötigte CA Zertifikat [7.3.2], beziehungsweise CRL Zertifikat aus der Liste gelesen werden kann:

```
bool get_x509(x509cert_t **x509cert,chunk_t issuer)
bool get_CRL(crl_t **crl,chunk_t issuer)
```

7.2.4 äussere Schnittstellen

Die folgenden Funktionen werden von anderen Programmteilen aufgerufen, um entweder die Zertifikate einzulesen, ein Client Zertifikat überprüfen zu lassen oder die Zertifikatslisten aufzufrischen [7.3.3]:

```
extern bool init_host_certs(void)
extern bool verify_x509_public_key(const x509cert_t recv_cert)
extern bool reload_host_certs(void)
```

Weitere Funktionen die ausserhalb von verify_x509.c aufgerufen werden, sind die Funktionen, welche allozierten Speicher freigegeben [7.3.4], bevor der Dämon Pluto beendet wird:

```
extern bool freeMemory_CRL(void)
extern bool freeMemory_x509(void)
```

Die erwähnte "Parse" – Funktion [7.2.2] von CRL-Zertifikaten befindet sich im x509.c. Diese Funktion encodiert mit Hilfe von ASN.1 (Abstract Syntax Notation One) einen BER (Basic Encoding Rules) codierten Datenblock in die unten beschriebene crl_t Struktur:

```
extern bool parse_crl(chunk_t blob, crl_t *current_crl);
```

7.2.5 Datenstrukturen

Die BER codierte Datenstruktur muss so decodiert werden, damit man auf bestimmte Felder eines Zertifikates zugreifen kann. Hierfür wurde eine Datenstruktur definiert, welche die wichtigsten Felder einer CRL beinhaltet. Die einzelnen Felder sind von einem Typ `chunk_t`.

Dieser besteht aus einem Zeiger auf den Anfang einer Reihe von Daten im Speicher und der Länge dieser Daten. Somit kann mit beliebigen Datenkonstrukte gearbeitet werden.

Da ein Zertifikat als Block in den Speicher geschrieben wird, können mit der "Parse" – Funktion die einzelnen Felder eines Zertifikates innerhalb dieses Blockes gefunden werden und als Typ `chunk_t` dargestellt werden.

Struktur, welche ein CRL Zertifikat repräsentiert:

```
struct crl{
      chunk_t certificateList;
      chunk_t tbsCertList;
      u_int version;
      chunk_t sigAlg;
      chunk_t issuer;
      chunk_t thisUpdate;
      chunk_t nextUpdate;
      revokedCert_t revokedCertificates;
      chunk_t algorithm;
      chunk_t signature;
};
```

Da die wichtigen Daten einer CRL, die ungültigen Zertifikate, mehrmals vorkommen können [6.2], wurde eine weitere Datenstruktur definiert. Die ungültigen Zertifikate werden als Liste gespeichert, indem innerhalb der Struktur auf das nächste Element gezeigt wird.

Listenelement eines ungültigen Zertifikates:

```
struct revokedCert{
      chunk_t userCertificate;
      chunk_t revocationDate;
      struct revokedCert *next;
};
```

Als Element der Zertifikate wurde eine gut ausbaufähige Listenstruktur gewählt. Das x509-Zertifikat-Element ist folgendermassen definiert:

```
typedef struct element {
```

```
        struct element *next;
        x509cert_t *data;
    } TElement;
```

mit dem Anker

```
    TElement *anchor_x509;
```

Das CRL-Zertifikat-Element ist folgendermassen definiert:

```
    typedef struct element2 {
        struct element2 *next;
        crl_t *data;
    } TElement2;
```

mit dem Anker

```
    TElement2 *anchor_CRL;
```

7.3 Implementation

In diesem Abschnitt wird der Ablauf der verschiedenen Überprüfungen anhand der obigen Flussdiagramme [7.2.1]/[7.2.2] beschrieben und wichtige Segmente des Codes genauer erläutert.

7.3.1 Einlesen der Zertifikatdateien

Beim Setup des Plutos wird die Funktion init_host_certs() [7.3.1.1] aufgerufen. Diese ist im der Datei `verify_x509.c` abgelegt wie auch alle weiteren von uns geschriebenen Funktionen, ausser der Funktion `parse_crl()`, welche sich im `x509.c` befindet.

Die in den Abschnitten 7.3.1.x beschriebenen Abläufe und Funktionen sind für x509- und CRL-Zertifikate identisch, deshalb beschränken wir uns auf das Erstellen der x509-Liste. Die Namensgebung bei den CRL-Zertifikaten werden mit folgender Syntax: (CRL: ...) aufgeführt.

