Studienarbeit Informatik

NAT-Traversal for strongSwan II

Tobias Brunner
Daniel Röthlisberger

Sommersemester 2006
7. Juli 2006

Betreut durch:
Martin Willi, Prof. Dr. Andreas Steffen
Institut für Internet-Technologien und Anwendungen
Hochschule für Technik, Rapperswil

Copyright © 2006, Tobias Brunner (`tbrunner`), Daniel Röthlisberger (`droethli`)

“The Linux philosophy is ‘laugh in the face of danger’.
Oops. Wrong one. ‘Do it yourself’. That’s it.”

– Linus Torvalds.

Abstract

Zielsetzung

strongSwan ist eine populäre, IPsec basierte Open Source VPN-Lösung für Linux. IKEv2 ist die neue Generation des verwendeten Schlüsselaustausch-Protokolls. Im Rahmen der Diplomarbeit strongSwan II von Jan Hutter und Martin Willi wurden die Grundzüge einer IKEv2-Implementierung für strongSwan entwickelt.

Damit eine VPN-Lösung im heutigen Internet breit eingesetzt werden kann, ist es unumgänglich, auch über NAT-Router hinweg Verbindungen betreiben zu können. Diese Fähigkeit wird NAT-Traversal (NAT-T) genannt. Mittels Dead Peer Detection (DPD) kann ein IKEv2-Daemon detektieren, ob sein Gegenüber noch aktiv ist, und entsprechend auf den Ausfall des Peers reagieren. Sowohl NAT-T als auch DPD sind Bestandteil der Spezifikation von IKEv2.

Im Rahmen dieser Studienarbeit war das Ziel, die bestehende IKEv2-Implementierung von strongSwan II um NAT-Traversal zu erweitern. Als Nebenziel war Dead Peer Detection (DPD) zu realisieren.

Ergebnis

Als Resultat unserer Arbeit ist strongSwan II heute in der Lage, Verbindungen mit IKEv2 in diversen NAT-Szenarien aufzubauen. Allfällige NATs werden detektiert, und alle nötigen Massnahmen getroffen, damit sowohl die IKEv2-Verbindung als auch die darüber ausgehandelten IPsec-Nutzverbindungen trotz NAT zustande kommen und betrieben werden können.

Zusätzlich haben wir die wesentlichen Bestandteile von Dead Peer Detection implementiert. Tote Verbindungen werden detektiert, und soweit es der derzeitige Stand von strongSwan II zulässt, wird darauf reagiert.

Als Nebenprodukt der Arbeit im Umfeld von strongSwan II haben wir diverse Verbessungen und Korrekturen an nicht direkt mit NAT-T oder DPD in Zusammenhang stehendem Code vorgenommen oder vorgeschlagen.

Ausblick

Unsere Arbeit ist in Form von Patches bereits in das strongSwan-Projekt eingeflossen, und wird dort von Martin Willi und Prof. Dr. Andreas Steffen weiterentwickelt werden.

Inhaltsverzeichnis

Abstract **v**

1. Aufgabenstellung **1**

2. Management Summary **3**

3. Einleitung **5**

3.1. Problemstellung 5
3.2. strongSwan II 5
3.3. Gliederung der Dokumentation 6

4. Grundlagen **7**

4.1. Network Address Translation (NAT) 7
4.1.1. Entstehungsgeschichte 7
4.1.2. Beispiel Masquerading 8
4.1.3. Andere Formen von NAT 9
4.1.4. Probleme für Protokolle und Anwendungen 11
4.2. Internet Protocol Security (IPsec) 12
4.2.1. AH, ESP, Tunnel und Transport Mode 12
4.2.2. Security Associations, Policies, SPI 13
4.2.3. Internet Key Exchange Version 2 (IKEv2) 14
4.2.4. NAT Traversal (NAT-T) 15
4.2.5. Dead Peer Detection (DPD) 18
4.3. Kernel-Schnittstelle 19
4.3.1. IPsec im Linux Kernel 19
4.3.2. PF_KEY 20
4.3.3. Netlink/XFRM 21
4.4. UML — User Mode Linux 26

5. Design / Implementation **27**

5.1. Methodik 27
5.1.1. Programmierrichtlinien 27

5.1.2. Source Code Management 27
5.1.3. Issue Tracking, Ticketing 28
5.2. Architektur im Überblick 28
5.3. Teilsysteme 29
5.3.1. Network Sockets 29
5.3.2. Kernel Interface 35
5.3.3. NAT Detection (NAT-D) 40
5.3.4. Adress-Update und Port-Agilität 41
5.3.5. NAT Keepalives 42
5.3.6. Dead Peer Detection (DPD) 42
5.4. Sonstige Resultate 44
5.5. Angeregte Verbesserungen 44
5.5.1. Integer Overflows in `event_queue` 44
5.5.2. SPI zufällig erzeugen 45
5.5.3. OOD: Job-Logik in Jobs statt Thread Pool 46

6. Tests **47**
6.1. Methodik 47
6.1.1. Reale Tests 47
6.1.2. UML Tests 47
6.2. Testfälle 49
6.2.1. SNAT 49
6.2.2. Double SNAT 50
6.2.3. DNAT 51
6.2.4. Double NAT (SNAT/DNAT) 51
6.2.5. Spezialfälle 52

7. Projektstand **53**
7.1. Zukunftvisionen 53
7.1.1. Verifikation von NAT-T/NAT-D 53
7.1.2. DPD mit konfigurierbaren Actions 53
7.1.3. `conntrack` im UML Test Framework 54
7.1.4. Netzwerktopologie im UML Test Framework 54
7.1.5. XFRM Address Change Notification Kernel Patch 54
7.1.6. Unterstützung für `subnetwithin` 55
7.1.7. IPv6-Support 55
7.1.8. Timeout in Kernel-Interface 55

A. Projektmanagement **57**
A.1. Projektplan 57
A.2. Zeitabrechnung 57
A.3. Qualitätssicherung 58
A.3.1. Massnahmen 58
A.3.2. Effektivität 59
A.4. Protokolle 60
A.4.1. Sitzung Woche 1 60
A.4.2. Sitzung Woche 2 62
A.4.3. Sitzung Woche 3 63
A.4.4. Sitzung Woche 4 64
A.4.5. Sitzung Woche 6 65
A.4.6. Sitzung Woche 7 66
A.4.7. Sitzung Woche 8 67
A.4.8. Sitzung Woche 9 68
A.4.9. Sitzung Woche 10 69
A.4.10. Sitzung Woche 11 70
A.4.11. Sitzung Woche 12 71
A.4.12. Sitzung Woche 13 72
A.4.13. Sitzung Woche 14 74

B. Erfahrungsberichte **75**
B.1. Tobias Brunner 75
B.2. Daniel Röthlisberger 76

Abkürzungsverzeichnis **77**

1. Aufgabenstellung

Einführung

strongSwan II ist ein Grundlagenprojekt des Instituts für Internet Technologien und Anwendungen (ITA). Über einen Zeitraum von zwei Jahren soll eine komplette IKEv2 Implementation geschaffen werden. Einen wichtigen Platz nimmt dabei die Transversierung von NAT Routern ein, weil nur damit IPsec-basierte VPNs effizient in einem Remote Access Umfeld eingesetzt werdem können. In dieser Studienarbeit sollen, aufsetzend auf dem durch die Diplomanden Jan Hutter und Martin Willi geschaffenen strongSwan II Grundkonzept, die NAT Traversal Standards RFC 3947 und 3948 integriert werden. Die NAT-T Realisierung soll anhand von SNAT, DNAT sowie Double NAT Szenarien unter Einbezug der Rekeying-Problematik ausgiebig getestet werden. Als Programmiersprache kommt C zum Einsatz. Dabei sollen die Coding Rules des strongSwan II Projekts strikt eingehalten werden.

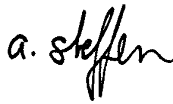
Aufgabenstellung

- Einarbeitung in den strongSwan II Source Code und in die IKE/IPsec NAT-Traversal RFCs 3947 und 3948
- Implementierung der NAT Detektion, sowie der Aushandlung der NAT-Traversal Methode via das IKEv2 Protokoll. Weiter soll die Konfiguration der UDP Enkapsulierung von ESP Paketen via das XFRM Kernel Interface, sowie das Senden von periodischen NAT Keep-Alive Paketen realisiert werden.
- Testen der NAT Traversierung mittels SNAT, DNAT, sowie Double NAT User-Mode-Linux Szenarien, sowie in praktischen Netzwerk Setups.
- Optional kann zusätzlich das Dead Peer Detection Protokoll (DPD) implementiert werden.

Links

- strongSwan Projekt:
  `http://www.strongswan.org/`
- Negotiation of NAT-Traversal in the IKE:
  `http://www.ietf.org/rfc/rfc3947.txt`
- UDP Encapsulation of IPsec ESP Packets:
  `http://www.ietf.org/rfc/rfc3948.txt`
- Internet Key Exchange (IKEv2) Protocol:
  `http://www.ietf.org/rfc/rfc4306.txt`
- Dead Peer Detection (DPD) Protocol:
  `http://www.ietf.org/rfc/rfc3706.txt`

Rapperswil, 3. April 2006



Prof. Dr. Andreas Steffen

2. Management Summary

Projektauftrag

strongSwan ist eine populäre, IPsec basierte Open Source VPN-Lösung für Linux. IKEv2 ist die neue Generation des verwendeten Schlüsselaustausch-Protokolls. Im Rahmen der Diplomarbeit strongSwan II von Jan Hutter und Martin Willi wurden die Grundzüge einer IKEv2-Implementierung für strongSwan entwickelt.

Damit eine VPN-Lösung im heutigen Internet breit eingesetzt werden kann, ist es unumgänglich, auch über NAT-Router hinweg Verbindungen betreiben zu können. Diese Fähigkeit wird NAT-Traversal (NAT-T) genannt. Mittels Dead Peer Detection (DPD) kann ein IKEv2-Daemon detektieren, ob sein Gegenüber noch aktiv ist, und entsprechend auf den Ausfall des Peers reagieren. Sowohl NAT-T als auch DPD sind Bestandteil der Spezifikation von IKEv2.

Im Rahmen dieser Studienarbeit war das Ziel, die bestehende IKEv2-Implementierung von strongSwan II um NAT-Traversal zu erweitern. Als Nebenziel war Dead Peer Detection (DPD) zu realisieren.

Das Ergebnis

Als Resultat unserer Arbeit ist strongSwan II heute in der Lage, Verbindungen mit IKEv2 in diversen NAT-Szenarien aufzubauen. Allfällige NATs werden detektiert, und alle nötigen Massnahmen getroffen, damit sowohl die IKEv2-Verbindung als auch die darüber ausgehandelten IPsec-Nutzverbindungen trotz NAT zustande kommen und betrieben werden können.

Zusätzlich haben wir die wesentlichen Bestandteile von Dead Peer Detection implementiert. Tote Verbindungen werden detektiert, und soweit es der derzeitige Stand von strongSwan II zulässt, wird darauf reagiert.

Als Nebenprodukt der Arbeit im Umfeld von strongSwan II haben wir diverse Verbessungen und Korrekturen an nicht direkt mit NAT-T oder DPD in Zusammenhang stehendem Code vorgenommen oder vorgeschlagen.

NAT

Unter dem Sammelbegriff Network Address Translation (NAT) versteht man Verfahren, welche die Sender- und/oder Empfängeradressen von IP-Paketen unterwegs verändern. In der häufigsten Anwendungsform wird NAT verwendet, um Client-Rechner in einem lokalen Netzwerk über eine oder mehrere öffentliche IP-Adresse(n) mit einem externen Netz zu verbinden — in der Regel mit dem Internet. NAT aller Ausprägungen haben teilweise schwerwiegende Auswirkungen auf Internet-Protokolle, insbesondere auf IPsec.

NAT-Traversal

NAT-Traversal ist ein freiwilliger Bestandteil von IKEv2. Zu NAT-Traversal gehören im Wesentlichen folgende Massnahmen:

- NAT Detection (NAT-D), um allfällige NAT auf Seiten des Senders und des Empfängers zu detektieren,

- UDP Encapsulation und Wechsel auf Ports 4500, um IPsec-Pakete NAT-fähig zu verpacken,
- Dynamisches Adressupdate und Port-Agilität, um durch NAT-Router verursachte Änderungen von Adressen und Ports während der Dauer einer Verbindung zu kompensieren, sowie
- NAT Keepalives, um die NAT-Mappings auf den NAT-Routern aktiv zu halten.

Diese Massnahmen sind im Standard zu IKEv2 [Kau05] dokumentiert, es existierte aber vor unserer Arbeit keine Implementierung von NAT-T für IKEv2.

Linux Kernel Der Linux-Kernel betreibt für strongSwan II die eigentlichen IPsec-Nutzverbindungen. Zu diesem Zweck verwaltet er Verbindungsparameter wie Adressen, Ports, verwendete kryptographische Algorithmen und Schlüsselmaterial. Es ist Aufgabe von strongSwan II, diese Parameter über IKEv2 auszuhandeln, und dem Kernel dann alle nötigen Informationen mitzuteilen. Im Falle von NAT-Traversal kommen noch Parameter zur UDP-Enkapsulierung hinzu, und aufgrund der dynamischen Adressupdates müssen diese Parameter im laufenden Betrieb aktualisiert werden können. Und für NAT Keepalives und DPD ist es notwendig, Informationen zur Idle Time von IPsec-Verbindungen abzufragen.

Zu diesen Zwecken haben wir intensiv mit der `Netlink/XFRM` -Kernel-Schnittstelle von Linux 2.6 gearbeitet.

Ausblick Unsere Arbeit ist in Form von Patches bereits in das strongSwan-Projekt eingeflossen, und wird dort von Martin Willi und Prof. Dr. Andreas Steffen weiterentwickelt werden. Somit ist strongSwan die erste Implementierung von IKEv2 mit Unterstützung für NAT-Traversal.

3. Einleitung

3.1. Problemstellung

Die Zielsetzung dieser Studienarbeit war im Wesentlichen die Erweiterung des IKEv2-Daemons `charon` von strongSwan um die Fähigkeit, IPsec-Verbindungen über Network Address Translation (NAT) auszuhandeln und zu betreiben. Diese Fähigkeit wird NAT-Traversal genannt, und ist wie IKEv2 selber in [Kau05] beschrieben.

Im heutigen Internet ist ein grosser Anteil der Benutzer über NAT angeschlossen, weshalb die Implementation von NAT-Traversal von grosser Bedeutung für die Benutzbarkeit von `charon` in der Praxis ist.

Als optionales Ziel der Arbeit war die Implementation von Dead Peer Detection (DPD) gegeben. Ohne Dead Peer Detection wird eine unterbrochene IKE-Session unter Umständen erst Stunden später als nicht mehr aktiv erkannt. Mit Dead Peer Detection wird sichergestellt, dass unter- oder abgebrochene Verbindungen so schnell wie möglich erkannt werden können und entsprechend reagiert werden kann.

3.2. strongSwan II

strongSwan II ist der interne Projektname der IKEv2-Implementation von strongSwan. Diese ist im Rahmen der Diplomarbeit [HW05] von Jan Hutter und Martin Willi entstanden.

strongSwan ist eine Open Source, IPsec basierte VPN-Lösung mit Fokus auf X.509-Zertifikaten. Entstanden ist strongSwan als Fork von FreeS/WAN sowie dem von Prof. Dr. Andreas Steffen entwickelten X.509 Patch. FreeS/WAN war die erste komplette Implementation von IPsec für Linux, ursprünglich auf die Initiative von John Gilmore hin entstanden. Die Entwicklung von FreeS/WAN wurde 2003 eingestellt. Neben dem ebenfalls von FreeS/WAN abstammenden Openswan ist strongSwan heute eine der zwei bedeutendsten IPsec-Lösungen für Linux.

strongSwan besteht aus folgenden Komponenten:

- Der Daemon `pluto` implementiert IKEv1, während
- `charon` IKEv2 in einem separaten Daemon implementiert.
- `whack` ist das Kontrolltool zu `pluto`, und

- `stroke` das Pendant bei `charon`.
- Der `starter` startet, konfiguriert und überwacht die Daemons, und mit
- `KLIPS` ist der IPsec Kernel-Patch zu Linux 2.4 weiterhin verfügbar.

Ein Teil der gemeinsamen Funktionalität ist dabei in shared Libraries ausgelagert. Mehr zum Kernel und seiner Schnittstelle zu Userland-Daemons wie `pluto` und `charon` beschreiben wir in Kapitel 4.3.

3.3. Gliederung der Dokumentation

Die Dokumentation unserer Studienarbeit ist in folgende Teile gegliedert:

- Im Kapitel Grundlagen erläutern wir technische Grundlagen, welche für das Verständnis der Arbeit relevant sind.
- Im Kapitel Design / Implementation befassen wir uns mit den Fragen des konkreten Softwaredesigns und erläutern nennenswerte Implementation-Details.
- Im Kapitel Tests haben wir konkrete Testfälle sowie unser Vorgehen beim Testen dokumentiert.
- Im Kapitel Projektstand analysieren wir das Resultat und zeigen Möglichkeiten der Weiterentwicklung auf.

4. Grundlagen

In diesem Kapitel beschreiben wir die wichtigsten technischen Grundlagen, welche für das Verständnis der folgenden Kapitel notwendig sind. Wir beschränken uns hierbei auf das Notwendige, und verweisen für eine weitergehende Beschreibung auf die jeweiligen Standards oder Originaldokumente.

Dieses Kapitel kann auch gut übersprungen werden, um später bei Bedarf auf einzelne Abschnitte zurückzukommen.

4.1. Network Address Translation (NAT)

Unter dem Sammelbegriff Network Address Translation (NAT) versteht man Verfahren, welche die Sender- und/oder Empfängeradressen von IP-Paketen unterwegs verändern. In der häufigsten Anwendungsform wird NAT verwendet, um Client-Rechner in einem lokalen Netzwerk über eine oder mehrere öffentliche IP-Adresse(n) mit einem externen Netz zu verbinden — in der Regel mit dem Internet. Andere Anwendungsformen machen eine 1:1 Zuordnung von Adressen aus einem Adressblock auf Adressen in einem anderen Adressblock, oder machen einen Server in einem internen Netzwerk unter einer externen IP-Adresse sichtbar.

Man unterscheidet grundsätzlich zwischen einfacher Network Address Translation, auch Basic NAT genannt, wo nur die IP-Adressen modifiziert werden, und Network Address Port Translation (NAPT), auch Port Address Translation (PAT), Hiding NAT, IP Masquerading (Linux) oder NAT Overloading (Cisco IOS) genannt, wo auch die Ports verändert werden, damit mehrere interne Adressen auf die gleiche externe Adresse gemappt werden können.

4.1.1. Entstehungsgeschichte

Entwickelt wurde NAT ursprünglich infolge der Adressknappheit des Internet Protokolls in Version 4 (IPv4). IP-Adressen sind 32 Bit gross, somit gäbe es theoretisch $2^{32} \approx 4.3$ Mia. mögliche Adressen. In den Anfängen des Internet wurden IP-Adressen jedoch regelrecht verschwendet. Ursprünglich wurden die ersten 8 Bit als Netzwerknummer verwendet, der Rest zur Identifikation des Hosts. Um eine grössere Anzahl von Netzen zu ermöglichen, hat man später mit Classful Addressing [Pos81] die Länge der Netznummer abhängig gemacht vom Wert der ersten 1–4 Bit: je nach Wert dieser Bits wurden die Adressen in die Klassen A (8 Bit Netznummer), B (16 Bit Netznummer) und C (24 Bit Netznummer) eingeteilt (und einige Adressen für spezielle Zwecke wie

Multicast reserviert). Doch auch dies reichte nicht aus, weshalb man mit Classless Inter-Domain Routing (CIDR) [RL93; FLYV93] schliesslich die Länge der Netzwerknummer komplett variabel gemacht hat, indem neben den Adressen immer eine sogenannte Netzmaske angegeben wird, welche spezifiziert, welche Bits der Adresse die Netzwerkidentifikation repräsentieren.

Doch mit der explosionsartigen Verbreitung des Internet stiess man auch hier bald an Grenzen. Heute weist man schon lange nicht mehr jedem ans Internet angeschlossenen Rechner eine eigene Adresse fix zu. Stattdessen werden z.B. den Kunden eines Internet Service Providers (ISP) Adressen aus einem kleinen Pool von öffentlichen IP-Adressen dynamisch zugewiesen.

Während CIDR als kurzfristige Lösung für die Probleme von IPv4 angesehen wurde, und mit der Version 6 des Internet Protokolls (IPv6) gleichzeitig an der langfristigen Lösung gearbeitet wurde, hat man als weitere kurzfristige Überbrückungslösung Network Address Translation entwickelt [EF94; SE01], um lokale Netze unter einer oder mehrerer öffentlichen IP-Adressen ans Internet anzuschliessen, ohne für das lokale Netz einen eigenen öffentlichen Adressblock und somit viele öffentliche Adressen zu verbrauchen.