7.3.1.1 Initialisierung

Für die Anker der Listen [7.2.5] wird zuerst Speicher alloziert. Diese Anker zeigen entweder auf NULL (`anchor_x509->next = NULL`) (CRL: `anchor_CRL->next`) oder auf das erste Element in der entsprechenden Liste. Anschliessend werden die x509-Zertifikate, welche im Verzeichnis `[x509CertPath]="/etc/ipsec.d/cacerts"` (CRL: `[crlCertPath]="/etc/ipsec.d/crls"`) abgelegt sind, eingelesen. Mit dem Aufruf `loadfiles()` werden alle Dateien mit der Extension „.der“ (CRL : „.crl“) geladen.

7.3.1.2 Einbinden eines CA-Zertifikates

Ist ein „.der"-File vorhanden, wird die Funktion `putCert()` (CRL: `putCRL()`) aufgerufen. Als erstes wird Speicherplatz für ein neues x509-Zertifikat (`x509cert_t`) alloziert. Die Liste für diesen Zertifikatstyp besteht aus Elementen der Struktur `TElement`, diejenige für CRL-Zertifikate aus `TElement2`.

Wird ein neues Zertifikat in die Liste gehängt, so geschieht dies indem ein neues Element am Anfang der Liste hineingehängt und sein `data-Pointer` neu auf einen Speicherbereich zeigt, welcher das geladene Zertifikat beinhaltet. Der `chunk_t subject` wird gefüllt mit den Daten des Inhabers und der `next-Pointer` zeigt auf das nächste Element.

```
...
i = fread(blob.ptr,sizeof(char),file_len,cf);
...
if (parse_x509cert(blob, curEptr->data))
{
     ...
     /* get the new Cert on first place */
     curEptr->next = (*myanchor)->next;
     (*myanchor)->next = curEptr;
     ...
```

Um das Laden eines Dublikates zu vehindern, wird die Liste nach einem Element des selben Inhabers (`subject`) abgesucht und sofern eines gefunden wurde, gelöscht.

```
     ...
     /* looking for a similar cert */
     tmpEptr = (*myanchor)->next;
     curEptr = (*myanchor)->next->next;
     for (i=0;i < nCerts;i++)
     {
          if (!(memcmp((*myanchor)->next->data->subject.ptr,
               curEptr->data->subject.ptr,
               curEptr->data->subject.len)))
          {

               /*past->next = present->next*/
               tmpEptr->next = curEptr->next;
               DBG_log("there is a similar Certificate\n");

               pfreeany(curEptr->data);
               pfreeany(blob.ptr);
               free(curEptr);
               break;     //Schleife beenden
```

```
        }
        else
        {
            tmpEptr = tmpEptr->next;        //past++
            curEptr = curEptr->next;        //present++
        }
    }
    retValue = TRUE;
    ...
```

Der `barf` extrahiert alle Messages, welche mit dem Pluto-Dämon zu tun hat, aus der Datei `/var/log/message`. Diese können mit dem Befehl `barf > barf.txt` in eine Textdatei kopiert werden. Als letzte Meldung im `barf` beim setup sollte die Anzahl der geladenen Zertifiakte sein.

Ein Auszug aus der Datei `barf.txt` beim Befehl `ipsec setup start`:

```
... | \012******************************\012
... | *   START getting Files        *\012
... | ******************************\012\012\012
... | Change the Directory to  = /etc/ipsec.d/cacerts\012
... | Current Working Directory = /etc/ipsec.d/cacerts\012
... | Number of files = 3\012
... | Swisskey_Personal_ID_CA.der \012
... | size of the file: 637\012
... | L0 - certificate:
    ...
... | end of getting x509-Certificicates\012
... | counting Elements in the x509-List: 3\012
... | counting Elements in the CRL-List: 3\012
... | ******************************\012
... | *   END getting Files         *\012
... | ******************************\012\012\012
    ...
```

Für jede Datei wird dementsprechend Speicher alloziert und sollte deshalb beim Beenden von Pluto wieder freigegeben werden, dies geschieht in den Funktionen `freeMemory_x509()` und `freeMemory_CRL()`. Sie prüft zuerst, ob überhaupt etwas alloziert wurde, falls ja wird dieser Speicherplatz wieder freigeben.

Um nicht jedensmal den Pluto neu zu starten wenn neue Zertifikatsdateien existieren, haben wir die Funktion `relead_host_certs()` implementiert. Sie macht nichts anderes als zuerst beide Listen zu löschen, den Speicherplatz freizugeben um anschliessend wieder die Listen neu zu erstellen. Falls keine Dateidublikate vorhanden waren, werden alle Dateien in die Liste hineingehängt.

Als Anmerkung zur Identifikation der Zertifikate: es werden jeweils nur „issuer"-Felder und „subject"-Felder als Identität genommen. Das Seriennummer-Feld oder sonstige Extension-Felder werden nicht unterstützt.

Als Kontrolle wird am Ende mit Hilfe der Funktion countcert() (CRL: countCRL()) die Anzahl der Listenelemente gezählt und ausgegeben.