4.1.2. Beispiel Masquerading

Am Beispiel des häufigsten Falles möchten wir die Funktionsweise von NAT im Detail aufzeigen. In dieser Anwendungsform wird ein internes Netzwerk über einen NAT-Router mit einer einzigen vom ISP zugewiesene öffentliche Adresse mit dem Internet verbunden. Diese Ausgangslage ist in Abbildung 4.1 dargestellt. Diese Form von NAT wird auch PAT, NAPT, Hiding NAT und NAT Overloading genannt. Anstelle dem Internet könnte das interne Netz natürlich analog zu der hier dargestellten Situation auch mit einem anderen WAN verbunden werden.

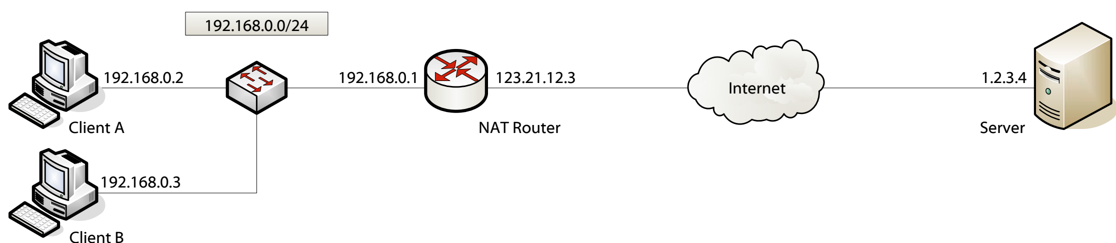


Abbildung 4.1.: NAT in der Übersicht

Der Gateway (NAT-Router) schreibt hierzu die Source-Adressen von Paketen, welche das lokale Netz verlassen, auf die externe IP-Adresse um. Nach aussen hin ist also nur die externe IP-Adresse des NAT-Routers sichtbar. Wenn der Client von seiner lokalen Adresse `192.168.0.2` aus eine HTTP-Verbindung zum Server `1.2.3.4` startet, so sieht der Server eine eingehende Verbindung von der Adresse `123.21.12.3`, also der externen IP-Adresse des NAT-Routers. Der Source-Port wurde vom NAT-Router auf eine eindeutige Port-Nummer umgeschrieben, sodass

er die Antwortpakete des Servers wieder dem richtigen internen Host zuordnen kann. Zu diesem Zweck führt der NAT-Router eine Tabelle, in der er sich die Source-Adress und -Port Tupel aller laufenden Verbindungen[1] speichert. Der Vorgang ist in Abbildung 4.2 nochmals genauer dargestellt.

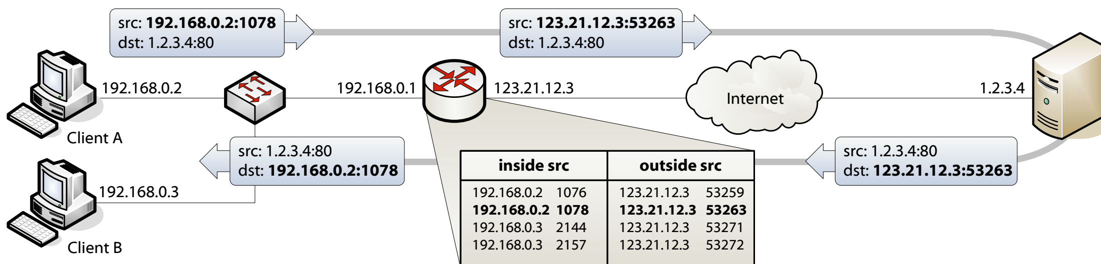


Abbildung 4.2.: NAT: Beispiel

Masquerading als Sicherheitsfeature?

Aufgrund der Tatsache, dass Hosts im lokalen Netz hinter dem NAT-Router nicht direkt von aussen erreichbar sind, gelten NAT dieser Art heute auch als Sicherheitsfeature. Auch wenn die Aussage berechtigt ist, dass ein Client hinter einem solchen NAT-Router vor vielen aktiven Angriffen aus dem Internet geschützt ist, so muss man sich aber vor Augen halten, dass NAT nicht primär der Sicherheit dient, und den Client sowieso nur vor einer eingeschränkten Angriffsklasse "schützt". Angriffe z.B. über den Web-Browser oder via E-Mail können damit nicht verhindert werden, und sind wesentlich gefährlicher, als "nur" von der eigenen Personal Firewall geschützt direkt und ohne NAT ans Internet angeschlossen zu sein. Schutzmassnahmen auf Netzwerk-Ebene können effektiver und sinnvoller mit Firewalls realisiert werden, ohne NAT einsetzen zu müssen.

4.1.3. Andere Formen von NAT

Es existieren diverse andere Arten von NAT. Einige der bei der Beschreibung von NAT-Szenarien verwendeten Begriffe möchten wir hier erklären.

[1]Eine Verbindung ist in diesem Zusammenhang nicht auf verbindungsorientierte Protokolle wie TCP beschränkt, sondern bezieht sich auch auf verbindungslose Protokolle wie UDP und ICMP. Als UDP-Verbindung bezeichnet man hier alle Datagramme, welche zwischen zwei konkreten Host:Port-Kombinationen ausgetauscht werden. ICMP-Pakete werden je nach Typ der zugehörigen UDP- oder TCP-Verbindung zugeordnet (z.B. Host / Port Unreachable, Source Quench), oder als eigene Verbindung behandelt (z.B. Echo / Echo Reply).

Source NAT vs Destination NAT

Mit Source NAT (SNAT) bezeichnet ein NAT wo die Quelladresse und/oder der Quellport verändert wird, und bei Destination NAT (DNAT) analog dazu die Zieladresse und/oder der Zielport.

Implizit wird dadurch auch eine Richtung vorgegeben — bei Source NAT werden Verbindungen von innen nach aussen [2] vom NAT erfasst, bei Destination NAT werden Verbingungen von aussen nach innen erfasst.

Port Forwarding ist eine Form von Destination NAT, während Masquerading eine Form von Source NAT darstellt.

Statisches NAT

Bei statischem NAT wird eine Adresse aus dem internen Netz fix auf eine Adresse des externen Netzes gemappt. Es findet keine Wiederverwendung oder Mehrfachbelegung (Overloading) von Adressen statt.

Diese Form von NAT wird beispielsweise verwendet, wenn man zwei Firmennetze miteinander komplett verbinden will, ohne eines der Netze komplett mit neuen Nummern auszustatten. Stattdessen findet auf den verbindenden Gateways eine 1:1 Umsetzung von einem Adressbereich in einen anderen Adressbereich statt.

Diese Form von NAT ist zugleich am wenigsten problematisch für Protokolle auf höheren Schichten, weil die End-zu-End-Semantik des Internet Protocol beibehalten wird.

Port Forwarding

Unter Port Forwarding wird gemeinhin das Weiterleiten von Verbindungen an einen oder mehrere Ports eines NAT-Routers an eine Drittmaschine verstanden. In der Regel befindet sich diese Drittmaschine im internen Netz hinter Masquerading, und könnte ohne Port Forwarding nicht von aussen her erreicht werden. Durch das Weiterleiten von Ports können so auch Server hinter einem solchen NAT-Router betrieben werden. Port Forwarding ist somit eine spezielle Form von Destination NAT.

Port Forwarding wird von einigen Herstellern von Enduser-Produkten ziemlich seltsam bezeichnet. ZyXEL beispielsweise nennt Port Forwarding “SUA Server”, wobei SUA für Single User Account steht, was daher rührt, dass nur eine IP-Adresse vom ISP bezogen wird (Single User), und daher NAT notwendig ist.

[2] Die Begriffe “innen” und “aussen” machen nur dann so Sinn, wenn es sich um ein Setup wie in Abbildung 4.1 handelt, was natürlich nicht zwingenderweise gegeben ist.

Einzeladressen vs Adresspools

Bei den meisten Formen von NAT ist es prinzipiell möglich, anstatt einzelne Adresse(n) einen Pool von Adressen zu verwenden. Beispielsweise Masquerading anstelle auf nur eine externe Adresse auf einen Pool von n externen Adressen. Oder Port Forwarding an mehrere anstatt nur eine interne Adresse, wodurch eine Form von Load Balancing erreicht werden kann (Verteilung der Last auf mehrere physikalische Server, welche Anfragen an eine virtuelle IP-Adresse beantworten).

4.1.4. Probleme für Protokolle und Anwendungen

Während NAT auf Ebene der Network und Transport Layer Sinn machen kann, und spannende Lösungen für komplexe Probleme bietet, bereitet es darüberliegenden Layern unter Umständen grosse Probleme. Protokolle, welche auf die ursprüngliche End-zu-End-Semantik des Internet-Protokolls angewiesen sind, funktionieren plötzlich über NAT hinweg nicht mehr. Protokolle, welche IP-Adressen als Teil des Protokolls austauschen und darauf basierend weitere Verbindungen öffnen, versagen kläglich, wenn die Kommunikation über einen oder mehrere NAT-Router läuft, die beim Gegenüber sichtbare Adresse also von der effektiven Adresse eines Peers abweicht. Oder anders ausgedrückt kann ein Host nicht mehr ohne fremde Hilfe herausfinden, unter welcher Adresse er erreichbar ist, ja diese Adresse kann gar variieren, je nachdem von woher er erreicht werden soll.

Als konkretes Beispiel sei hier das File Transfer Protocol (FTP) genannt. Dieses verwendet zwei getrennte Verbindungen: eine Steuerverbindung (Control), und eine Datenverbindung (Data). Der Client baut zuerst eine Steuerverbindung zum Server auf. Über diesen Steuerkanal meldet der Client schliesslich dem Server seine Adresse und Port, wo er auf den Verbindungsaufbau für die Datenverbindung wartet. Der Server baut dann die Datenverbindung zu der vom Client genannten Adresse und Port auf.

Wenn hier jetzt ein NAT ins Spiel kommt, so sendet der Client dem Server eine "falsche" Adresse, sprich eine, welche nach aussen hin nicht erreichbar ist. Der Versuch des Servers, die Datenverbindung aufzubauen, wird also fehlschlagen.