7.3.2 Verifizieren eines empfangenen Zertifikates

Als erstes wird die Gültigkeit des Datums validiert [7.3.2.1], die Information darüber finden sich in den Feldern `notBefore` und `notAfter` des X.509 Zertifikates . Falls dieses abgelaufen ist, wird, sofern das Logging aktiviert ist, eine Warnung ausgegeben und FALSE zurückgegeben.

Als nächstes wird die Signatur überprüft [7.3.2.3]. Das Zertifikat des Unterzeichners wird in der x509-Liste gesucht. Ist es vorhanden, sollte die Funktion `check_Signature` abklären, ob die Signatur in Ordnung ist.

Ist dies der Fall, wird das dazugehörende CRL-Zertifikat gesucht [7.3.2.5]. Dieses wird ebenfalls auf Gültigkeit (Signatur) überprüft. Ist dies nicht der Fall, wird eine Warnung ausgegeben (Debug Mode), das Zertifikat jedoch akzeptiert (siehe Lösungsvarianten 7.1.2).

Weist das CRL-Zertifikat die geforderte Gültigkeit aus, so wird die „Revocation List" nach der x509-Zertifikats-ID des x509Zertifikates vom Client durchsucht. Wird es gefunden, ist das Zertifikat gesperrt und der Verbindungsaufbau scheitert.

Das in unserer Liste gefundene CA-Zertifikat könnte von einer Intermediate Certificate Authority stammen. Wenn dies der Fall ist, so muss die Trust-Chain bis zum Root-Zertifikat entlang geprüft werden, ob alle Zertifikate ihre Gültigkeit noch besitzen. Der Ablauf ist nochmals derselbe wie der Verifizierungsprozess des ursprünglichen Client-Zertifikates.

Im Normalfall existiert immer ein CRL-Zertifikat einer Intermediate- oder Root-CA. Wenn dies allerdings nicht der Fall sein sollte, so führt dies lediglich zu einer Meldung, dass kein CRL-Zertifikat in der Liste gefunden wurde. Eine aktuelle CRL-Datei kann von den jeweiligen Homepages der CA's hinuntergeladen werden.

7.3.2.1 Überprüfung der zeitlichen Gültigkeit

```
bool check_validity_x509(const x509cert_t *client_cert){
```

Die zeitliche Gültigkeit des Zertifikates wird überprüft, indem die Felder `notBefore` und `notAfter` mit dem aktuellen Datum verglichen werden. Zuerst muss der Typ der beiden X509 Felder ermittelt werden, da ASN.1 zwei unterschiedliche Zeit-Typen kennt:

- UTCTime (Format: YYMMDDHHMMSSZ)
  Dies ist ein Standard ASN.1 Typ welcher die Jahreszahl mit den zwei letzten Ziffern angibt. Falls YY grösser oder gleich 50 ist, entspricht dies 19YY, andernfalls, YY kleiner als 50, wird das Jahr als 20YY interpretiert.

- GeneraliezedTime (Format: YYYYMMDDSSMMSSZ)
  Ebenfalls ein Standard ASN.1 Typ, der gebraucht wird, um die Jahre grösser als 2049 anzugeben [SNA ASN].

Mit der Abfrage nach der Länge der Felder `notBefor, notAfter` wird geprüft, um welchen ASN.1 Typ es sich handelt (`GeneraliezedTime = 15, UTCTime = 13`). Einer `UTCTime` wird immer die beiden ersten fehlenden Ziffern angehängt, damit eine Subtraktion (siehe unten) das korrekte Resultat ergibt.

Die aktuelle Zeit und die Felder, `notAfter` bzw. `notBefore` werden in Strings umgewandelt, damit ein Vergleich mit der Funktion `int strcmp(const s1 *char, const s2 *char)` durchgeführt werden kann. Diese Funktion gibt folgende Werte W zurück:

- W = 0, wenn s1 und s2 identisch sind
- W < 0, wenn die Subtraktion (als unsigned char) s1 – s2 < 0 ist
- W > 0, wenn die Subtraktion (als unsigned char) s1 – s2 > 0 ist

Somit können die Daten einfach verglichen und festgestellt werden, ob das Zertifikat zeitlich gültig ist.

7.3.2.2 Berechnung des Hashes über das Zertifikat

```
chunk_t encrypt_cert(chunk_t tbs, int algorithm){
```

Der Hash eines Zertifikats kann mit einem von zwei Algorithmen, entweder dem MD5 [RFC 1321] oder dem SHA-1 [WW SH], berechnet werden. Beide Algorithmen arbeiten im Wesentlichen auf die gleiche Art.

Ein Dokument wird in Blöcke mit exakt 512 Bit Grösse aufgeteilt, nachdem dem Dokument eine 64 Bit Dokument – Länge, zusammen mit einem Padding, um eine ganze Anzahl von 512 Bit Datenblöcken zu erhalten, angefügt wird.

Von jedem Datenblock wird nun der Hash mit dem entsprechenden Algorithmus berechnet, wobei dieser dem nächsten Block angehängt wird, bis beim letzten Block dann der endgültige Hash berechnet wird. Dem ersten Block wird ein sogenannter Initialisierungs Vektor mit der der selben Länge wie die des Hashes hinzugefügt.

Ein Unterschied dieser beiden Algorithmen ist die Länge des berechneten Hashes:

- MD5 Hashlänge: 128 Bit (16 Byte)
- SHA-1 Hashlänge: 160 Bit (20 Byte)

Am Beispiel des MD5 Algorithmus, wird im folgenden der Code zur Berechnung des Hashes über ein Zertifikat beschrieben. Es werden jeweils die von den Entwicklern der Algorithmen zur Verfügung gestellten Implementationen verwendet.

Als erstes muss anhand eines der Zertifikat Felder `signature` oder `signatureAlgorithm` der zu benützende Algorithmus bestimmt werden.

```
if(algorithm == MD5_ALG){
 use md5 */
      MD5_CTX context;
      MD5Init (&context);
      MD5Update (&context, tbs.ptr, tbs.len);
      MD5Final (digest, &context);
      hash.len = MD5_LEN;
      hash.ptr = alloc_bytes(hash.len,"computed hash");
      memcpy(hash.ptr,&digest,hash.len);
      DBG_cond_dump(DBG_CRYPT,"Encypted Certificate:md5 ",
      hash.ptr, hash.len);
      return hash;
}
else if(algorithm == SHA1_ALG){

      ...
```

Zuerst muss ein MD5 Context deklariert werden. Ein solcher Context enthält einen Datenblock a 64 Bytes (512 Bits), den jeweiligen Hash des vorherigen Datenblockes, sowie die Anzahl Datenblöcke.

Die einzelnen Schritte des MD5 Algorithmus:

- MD5Init()
  Hier wird der Initial Vektor und die Anzahl Datenblöcke auf 0 initialisiert.

- MD5Update()
  Dieser Funktion wird das Zertifikat übergeben, beziehungsweise der Zeiger auf den Anfang des Datenblocks im Speicher und dessen Länge.
  Nachdem die Anzahl der Datenblöcke berechnet wurde, werden die Hashes aller Datenblöcke mittels der internen Funktion MD5Transform berechnet.

- MD5Final()
  Schliesst die Berechnung des Hashes ab, indem die verbliebenen Daten Bytes mit einem Padding zu einem letzten 64 Byte (512 Bit) grossen Datenblock aufgefüllt werden und die Funktion `MD5Update()` aufgerufen wird.
  Der endgültige Hash wird dem Byte Array `digest` übergeben.
  Bei der Verwendung des SHA-1 Algorithmus werden die gleichnamigen Funktionen analog zu MD5 eingesetzt.

Die Länge des von der Funktion `encrypt_cert()` zurückgegebenen Hashes beträgt je nach Algorithmus entweder 16 Byte (MD5) oder 20 Byte(SHA-1).

7.3.2.3 Entschlüsseln der Signatur eines Zertifikates

```
chunk_t decrypt_signature(const x509cert_t *server_cert,
                          chunk_t signature)
```

Das Entschlüsseln einer Signatur geschieht mittels dem Public Key desjenigen, der mit seinem Private Key den Hash verschlüsselt hat. Dieses System beruht auf dem RSA Public Key Cryptosystem [PK 05].
Der Public Key beinhaltet den Modulus n und den öffentlichen Exponenten e, welche in der Funktion $f(x) = x^e \bmod n$ eingesetzt werden um die Signatur x zu entschlüsseln.

Da es sich um eine Funktion handelt, welche präzise Ergebnisse fordert und mit grossen Zahlen rechnet , wird auf eine spezielle Bibliothek, der sogenannten GMP - GNU Multi-Precision Library [WW GP] zurückgegriffen.

Um eine Funktion dieser Bibliothek auszuführen, müssen die verwendeten Operanden in einen speziellen GMP Typ umgewandelt werden. Danach kann die Funktion aufgerufen und das erhaltene Resultat wieder zurück gewandelt werden, in unserem Fall ein Byte Array.

```
n_to_mpz(c, signature.ptr, signature.len);
n_to_mpz(e, server_cert->publicExponent.ptr,
            server_cert->publicExponent.len);
n_to_mpz(n, server_cert->modulus.ptr,
```

```
            server_cert->modulus.len);

    /* compute the hash: hash = c^e mod n,
    variable c is used for in - and output*/
    mpz_powm(c, c, e, n);

    /* back to bytes */
    hash_val = mpz_to_n(c, server_cert->modulus.len);

    /* free memory */
    mpz_clear(c);
    mpz_clear(e);
    mpz_clear(n);
```

Die hier benützte GMP – Funktion

```
mpz_powm (mpz_t rop, mpz_t base, mpz_t exp, mpz_t mod)
```

berechnet das Resultat der Funktion $f(x) = x^e \bmod n$, wobei hier die Variable der Basis mit dem Resultat überschrieben wird.
Die Umwandlung zurück in ein Byte Array wird durch eine zur Verfügung gestellte Funktion `mpz_to_n()` durchgeführt.