Grundsätzlich gibt es für derartige Probleme immer zwei Lösungen: entweder wird das Protokoll angepasst, oder dem NAT-Router wird Unterstützung für das Protokoll beigebracht. Im Falle von FTP sieht die Lösung auf Protokollebene so aus, dass es einen sogenannten Passive Mode gibt, in welchem die Datenverbindung ebenfalls vom Client zum Server hin aufgebaut wird. Die andere Lösung würde bedeuten, dass der NAT-Router den Inhalt der FTP-Pakete analysiert, und die übermittelte interne Adress/Port-Kombination des Clients on-the-fly durch eine externe Adress/Port-Kombination ersetzt, und die Datenverbindung später an den Client weiterleitet. Auch das wird heute von den meisten NAT-Routern unterstützt.

Damit bestehen mehrere Probleme. Das Inspizieren und Verändern von Daten höherliegender Schichten stellt eine Verletzung des OSI-Schichtenmodells dar. Ein Router arbeitet eigentlich nur

auf der Netzwerkschicht, womit er sich aus Gründen der Abstraktion und der Robustheit nicht für die Daten der höherliegenden Schichten zu interessieren hat.[3] Mit der Notwendigkeit, dass der NAT-Router ein konkretes Protokoll höherer Schichten unterstützen muss, reicht es plötzlich nicht mehr, wenn die beiden Endpunkte einer Verbindung sich über das Protokoll einig sind — der NAT-Router muss das Protokoll ebenfalls kennen und (korrekt) unterstützen. Ein besonderes Problem stellen verschlüsselte Verbindungen dar, dessen Inhalte vom NAT-Router folglich weder inspiziert noch kontrolliert verändert werden können.

4.2. Internet Protocol Security (IPsec)

IPsec ist eine umfassende Sicherheitsarchitektur für IP-basierte Netzwerke. IPsec ist die Lösung, welche von der Internet Engineering Task Force (IETF) entwickelt wurde, um der mangelnden Sicherheit im Internet-Protokoll zu begegnen. IPsec ist zwingender Bestandteil von IPv6, bei IPv4 aber noch optionaler Zusatz.

Eine häufige Anwendung von IPsec sind Virtual Private Networks (VPN). Ein VPN ist ein virtuelles sicheres Netzwerk, das anstelle eines physikalischen Mediums über ein anderes, öffentliches und grundsätzlich nicht vertrauenswürdiges Netz transportiert wird.

IPsec arbeitet im Gegensatz zu SSL/TLS, SSH, PGP und anderen Lösungen direkt auf dem Network Layer, und stattet diesen unter Verwendung von starker Kryptographie mit Sicherheitsattributen wie Vertraulichkeit, Integrität und Authentizität aus. Realisiert wird das auf dieser Ebene mit den Protokollen Encapsulated Security Payload (ESP) und Authenticated Header (AH), welche sich zwischen dem Link und dem Network Layer einfügen, also im heute häufigsten Fall zwischen Ethernet und IP. [wik]

Die kryptographischen Parameter (verwendete Algorithmen und Schlüssel) können entweder statisch konfiguriert werden, oder dynamisch über einen Schlüssel- und Konfigurationsparameteraustausch zwischen miteinander kommunizierenden Hosts abgemacht werden. Dieser Austausch geschieht über das ebenfalls zur IPsec-Gesamtarchitektur gehörende Protokoll Internet Key Exchange (IKE).

Für eine weitergehende Dokumentation von IPsec als Gesamtsystem verweisen wir auf [KA98c].

4.2.1. AH, ESP, Tunnel und Transport Mode

Authenticated Header (AH) bietet Authentizität, Integrität und Schutz vor Replay-Attacken, verschlüsselt die Daten aber nicht. AH schützt alle invarianten Header-Felder sowie die Nutzdaten. AH ist in [KA98a] ausführlich dokumentiert.

Encapsulated Security Payload (ESP) bietet Vertraulichkeit, Authentizität, Integrität sowie Schutz vor Replay-Attacken. Der Schutz deckt aber nicht wie bei AH auch Header-Felder ab, sondern nur die Nutzdaten. ESP ist in [KA98b] ausführlich dokumentiert.

[3] So gesehen ist NAT an sich schon eine Verletzung der Schichtenarchitektur.

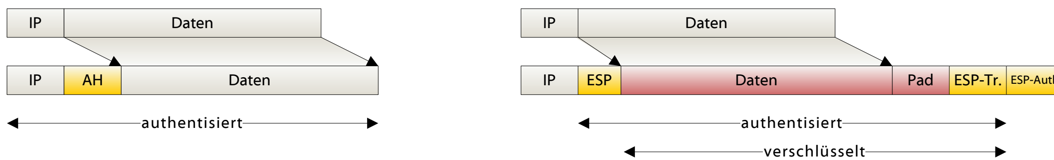


Abbildung 4.3.: AH/ESP in Transport Mode

Beide Protokolle können in zwei unterschiedlichen Anwendungsmodi betrieben werden: Transport Mode (Abb. 4.3) und Tunnel Mode (Abb. 4.4). Wie aus den Abbildungen ersichtlich ist, wird beim Transport Mode direkt das Nutzpaket mit AH und/oder ESP ausgestattet, d.h. der ursprüngliche IP-Header bleibt erhalten, während im Tunnel Mode das komplette Paket inklusive IP-Header in ein AH- oder ESP-Paket verpackt wird, und dieses mit einem neuen, äusseren IP-Header versehen wird. Tunnel Mode eignet sich deshalb, um Virtual Private Networks mit virtueller Adressierung zu implementieren, während Transport Mode mit weniger Overhead direkt die normale Host-Host-Kommunikation schützen kann. Man kann aber sagen, dass der Transport Mode eigentlich überflüssig ist, weil man die gleiche Funktionalität auch mit dem Tunnel Mode erreichen kann. Mit derselben Argumentation wird oft auch AH für überflüssig erklärt, da mit ESP in Tunnel Mode die gleiche Sicherheit erreicht werden kann, und AH ohne ESP in der Praxis praktisch nicht eingesetzt wird.

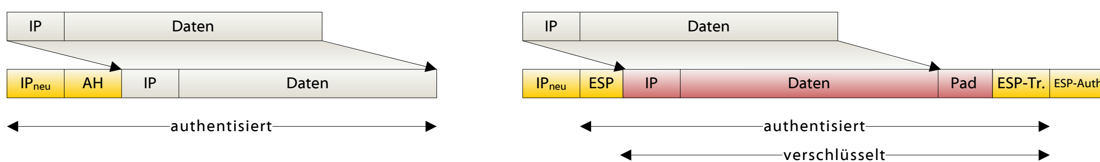


Abbildung 4.4.: AH/ESP in Tunnel Mode

4.2.2. Security Associations, Policies, SPI

Der IPsec-Standard definiert eine Security Association (SA) als die Parameter, welche eine IPsec-Verbindung beschreiben. Eine Security Association wird lokal durch das Tripel Security Parameter Index (SPI), IP-Adresse des Ziels sowie das verwendete Protokoll (AH oder ESP) eindeutig identifiziert. Zur SA gehören neben diesem Tripel auch alle Konfigurationsparameter, also Eigenschaften wie verwendete kryptographische Algorithmen und Schlüssel.

Jeder Host speichert die Security Associations in einer sogenannten Security Association Database (SADB oder SAD) ab. Dies ist im Grunde genommen lediglich ein in der Regel auf kernelebene implementierter Speicher, in welchem alle auf einer Maschine aktiven Security Associations abgelegt sind. Zugriff erfolgt über den Primärschlüssel (SPI, Destination Address, Protocol).

Der Security Parameter Index (SPI) ist ein zufälliger Wert, der zur Identifikation einer ESP-, AH- oder IKE-SA dient. Dieser Wert muss lokal eindeutig sein, und wird in sämtlichen Paketen übermittelt, damit die Gegenseite das Paket einer Verbindung zuordnen kann.

Es wird ferner im Kontext von IKEv2 unterschieden zwischen einer IKE-SA und einer Child-SA. Die IKE-SA schützt die IKE-Verbindung selber, während die Child-SA die eigentlichen SAs der Nutzverbindungen darstellen.

Während eine SA alle Informationen beinhaltet, um eine IPsec-Verbindung zu nutzen und Pakete entsprechend zu sichern, ist in der SA aber noch nicht definiert, *welcher* IP-Verkehr über eine IPsec-Verbindung geleitet werden soll. Diese Information ist in der sogenannten IPsec-Policy definiert. Eine solche Policy legt beispielsweise fest, dass jeglicher Datenverkehr von `192.168.0.1` nach `192.168.1.7` durch die IPsec-SA 1 geschützt werden soll. Policies können auch pro Socket Regeln definieren, um beispielsweise die Pakete eines Sockets speziell zu behandeln.

Diese Konzepte sind in [KA98c] ausführlicher beschrieben. Die Zuordnung von Policies zu Security Association geschieht in der IPsec-Implementierung von Linux 2.6 über die sogenannte `reqid`.

4.2.3. Internet Key Exchange Version 2 (IKEv2)

Das Internet Key Exchange Protokoll dient der automatischen Verwaltung von Security Associations. Es tauscht zwischen Peers alle nötigen Parameter aus, damit diese untereinander IPsec-Verbindungen aufbauen können. Es führt zu diesem Zweck eine Authentisierung der Verbindungsteilnehmer durch, erzeugt eine IKE Security Association, um die IKE-Verbindung selber zu schützen, und kann anschliessend für die Aushandlung von neuen IPsec-Nutzverbindungen verwendet werden (Child SAs).

IKEv1 basiert auf einem Framework mit dem Namen Internet Security Association and Key Management Protocol (ISAKMP), und ist in den zwei Standards ISAKMP [MSST98] und der IPsec Domain of Interpretation (DOI) für ISAKMP [Pip98] definiert. IKEv2 ist die neue, verbesserte Generation des Protokolls, definiert in [Kau05]. IKE basiert in beiden Versionen auf UDP-Datagrammen auf dem dafür vorgesehenen Port 500.

Gegenüber IKEv1 besitzt IKEv2 einige Vorteile. So ist es weniger komplex, flexibler, und durch nur einen einzigen Standard komplett beschrieben. Wir stellen die Funktionsweise von IKEv2 im Folgenden ganz kurz vor, soweit es für das Verständnis von NAT-T und DPD notwendig ist, und verweisen für die vollständige Beschreibung des Protokolls neben dem bereits erwähnten RFC auf die hervorrangende Diplomarbeit von Jan Hutter und Martin Willi [HW05].

Derjenige Host, welcher die IKE-Verbindung initiiert, wird in Bezug auf diese IKE-Verbindung Initiator genannt, die andere Seite Responder.