7.3.2.4 Vergleich der berechneten Werte

```
bool check_signature(const x509cert_t *server_cert,
                     const x509cert_t *client_cert,
                     const crl_t *current_crl)
```

Dieser Funktion kann entweder ein X.509 Zertifikat oder ein CRL Zertifikat übergeben werden. Somit wird zuerst geprüft, welches Zertifikat überprüft werden soll. Nachdem der Hash berechnet, beziehungsweise die Signatur entschlüsselt wurde, wird das Padding der entschlüsselten Signatur überprüft, bevor die eigentlichen Werte verglichen werden.

```
...

/* Padding Test */
if (*decrypted_sig.ptr != 0x00)
     DBG_log("no leading 00");
else if (*(decrypted_sig.ptr + 2) == 0x01)
{
     const u_char *p;
     /* just the first 8 padding bytes are checked */
     for (p = decrypted_sig.ptr+3; p != decrypted_sig.ptr + 8;
p++)
     {
          if(*p != 0xFF)
```

```
                {
                     DBG_log("invalid Padding String");
                     break;
                }
           }
      }
      else if (*(decrypted_sig.ptr + 2) == 0x02)

           ...

      else
      DBG_log("Block Type not 01 or 02");

      ...
```

Da das Padding nur für debugging Zwecke ausgewertet wird, genügt eine Überprüfung der relevanten ersten 3 Bytes, welche den Inhalt des Paddings definieren. Dies wird anhand der nächsten 8 Bytes getestet. Das Padding variert je nachdem welche Informationen der DidgestInfo (siehe unten) beigefügt wurden, oder welche Zusatzinformationen der Zertifikat Aussteller zusätzlich hinzufügt.

Der eigentliche Vergleich der beiden Hash – Werten erfolgt am Schluss dieser Funktion, anhand eines Speichervergleiches. Hierfür wird ein Zeiger an den Anfang des Hashes gesetzt, je nach Algorithmus werden 16 oder 20 Bytes vom Ende des Datenblockes subtrahiert.

Beispiel einer entschlüsselten Signatur (Länge 257 Bytes), dessen Daten mit dem MD5 Algorithmus "gehashed" wurden:

```
00:00:01:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF
FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF
FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF
FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF
FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF
FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF
FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF
FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF
FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF
FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF
FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF
FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF
FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF
FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF:FF
00:30:20:30:00:0C:06:08:2A:86:F7:0D:02:05:00:04
10:D8:B7:B1:63:57:3B:33:1F:54:76:D7:B5:ED:18:7C
3A
```

Diese entschlüsselte Signatur setzt sich folgendermassen zusammen:

- Padding

- Didgest Info

Die Didgest Information ist ein ASN.1 codierte Sequenz, die den benützten Algorithmus identifiziert und den entschlüsselten Hash Wert beinhaltet, welcher zum Vergleich extrahiert wird.

```
DigestInfo ::= SEQUENCE {
     digestAlgorithm AlgorithmIdentifier,
     digest OCTET STRING
     }
```

Die BER Codierung (T) und die dazugehörigen ASN.1 Object Identifier (OID) [PK 01] / [PK 07], welche zum signieren von Zertifikaten verwendet werden, MD5 bzw. SHA-1 sind vordefiniert. H steht für den jeweiligen Hash Wert:

- MD5:

```
   T = 30 20 30 0c 06 08 2a 86 48 86 f7 0d 02 05 05 00 04 10 || H
OID = OBJECT IDENTIFIER ::= {iso(1) member-body(2) us(840)
                            rsadsi(113549) digestAlgorithm(2) 5}
```

- SHA-1:

```
   T = 30 21 30 09 06 05 2b 0e 03 02 1a 05 00 04 14 || H.
   OID = OBJECT IDENTIFIER ::= {iso(1) identified-organization(3)
                                oiw(14) secsig(3) algorithms(2) 26}
```

7.3.2.5 Parsen eines CRL Zertifikats

```
bool parse_crl(chunk_t blob, crl_t *current_crl)
```

Mit Hilfe existierender Funktionen wurde eine "Parse" – Funktion für ein CRL Zertifikat geschrieben. Anhand der BER Codierung des Datenblockes und der Zertifikat Struktur in Form der ASN.1, kann der Datenblock sequentiell abgearbeitet werden.

Bei der BER Codierung wird immer zuerst die Bezeichnung des folgendenTyps, beziehungsweise der folgenden Struktur angegeben, gefolgt vom Längenfeld (ausser es handelt sich um eine sogenannte "indefinite form", wo keine Länge angegeben wird, sondern auf das End – Tag gewartet wird).

Auf diese Gegebenheit beruht die "Parse" – Funktion, indem das "Identifier Octet" gelesen wird und in einer Tabelle nachgeschaut wird, ob an dieser Stelle ein solcher Typ, oder eine solche Struktur im Zertifikat vorkommt. Falls ja, wird dieser Byte String identifiziert und zurückgegeben und in der bereits beschriebenen CRL –

Struktur [7.2.5] gespeichert. Der Zeiger springt anhand der Längenangabe im Längenfeld zum nächsten Eintrag im Datenblock.

```
if (!extract_object(blobs, loopAddr, &objectID, &object,
                    PARSE_CRL))
  return FALSE;

  switch (objectID) {
     case CRL_OBJ_CERTIFICATE_LIST:
       current_crl->certificateList = object;
       break;

     ...

     case CRL_OBJ_REVOCATION_DATE:
       {
       /* put all the serial numbers and the revocation
       * date in a list starting in the struct crl_t with
       * the revocedCertificates as anchor
       */
       revokedCert_t *next_element =
          revokedCert_t*)malloc(sizeof(revokedCert_t));
       next_element->userCertificate = userCertificate;
       next_element->revocationDate = object;
       next_element->next =
                    current_crl->revokedCertificates.next;
       current_crl->revokedCertificates.next = next_element;
       }
       dntoa(buf, BUF_LEN, object);
       DBG(DBG_PARSING,
       DBG_log(" '%s'",buf);
       )
       break;
     case CRL_OBJ_ALGORITHM:

     ...

     default:
     }
  objectID++;
}
return TRUE;
```

Da eine CRL eine nicht vorhersagbare Anzahl von ungültigen Seriennummern von X.509 Zertifikaten beinhalten kann, wurde dieser Eintrag in der CRL – Struktur als Liste implementiert. Wie schon erwähnt wurde, ist die Reihenfolge durch die ASN.1 Definition eines Zertifikates gegeben, somit kann immer nach dem Parsen des `RevocationDate` das neue Element vom Typ `revokedCert_t` [7.2.5] in die Liste eingefügt werden. Der Anker dieser Liste ist das Element `revokedCertificates` in

der Struktur `crl_t` [7.2.5] selbst. Ein neues Element wird immer am Anfang der Liste, sprich beim Anker eingefügt. Dies spart unnötiges Traversieren der Liste.

Die Funktion `extract_object()` gibt ein gefundenes Objekt zurück, welches zu einem CRL Zertifikat gehört. Falls der Eintrag in einem "identifier octet" nicht zu einem Objekt aus dem CRL Zertifikat passt, wird die "Parse" – Funktion verlassen und FALSE zurückgegeben.

Wird das Zertifikat angenommen, beinhaltet die Datei „barf.txt“ unter anderem die folgenden Zeilen:

```
..."schleand" #1: +++++++++  starting verify x509 +++++++++\012
...| get_x509 ok\012
...|Encypted Certificate:md5
...|   c4 34 ed 0b  78 89 e0 9e  11 27 f3 e6  00 11 bc 4d
...| decrypted SIG in decrypt_signature()
...| Check CRL-Signature ok\012
...| Check CRL ok\012
...| reached RootCA\012
..."schleand" #1: ************* verify x509 ok ************\012
```

7.3.3 Listen neu erstellen

Es kann vorkommen, dass neue Zertifikate hinzukommen, welche nach einer gewissen Zeitspanne oder auf Kommando ebenfalls im Speicher vorhanden sein sollten. Eine Funktion, welche beide Listen auffrischt ist vorhanden, jedoch wurden die zugehörigen Aufrufe im restlichen Quellcode noch nicht implementiert und sind für spätere Zwecke vorgesehen. Die Funktion reload_host_certs() verwendet nur bereits bestehende Funktionen. Sie gibt den allozierten Speicher frei und führt die Funktion `loadfiles()` nochmals aus.

7.3.4 Pluto beenden

Wird der Pluto-Dämon beendet, so sollte auch der allozierte Speicherplatz wieder freigegeben werden. Dies wird von den Funktionen freeMemory_x509() und freeMemory_CRL() bewerkstelligt. Der folgende unvollständige SourceCode-Ausschnitt soll die verwendeten Mechanismen zeigen:

```
bool
freeMemory_x509 ()
{
      TElement *curEptr;
      ...
      while (anchor_x509->next != NULL) /* certs exists */
      {
            curEptr = anchor_x509->next;
```

```
            pfreeany(curEptr->data); /* memory dealloc */
            anchor_x509->next = curEptr->next;
            pfreeany(curEptr); /* memory dealloc */
        }
        ...
        if (!(countCert(anchor_x509)))
        {
            pfreeany(anchor_x509); /* memory dealloc */
            return TRUE;
        }
        return FALSE;
    }
```

Der Pluto stellt einige einfache Speicherallozierungshilfsfunktionen zur Verfügung. Mit alloc_bytes(...) wird die gewünschte Speichergrösse alloziert und mit pfreeany(...) wieder dealloziert. Mit dem „Leak-Detektive“ können anschliessend Speicherlöcher, das heisst wenn Speicher nicht mehr freigegeben wurde, aufgefunden werden.