Der Verbindungsaufbau geschieht über die zwei Exchanges `IKE_SA_INIT` und `IKE_AUTH`. `IKE_SA_INIT` dient zur Erzeugung der IKE SA, also dazu, den Schutz der IKE-Verbindung zu initialisieren. Ab `IKE_AUTH` findet der Meldungsaustausch verschlüsselt und geschützt statt. `IKE_`

`AUTH` dient primär der gegenseitigen Authentifizierung, und sekundär wird auch gleich noch eine erste Child-SA ausgehandelt. Ab dann gilt die Verbindung als zustande gekommen (established). Mit dem Austausch `CREATE_CHILD_SA` werden dann bei Bedarf weitere Child-SAs erzeugt. In Fehlersituationen oder für Zusatzfunktionalität wie automatische Mode Config findet ein `INFORMATIONAL`-Exchange statt. Beispielhaft für verschiedene mögliche `INFORMATIONAL`-Exchanges ist in Abbildung 4.5 ein Austausch eines Notify-Payloads dargestellt. Mit `INFORMATIONAL`-Exchanges werden beispielsweise Ausnahmesituationen signalisiert, SAs gelöscht oder Rekeying initiiert.

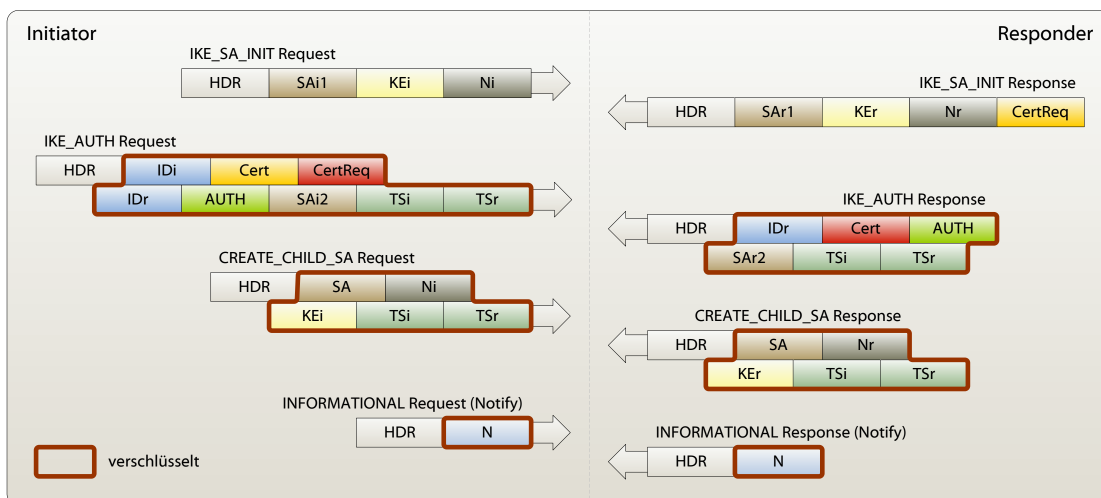


Abbildung 4.5.: IKEv2 Meldungsaustausch mit Payloads

4.2.4. NAT Traversal (NAT-T)

NAT-Traversal ist im Gegensatz zu IKEv1 nicht mehr ein separater Standard (vgl. [KSHV05]), sondern integrierter Bestandteil von IKEv2. NAT-Traversal besteht im Wesentlichen aus einem Verfahren, ein NAT zu detektieren (NAT Detection), eine Methode, die IKE-, ESP- und AH-Pakete in einem UDP-Paket zu verpacken, um sie "NAT-kompatibel" zu machen (UDP Encapsulation), und dem Senden von speziellen Paketen, um Timeouts von Verbindungsdaten auf NAT-Routern zu verhindern (NAT Keepalives).

Weil beim ursprünglichen Design von IPsec nicht damit gerechnet wurde, dass IP-Adressen und UDP/TCP-Ports unterwegs verändert werden, treten hier grosse Probleme auf. Bei AH ist hier sofort Schluss, weil die durch NAT veränderten Daten durch den HMAC kryptographisch gesichert sind. Bei ESP hingegen ist der UDP oder TCP Header für den NAT-Router gar nicht unverschlüsselt sichtbar. Es kann also gar kein NAPT stattfinden.

Primitive Lösungen beinhalten in der Regel einfach die fixe Weiterleitung von AH, ESP und IKE (Passthrough) an einen fixen Host hinter einem NAT. Was automatisch bedeutet, dass nur ein einziger Host hinter einem NAT-Router per IPsec kommunizieren kann. Weil für IKEv1 der Quellport zwingend auch Port 500 sein musste, beinhaltet diese IPsec-Passthrough-Funktionalität auch, dass IKE nicht dem normalen NAT unterzogen wird. Bei IKEv2 ist dies jedoch eher hinderlich als förderlich, denn IKEv2 würde auf Port 500 normal funktionieren, wenn die Pakete vom NAT-Router normal genattet würden. Deshalb benötigt auch IKEv2 selber Massnahmen für NAT-Traversal.

Ein wichtiges Merkmal von NAT-Traversal ist die Notwendigkeit, IKEv2-Pakete mit beliebigen Source Ports zu akzeptieren, und die Source-Adressen und -Ports einer IKE-Verbindung zu aktualisieren, wenn sie während der Lebenszeit der Verbindung ändern. Source-Adressen und -Ports können beispielsweise ändern, wenn ein NAT-Router neu gestartet wird.

NAT Detection (NAT-D)

Um erkennen zu können, ob sich die eigene oder die andere Seite hinter einem NAT irgendwelcher Art befindet, werden mit im `IKE_SA_INIT`-Request und der Antwort darauf spezielle NAT Detection Payloads mitübertragen. Diese erlauben der Gegenseite herauszufinden, ob das IP-Paket von der Gegenseite mit den gleichen IP-Adressen und UDP-Ports empfangen wurde wie es losgeschickt wurde. Man will also erkennen, ob die Source und/oder die Destination IP-Adressen und UDP-Ports unterwegs durch ein NAT verändert worden sind. Diese NAT-D-Payloads werden in den `IKE_SA_INIT`-Messages unmittelbar nach dem Nonce-Payload eingefügt, wie in Abbildung 4.6 ersichtlich.

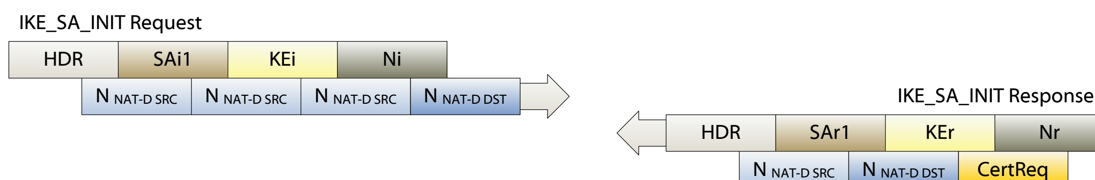


Abbildung 4.6.: NAT-Detection Payloads

Für diese NAT-D Payloads verwendet man Notify-Payloads der Typen `NAT_DETECTION_SOURCE_IP` und `NAT_DETECTION_DESTINATION_IP`. Der Aufbau eines solchen Payloads ist in Abbildung 4.7 dargestellt. Im Datenfeld der Notify-Payloads wird ein NAT-D-Hash übertragen, der aus dem SHA-1-Hash der beiden SPIs, der IP-Adresse sowie dem UDP-Port gebildet wird:

$$\text{NAT-D-Hash} = H_{\text{SHA-1}}(\text{SPI}_i||\text{SPI}_r||\text{IP-Adresse}||\text{UDP-Port})$$

Da der Initiator nicht in jedem Fall weiss, welche seiner unter Umständen mehrerer IP-Adressen als Absender-Adresse verwendet werden wird, beispielsweise im Falle von mehreren Netz-

werk-Interfaces, darf der Initiator mehrere NAT-D-Payloads des Typs `NAT_DETECTION_SOURCE_IP` senden. Der Responder muss dann jedes dieser Payloads auf Übereinstimmung prüfen. Ist keines mit dem Hash identisch, der sich aus der Adress/Port-Kombination des erhaltenen Pakets berechnet, so steckt der Initiator hinter einem NAT.

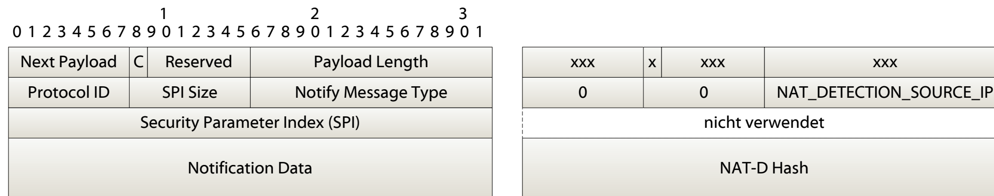


Abbildung 4.7.: Aufbau Notify-Payload für NAT-Detection

UDP Encapsulation

Wird ein NAT detektiert, so wird in IKEv2 sofort auf Port 4500 gewechselt, und anstatt ESP-Pakete direkt zu verschicken, werden sie mit einem zusätzlichen UDP-Header versehen, ebenfalls mit Source und Destination Port 4500. Diesen Vorgang nennt man UDP Encapsulation. Damit ESP und IKE-Pakete auf Port 4500 unterschieden werden können, wird IKE-Paketen ein Non-ESP-Marker vorangestellt, welcher aus 4 Null-Bytes besteht. Den Paketaufbau dieser Pakete mit UDP Encapsulation haben wir in Abbildung 4.8 dargestellt, während Abbildung 4.9 den Non-ESP Marker veranschaulicht.

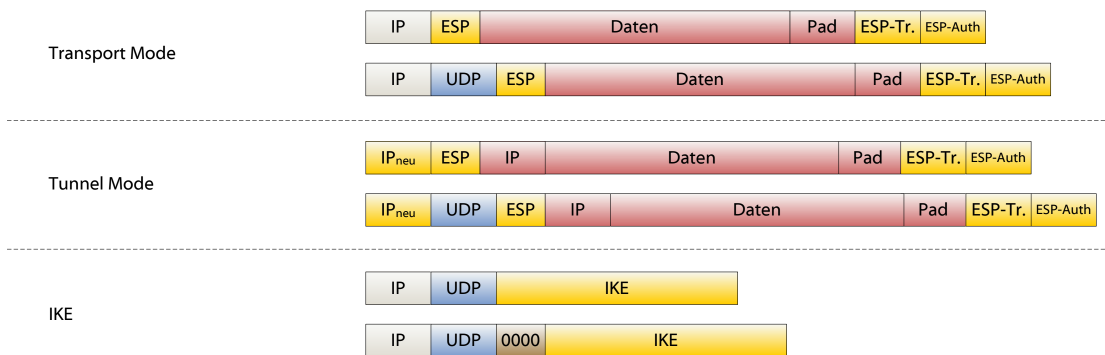


Abbildung 4.8.: UDP-Encapsulation

Dieser Wechsel auf Port 4500, auch Port Float genannt, ist erst seit Draft 06 von [HSV+05] im Standard. Der Non-ESP-Marker wurde in Draft 00/01 von [KSHV05] erstmals definiert. Es existieren auch Implementationen von älteren Standards, wie z.B. die Cisco-Variante mit ESP in UDP auf Port 10000.