8 Ausblick

Die aktuelle Version von FreeS/WAN ist 1.9.1, wir haben allerdings unser Diff-Patch auf die Version 1.9.0c erstellt. Die Anpassung an den aktuellen Versionsstand wird erst im Anschluss an diese Projektarbeit geschehen.

Wir haben uns noch einige Gedanken gemacht zur Erweiterung unseres Teiles. Die Funktionen, welche bis anhin für CRL- und x509-Zertifikate existieren, könnten polymorph gestaltet werden, das heisst dass für beide Typen dieselben Funktionen zu verwenden. Im Allgemeinen kann die Performance sicherlich noch verbessert werden.

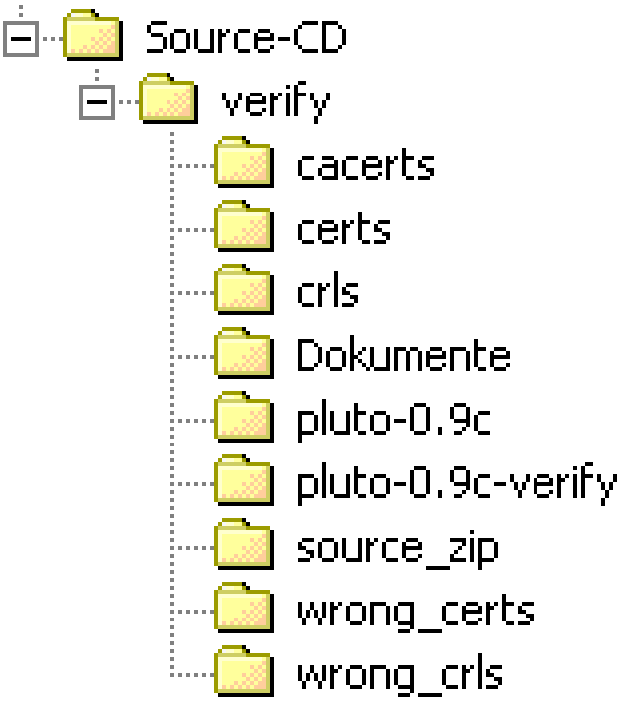
Weiter könnten die CA-Zertifikate und deren CRL's automatisch mit Hilfe des Internets aktualisiert werden.

9 Anhang

9.1 Source-CD

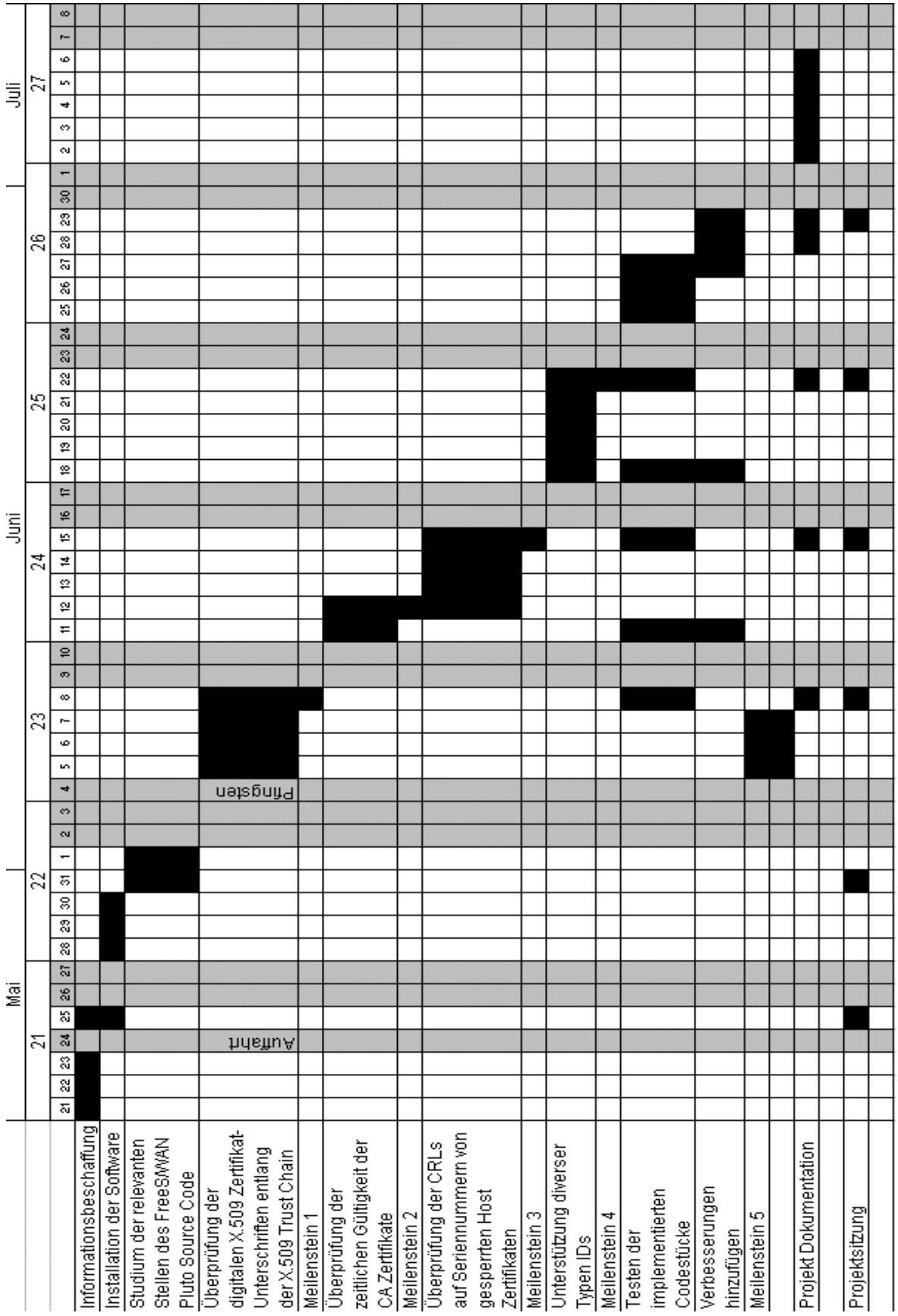
Die CD beinhaltet die Dokumentation, den Zeitplan, den Quellcode und verschiedene Zertifikate welche zu Testzwecken benutzt wurden.

9.1.1 Inhaltsverzeichnis



Der Ordner source_zip beinhaltet alle Source-Daten. Die sich darin befindliche Datei kann unter Linux mit `tar xvzf verify.tar.gz` entpackt werden.

9.2 Zeitplan


Mai
Juni
Juli
Informationsbeschaffung
Installation der Software
Studium der relevanten Stellen des FreeSWAN Pluto Source Code
Überprüfung der digitalen X.509 Zertifikat-Unterschriften entlang der X.509 Trust Chain
Auffahrt
Pfingsten
Meilenstein 1
Überprüfung der zeitlichen Gültigkeit der CA Zertifikate
Meilenstein 2
Überprüfung der CRLs auf Seriennummern von gesperrten Host Zertifikaten
Meilenstein 3
Unterstützung diverser Typen IDs
Meilenstein 4
Testen der implementierten Codestücke
Verbesserungen hinzufügen
Meilenstein 5
Projekt Dokumentation
Projektsitzung

9.3 Quellen

[DA 99] Diplomarbeit Kommunikationsnetze sna99/1
O.Gärtner/ Berkant Uenal
Herbst 1999

[PA 00] Projektarbeit „Linux - IPSec mit PGP und X.509 Zertifikaten“
P.Lichtsteiner/ M. Wegmann/ A. Hess
Frühling 2000

[WW ST] http://www.strongsec.ch/
Strongsec GmbH
Zürich

[RFC 2408] Security Association (SA)
[RFC 2409] Internet Key Exchange- Protokoll

[RFC 1321] The MD5 Message-Digest Algorithm
[RFC 2459] Internet X.509 Public Key Infrastructure Certificate and CRL Profile

[WW GP] GMP - GNU Multi-Precision Library
http://swox.com/gmp/

[WW FS] Linux - FreeS/WAN Projekt
http://www.freeswan.org

[WW FD] FreeS/WAN Dokumentation
http://www.freeswan.org/freeswan_trees/freeswan-1.9/doc/index.html

[WW RS] www.rsasecurity.com
RSA Security Inc.:
20 Crosby Drive, Bedford, MA 01730
Tel: 877 RSA 4900 or 781 301 5000
Fax: 781 301 5170
GPS: 42° 30' 42" N, 71° 14' 47" W

[PK 01] PKCS #5 - Password-Based Cryptography Standard

http://www.rsasecurity.com/rsalabs/pkcs/pkcs-5/index.html

[PK 05] PKCS #1 - RSA Cryptography Standard
http://www.rsasecurity.com/rsalabs/pkcs/pkcs-1/index.html

[PK 07] PKCS #7 - Cryptographic Message Syntax Standard
http://www.rsasecurity.com/rsalabs/pkcs/pkcs-7/index.html

[WW SH] Secure Hash Standard (SHS)
http://csrc.nist.gov/cryptval/shs.html

[SNA ASN] ASN.1 Encoding Rules
Unterrichtsstoff von Dr. A. Steffen