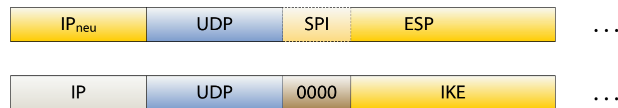


Abbildung 4.9.: Non-ESP Marker

NAT Keepalives

Wie wir in Abschnitt 4.1.2 erläutert haben, führen NAT-Router eine Tabelle über alle aktiven Verbindungen. Diese Tabelleneinträge werden nach einer gewissen Zeit der Inaktivität gelöscht. Damit diese Mappings nicht gelöscht werden, wird von einem Peer hinter einem NAT periodisch alle 20 Sekunden ein NAT Keepalive Paket gesendet. Das ist ein nach den im Abschnitt 4.2.4 beschriebenen Regeln verpacktes Datenbyte `0xFF`, ohne vorangestelltem Non-ESP Marker. Ein solches Paket haben wir in Abbildung 4.10 dargestellt.

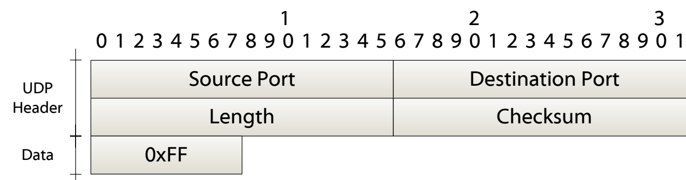


Abbildung 4.10.: NAT-Keepalive

4.2.5. Dead Peer Detection (DPD)

Wie NAT Traversal auch ist Dead Peer Detection (DPD) ein integraler Bestandteil von IKEv2 geworden, was bei IKEv1 noch nicht der Fall war (vgl. [HBR04]).

IKE ist ein UDP basiertes Protokoll, das sehr wenig Pakete über sehr lange Zeit austauscht. Ohne spezielle Massnahmen zu ergreifen, würde man erst Stunden später erkennen, wenn der Host am anderen Ende einer IKE-Verbindung nicht mehr aktiv ist.

Da es aber wünschenswert ist, auf den Ausfall einer IKE-Verbindung innert nützlicher Frist reagieren zu können, möchte man mit Hilfe von Dead Peer Detection periodisch Meldungen austauschen. Wird ein solcher DPD-Request nicht beantwortet, wird angenommen, der Peer sei nicht mehr aktiv, und entsprechend reagiert. Möglichkeiten sind beispielsweise die Verbindung neu zu initieren, die Verbindung zu löschen, oder auf Wartestatus ("hold") zu setzen.

Auf Protokollebene wird hierfür ein leerer `INFORMATIONAL`-Exchange verwendet, wenn für eine festgelegte Zeit keine IKE-Messages mehr empfangen wurden. Wenn dieser leere `INFORMATIONAL`-Request trotz Resends nicht quittiert wird, wird angenommen, dass die Gegenstelle nicht mehr aktiv ist, und entsprechend reagiert. Grundsätzlich ist DPD symmetrisch und freiwillig, es ist

also möglich, dass nur eine Seite DPD verwendet, die andere nicht, oder dass beide Seiten DPD verwenden.

4.3. Kernel-Schnittstelle

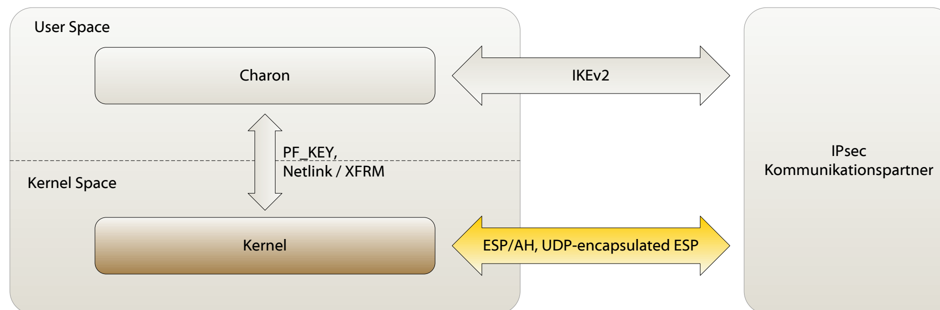


Abbildung 4.11.: IPsec in Linux

In den folgenden Abschnitten möchten wir auf die Schnittstelle zwischen strongSwan II Daemon im User Space und Linux Kernel eingehen. Als Einführung zeichnen wir in einem ersten Abschnitt kurz die Geschichte der IPsec Implementation in Linux nach, um dann die zwei verfügbaren Schnittstellen genauer anzusehen. Die in Abschnitt 4.3.2 beschriebene `PF_KEY` -Schnittstelle beschreiben wir hier nur grob, da strongSwan II momentan nur die in Abschnitt 4.3.3 ausführlich beschriebene `Netlink/XFRM` -Schnittstelle verwendet.

4.3.1. IPsec im Linux Kernel

Die erste vollständige und frei verfügbare Implementation von IPsec für Linux war das gegen der 1990er Jahre veröffentlichte FreeS/WAN. Dieses implementierte sowohl den IKE Daemon im User Space als auch die als *Kernel IPsec Stack (KLIPS)* bekannte IPsec Unterstützung im Kernel. Ein ähnliches *Komplettpaket* entstand damals in Form des *KAME* Projekts auch für BSD-Derivate.

Interessanterweise basiert aber die mit dem 2.6er Release des Kernels veröffentlichte IPsec Implementation aber auf keiner dieser bereits vorhandenen Lösungen. So entwickelten Ende 2002 Alexey Kuznetsov and David S. Miller den auch als *native* oder *NETKEY* bekannte IPsec Stack von Grund auf neu. [wik]

Diese Implementation unterstützt sowohl die in [MMP98] veröffentlichte `PF_KEY` -Schnittstelle als auch die Linux spezifische `XFRM` -Schnittstelle. Diese beiden Schnittstellen stellen wir in den folgenden beiden Abschnitten genauer vor.

4.3.2. PF_KEY

Die *PF_KEY Key Managment API* wurde 1998 in der Version 2 als RFC 2367 [MMP98] veröffentlicht. Die erste Version dieses API wurde als Teil der IPv6 und IPsec Implementation von *4.4BSD-Lite* und des *Cisco ISAKMP/Oakley Key Management Daemon* entwickelt. Die zweite Version erweitert und verfeinert diese API. Ein massgeblicher Grund für die Veröffentlichung von PF_KEY als RFC war es die Portabilität von Key-Management-Applikationen zu erhöhen. So ist es dank der Standardisierung und der damit einhergegangenen, weit abgestützten Implementation der PF_KEY -Schnittstelle beispielsweise möglich, die beiden für BSD-Derivate entwickelten IKE-Lösungen `racoon` (KAME-Projekt) und `isakmpd` (OpenBSD) auch unter Linux zu betreiben. Was PF_KEY ausserdem auszeichnet ist die Möglichkeit, nicht nur das IPsec-Subsystem zu konfigurieren, sondern beispielsweise auch für OSPFv2 [Moy98] oder RIPv2 [BA97] Schlüssel an den Kernel zu übermitteln.

PF_KEY ist als Socket Protokoll Familie konzipiert, welche teilweise an die BSD Routing Sockets (PF_ROUTE) angelehnt wurde. Entsprechend definiert ein Grossteil des RFC die Nachrichten, welche über die PF_KEY -Sockets ausgetauscht werden können. Die abgedeckte Funktionalität erstreckt sich über das Hinzufügen und Abfragen von Informationen bezüglich Security Associations (SA) hin zur Löschung derselben.

Die Nachrichten selbst sind simpel aufgebaut. Sie bestehen aus einem generischen Header und jeweils vom Nachrichtentyp abhängigen Zusatzdaten. Auf die Daten einzelner Nachrichten wollen wir hier nicht genauer eingehen, allerdings ist wichtig zu beachten, dass alle Daten auf 64 Bit ausgerichtet werden müssen und die Längen der Pakete jeweils in Einheiten von 64 Bit Blöcken angegeben werden.

Üblicherweise öffnet eine Applikation im Userland einen PF_KEY -Socket und sendet die gewünschten Nachrichten an den Kernel. Dabei wird jede gesendete Nachricht auf allen anderen geöffneten PF_KEY -Sockets repliziert, auch auf dem des Senders. In bestimmten Situationen versendet der Kernel auch spontan Nachrichten. Diese Nachrichten werden nur an jene Sockets geschickt die dies auch explizit verlangt haben. Im Falle von IPsec tritt dies beispielsweise dann auf, wenn eine Policy für ein ausgehendes Paket eine bestimmte SA erfordert, diese aber noch nicht aufgebaut ist. Der Kernel wird dann eine SADB_ACQUIRE Nachricht an zuvor mit SADB_REGISTER registrierte Applikationen senden. Auch wenn eine SA verfällt, sprich ihre Soft- oder Hardlifetime abgelaufen ist, wird der Kernel eine Nachricht versenden, so dass ein Rekeying initiieren werden kann.

Was der PF_KEY Standard nicht abdeckt sind Security Policies. Die Verfasser des Standards vertreten diesbezüglich die Meinung, dass es wichtig ist zwischen Sicherheitsmechanismen und Sicherheitsrichtlinien zu trennen. Der Standard sieht aber ein Erweiterungskonzept vor, welches Implementationen erlaubt den Funktiosumfang zu erweitern und damit auch die Verwaltung von Policies zu ermöglichen. Da solche Extensions aber nicht im Standard festgehalten sind, besteht die Gefahr, dass die Portabilität darunter leidet.

4.3.3. Netlink/XFRM

XFRM ist wie PF_KEY eine Schnittstelle auf das IPsec-Subsystem im Linux Kernel. Im Gegensatz zu PF_KEY ist XFRM aber nicht standardisiert und deshalb auch nicht inhärent portabel. Wie PF_KEY verwendet XFRM Sockets und darüber gesendete Nachrichten um mit dem Kernel zu kommunizieren. Im Gegensatz zu PF_KEY definiert XFRM aber keine eigene Socket Familie sondern nur die Nachrichten, die ausgetauscht werden. Die eigentliche Kommunikation findet über Netlink statt.

Netlink

Netlink ist eine Schnittstelle um Daten zwischen User Land und Linux Kernel Modulen auszutauschen. Der Zugriff erfolgt dabei über Standard Sockets, aus dem User Land, beziehungsweise einer speziellen API, aus den Kernel Modulen. Listing 4.1 zeigt die Erstellung eines solchen Sockets. Die Protokollfamilie muss dabei immer auf PF_NETLINK festgelegt sein. Da Netlink ein Datagramm basierter Dienst ist, kann als Sockettyp sowohl SOCK_RAW als auch SOCK_DGRAM verwendet werden, wobei dies letztlich aber keinen Unterschied macht. Der einzige variable Parameter nl_family legt sowohl das Protokoll als auch das Kernel Modul fest, mit denen kommuniziert wird. Ein gültiger Wert dafür ist beispielsweise NETLINK_ROUTE um die Routing Informationen des Systems zu manipulieren. Die für uns relevante Netlink Familie ist NETLINK_XFRM.

Listing 4.1: Netlink Socket

```
1       nl_socket = socket(PF_NETLINK, SOCK_RAW, nl_family);
```

Wie bereits erwähnt, ist Netlink ein auf Datagrammen basierender, verbindungsloser Dienst, welcher ganz ähnlich wie UDP funktioniert. So werden Nachrichten üblicherweise über die Funktionen sendto und recvfrom gesendet bzw. empfangen. Wie auch bei UDP ist nicht garantiert, dass Datagramme beim Empfänger ankommen. Die Flags im Header von Netlink-Nachrichten bietet aber Möglichkeiten, ein gewisses Mass an Zuverlässigkeit zu erreichen, sofern die Applikation dies erfordert.

Nachrichtenformat

Die Nachrichten, die über Netlink ausgetauscht werden bestehen immer aus einem Standardheader und vom Nachrichtentyp abhängigen Nutzdaten variabler Länge. Listing 4.2 zeigt den Header als C-Struct. Die ersten beiden Member des Structs sollten selbsterklärend sein. Mit in nlmsg_flags übergebenen Flags kann gesteuert werden wie Nachrichten bearbeitet und interpretiert werden. So wird beispielsweise über NLM_F_REQUEST festgelegt, dass es sich bei einer

Nachricht um einen Request handelt. Das Flag `NLM_F_ACK` andererseits verlangt eine Bestätigung, wenn der Request erfolgreich war. Die Sequenznummer in `nlmsg_seq` dient dem Absender dazu, Anfragen ihrer jeweiligen Antwort zuzuordnen. Desweiteren sollte der Absender seine Prozess-ID in `nlmsg_pid` übergeben. Eine vom Kernel verschickte Nachricht trägt immer die PID Null.

Listing 4.2: Netlink Header

```
1 struct nlmsghdr {
2     __u32    nlmsg_len;     // Länge der Message inkl. Header
3     __u16    nlmsg_type;    // Nachrichtentyp
4     __u16    nlmsg_flags;   // Header Flags
5     __u32    nlmsg_seq;     // Sequenznummer
6     __u32    nlmsg_pid;     // PID des Senders
7 }
```

Um die Pakete zu adressieren kommt das Struct `sockaddr_nl` zum Einsatz, welches in Listing 4.3 aufgeführt ist. Weitere Informationen zu diesen beiden Structs und ihren Membern liefert auch die Manpage `netlink(7)`.

Listing 4.3: Netlink Adressen

```
1 struct sockaddr_nl {
2     sa_family_t    nl_family;   // AF_NETLINK
3     unsigned short nl_pad;      // NULL
4     pid_t          nl_pid;      // PID des Empfängers
5     __u32          nl_groups;   // Multicast Gruppen Maske
6 }
```

Um den Aufbau und die Manipulation der Nachrichten zu vereinfachen werden einige Makros zur Verfügung gestellt, die wir jetzt im Zusammenhang mit dem eigentlichen Aufbau der Nachrichten ansehen wollen. Abbildung 4.12 zeigt beispielhaft den Aufbau einer solchen Nachricht. Die Länge des Headers und der Payload müssen jeweils auf die nächste `NLMSG_ALIGNTO`[4] Grenze aufgerundet werden. Die Makros nehmen uns diese Arbeit grösstenteils ab. So liefert `NLMSG_LENGTH` aus der Länge der Payload sogleich den Wert für `nlmsg_len` im Netlink-Header, einschliesslich des erforderliche Paddings. Analog liefert uns der Aufruf des Makros `NLMSG_DATA` ausgehend von von einem Pointer auf den Header einen Pointer auf die Nutzdaten der Nachricht. Ein Beispiel für das Zusammenspiel der Makros ist in Listing 4.4 auf Seite 24 zu sehen.

[4] Üblicherweise auf 4 Bytes festgelegt.

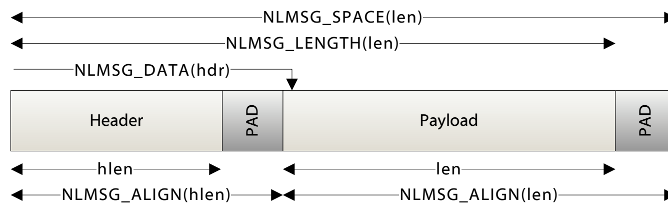


Abbildung 4.12.: Messageaufbau mit Makros

rtnetlink

Die Nachrichtentypen und deren Payloads werden durch die jeweiligen `Netlink`-Familien definiert. Prinzipiell können die Payloads beliebig aufgebaut sein. Infolgedessen bietet die umfangreichste und am weitesten entwickelte `Netlink` Protokoll Familie, `NETLINK_ROUTE` oder `rtnetlink`, eine Reihe von eigenen Makros an, um die Daten innerhalb der Payloads weiter zu strukturieren. Das Prinzip ist ganz ähnlich wie bei den `Netlink`-Makros zuvor. Jede Payload besteht aus Hauptdaten denen, jeweils mit einem `rtattr`-Struct ausgezeichnete, Datenblöcke angehängt werden können. Der Aufbau einer solchen Payload ist in Abbildung 4.13 dargestellt. Wie ersichtlich ist, kann jeweils über das Makro `RTA_NEXT` auf die nächstfolgende Teilpayload zugegriffen werden. Dieses Konzept (und somit auch die Makros) werden auch von `XFRM` zur Strukturierung der Payloads verwendet.

Es sei noch erwähnt, dass ein derartiges Aneinanderreihen prinzipiell auch auf Ebene der `Netlink`-Messages möglich ist. Da wir dies aber nie verwendet haben, belassen wir die Behandlung dieses Themas bei diesem einen Satz.

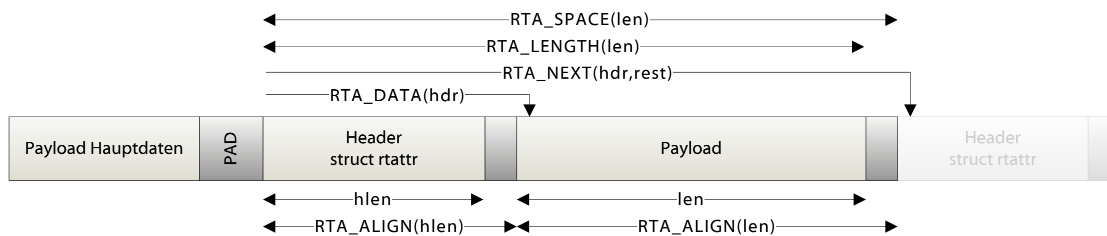


Abbildung 4.13.: Payload mit `rtnetlink` Makros

`Netlink` ist ein mächtiges Werkzeug, leider jedoch immer noch recht spärlich dokumentiert. So können als Informationsquelle oft nur die Kernel-Sourcen zu Rate gezogen werden. Einen Einstieg und weiterführende, wenngleich nicht mehr ganz taufrische, Informationen versucht [Hor04] zu geben.

XFRM

Damit kommen wir nun zu einigen Anwendungsbeispielen und dem eigentlichen Kern dieses Kapitels. XFRM ist das Netlink Protokoll, um auf das IPsec-Subsystem im Linux Kernel zuzugreifen. Dazu werden, analog zu PF_KEY, Nachrichten definiert, welche die Manipulation von SAs und Policies erlauben. Ebenso wie bei PF_KEY dienen einige Nachrichtentypen dem Datenaustausch mit dem Kernel, während andere wiederum Applikationen im User Space über Ereignisse wie ablaufende Hard- oder Softlifetimes oder Aufforderungen zum Aufbau einer benötigten SA benachrichtigen sollen.

Sehen wir uns den Aufbau einer XFRM -Message mit den jeweiligen Makros am Beispiel der Einrichtung einer SA an. Listing 4.4 zeigt einen ersten Ausschnitt der Funktion add_sa in der Datei kernel_interface.c. Die ganze Nachricht wird in einem Buffer direkt auf dem Stack erstellt. Zuerst holen wir uns einen Pointer auf einen Netlink Header auf dem wir dann die Flags (siehe Nachrichtenformat, Seite 21), den Nachrichtentyp und die Länge setzen. Um Letzteres korrekt festzulegen verwenden wir das entsprechende Makro, welches uns zur übergebenen Länge die Länge des Headers (einschliesslich Padding) addiert. Als nächstes werden die Nutzdaten der Nachricht angefügt. Dazu verwenden wir das Makro NLMSG_DATA welches uns, wie in Abbildung 4.12 gezeigt, einen Pointer auf die Payload der Message liefert. Diesen casten wir noch auf den entsprechenden Datentypen um die Daten anschliessend abzufüllen.

Listing 4.4: Message Aufbau in add_sa

```
1 unsigned char request[BUFFER_SIZE];
2 struct nlmsghdr *hdr = (struct nlmsghdr*)request;
3 hdr->nlmsg_flags = NLM_F_REQUEST | NLM_F_ACK;
4 hdr->nlmsg_type = XFRM_MSG_NEWSA;
5 hdr->nlmsg_len = NLMSG_LENGTH(sizeof(struct xfrm_usersa_info));
6
7 struct xfrm_usersa_info *sa = (struct xfrm_usersa_info*)
     NLMSG_DATA(hdr);
8 sa->saddr = me->get_xfrm_addr(me);
9 sa->id.daddr = other->get_xfrm_addr(other);
```

Als nächstes folgen die bereits erwähnten und an die Payload angehängten Zusatzdaten. Listing 4.5 zeigt am Beispiel des Verschlüsselungsalgorithmus wie Zusatzdaten an die Nachricht angehängt werden. Dasselbe Verfahren wird auch für den Authentisierungsalgorithmus oder die Konfiguration der UDP-Encapsulation verwendet (siehe Abschnitt 5.3.2).

Das Vorgehen ist wie folgt. Mit dem selbst definierten Makro XFRM_RTA wird ein Pointer auf das erste rtattr-Struct nach den Hauptdaten geholt. Dieser wird anschliessend jeweils mit dem Makro XFRM_RTA_NEXT verschoben um weitere Daten anzuhängen. Um jeweils die eigentlichen

Nutzdaten an die rtattr-Structs anzufügen verwenden wir die beiden Makros RTA_LENGTH und RTA_DATA (siehe dazu rtnetlink, Seite 23). Ausserdem wird, wie in Zeile 10 des Listing ersichtlich, jeweils die Länge der gesamten Netlink Nachricht nachgeführt.

Listing 4.5: Message Aufbau in add_sa

```
1 /* #define XFRM_RTA(nlh, x) ((struct rtattr*)(NLMSG_DATA(nlh) +
      NLMSG_ALIGN(sizeof(x)))
2  * #define XFRM_RTA_NEXT(rta) ((struct rtattr*)(((char*)(rta))
      + RTA_ALIGN((rta)->rta_len)))
3  */
4 struct rtattr *rthdr = XFRM_RTA(hdr, struct xfrm_usersa_info);
5 if (enc_alg->algorithm != ENCR_UNDEFINED)
6 {
7     rthdr->rta_type = XFRMA_ALG_CRYPT;
8     /* ... */
9     rthdr->rta_len = RTA_LENGTH(sizeof(struct xfrm_algo) +
         key_size);
10    hdr->nlmsg_len += rthdr->rta_len;
11    /* ... */
12    struct xfrm_algo* algo = (struct xfrm_algo*)RTA_DATA(rthdr)
         ;
13    algo->alg_key_len = key_size;
14    /* ... */
15    rthdr = XFRM_RTA_NEXT(rthdr);
16 }
```

Grafisch aufgearbeitet sieht die erstellte Nachricht etwa wie in Abbildung 4.14 dargestellt aus. Wobei anzumerken ist, dass die Darstellung recht einfach gehalten ist und einige Details weggelassen wurden.

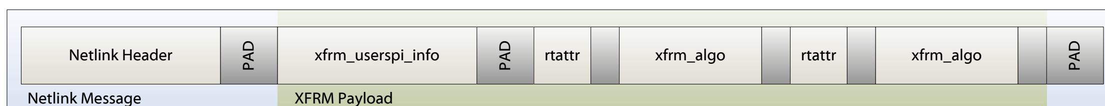


Abbildung 4.14.: XFRM Nachricht um eine SA zu erstellen

Limitierungen

Der Funktionsumfang von XFRM ist leider nicht genau deckungsgleich mit demjenigen von PF_KEY—die Funktionalität überlappt zwar zu grossen Teilen, beide Schnittstellen aber bieten Dinge, wel-

che die andere nicht bietet. Im Rahmen unserer Arbeit sind wir insbesondere bei der Benachrichtigung durch den Kernel im Falle einer Adressänderung an UDP enkapsulierten ESP-Paketen auf Grenzen von `XFRM` gestossen. Siehe hierzu die Vision in Abschnitt 7.1.5.

4.4. UML — User Mode Linux

User Mode Linux erlaubt mehrere virtuelle Linux-Systeme als Applikationen innerhalb eines gewöhnlichen Linux-Systems auszuführen. Da jedes virtuelle System als normaler Prozess im User Space abläuft, bietet dieser Ansatz eine Möglichkeit, um auf sichere Art und Weise mehrere virtuelle Linux-Systeme auf ein und derselben Hardware auszuführen, ohne dabei die Konfiguration oder Stabilität des Host-Systems zu kompromittieren. [wik]

Die auf UML basierende strongSwan-Testsuite wurde von Eric Marchionni and Patrik Rayo innerhalb ihrer Diplomarbeit an der Zürcher Hochschule Winterthur entwickelt. Für weiterführende Informationen möchten wir deshalb auf ihre Arbeit [MR04] verweisen.

5. Design / Implementation

5.1. Methodik

Da wir aufgrund der Ausgangslage ausschliesslich im Rahmen des bestehenden Quellcodes von strongSwan arbeiteten, waren wir gezwungen, unser Vorgehen entsprechend anzupassen. Was Richtlinien und Stilfragen abgelangt, passten wir uns voll und ganz den Vorgaben von strongSwan II an. Bei Designfragen waren wir insofern eingeschränkt, als dass wir stets eine Lösung finden mussten, welche sich gut in den bestehenden Rahmen von strongSwan II einpassen würde. Auf diese Aspekte gehen wir im Folgenden kurz ein, bevor wir die realisierten Teilsysteme im Detail beschreiben.

5.1.1. Programmierrichtlinien

Die Programmierrichtlinien waren durch die Vorgaben von Jan Hutter und Martin Willi gegeben. Wir verzichten an dieser Stelle auf eine komplette Wiedergabe der Programmierrichtlinien, und verweisen stattdessen auf [HW05]. Für das Verständnis der folgenden Abschnitte möchten wir aber die grundlegenden Merkmale des Quellcodes kurz beschreiben.

Jan Hutter und Martin Willi verwendeten für den im Rahmen ihrer Diplomarbeit entwickelten Charon einen objektorientierten Ansatz in C. Implementiert wurde das mittels Funktionszeigern in `struct`s, mit expliziter `this`-Übergabe und ohne Vererbung. Gegenüber herkömmlichem C gewinnt man also den objektorientierten Ansatz, welcher objektorientiertes Design erlaubt, gegenüber C++ verliert man aber viele Automatismen und "syntactic sugar", was mehr explizite Schreibarbeit zur Folge hat, und es nicht immer erlaubt, wichtige Design-Prinzipien wie Don't Repeat Yourself (DRY) konsequent umzusetzen.

5.1.2. Source Code Management

Wir haben den Source Code unserer Arbeitsversion von strongSwan II in einem eigenen Subversion-Repository verwaltet, während am Repository von strongSwan II parallel laufend weiterentwickelt wurde. Um diese Situation zu bewältigen, griffen wir auf die bewährten Vendor-Branches zurück. Bei dieser Arbeitstechnik wird in unserem Repository unter `/vendor` ein Vendor-Branch mitgeführt, in welchem jeweils die gleiche Version des Quellcodes wie im Trunk abgelegt wird, einfach ohne unsere Modifikationen. Davon wird bei jedem Code-Import ("Merge") ein Vendor-Tag erzeugt, der den Zustand des strongSwan-Repositories zum Zeitpunkt des Merges darstellt.

Auf diese Art und Weise konnten wir zu jedem Zeitpunkt ein Delta ("Patch", "Diff") mit nur unseren Änderungen erzeugen, oder ein Delta zwischen der letzten importierten Version von strongSwan II gegenüber der neuen im Repository. Beides hat sich als enorm hilfreich erwiesen beim Mergen von neuem strongSwan II Code, oder beim Erzeugen von Patches gegen strongSwan II. Des weiteren erlaubt diese Arbeitsweise die Nutzung der Merge-Automatismen von Subversion. Diese nehmen einem die Konfliktbereinigung im Detail zwar nicht ab, unterstützen aber den ganzen Prozess des Mergens.

Die Arbeit mit Vendor-Branches ist im Subversion-Buch [CSFP06] ausführlich beschrieben.

5.1.3. Issue Tracking, Ticketing

Um den Überblick über die offenen Tasks sowie auftretende Probleme und Aufträge zu behalten, haben wir mit dem Ticketing-System von Trac gearbeitet. Wir haben Trac genutzt, um unsere eigenen Arbeitspakete zu verwalten, aber auch zur Kommunikation mit Martin Willi, dem Lead Developer von strongSwan II. Weitere Informationen zu unseren Anregungen haben wir in Abschnitt 5.5 dokumentiert.

5.2. Architektur im Überblick

Da für das Verständnis der Implementationsdetails der von uns realisierten Teilsysteme eine grundsätzliche Übersicht über die Gesamtarchitektur von `charon` notwendig ist, beschreiben wir diese hier kurz. Abbildung 5.1 gibt hierzu den Überblick über die wichtigsten Bestandteile und ihre Zusammenhänge.

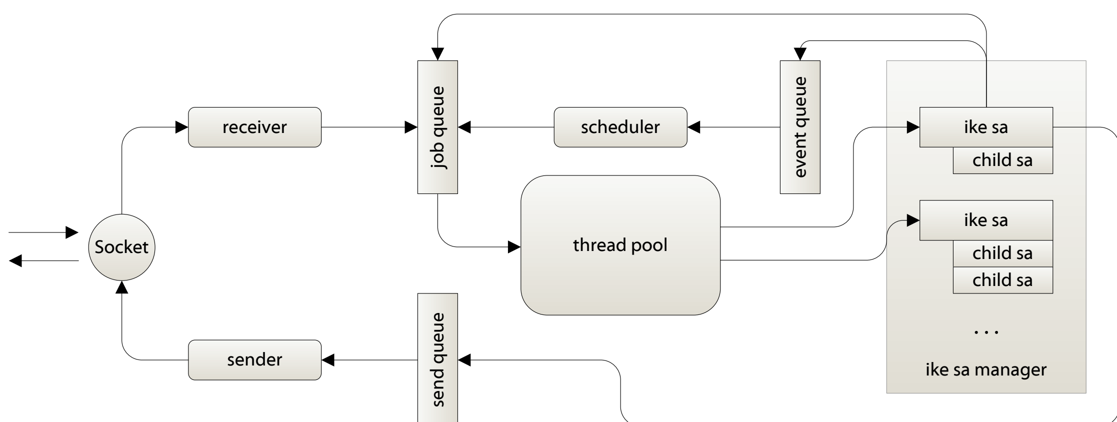


Abbildung 5.1.: Architektur von `charon`

Über den Socket werden Pakete gesendet und empfangen. Zu diesem Zweck existiert ein Receiver-Thread, der Pakete vom Socket liest und in der Job-Queue ablegt, sowie ein Sender-Thread,

der Pakete aus der Send-Queue nimmt und verschickt.

Die Job-Queue wird von den Worker-Threads des Thread Pools abgearbeitet. Ein vom Receiver-Thread in der Job-Queue abgelegtes Paket wird also später von einem Worker-Thread geparst und verarbeitet. Der Worker-Thread holt sich vom IKE-SA-Manager dazu die zu einem Job gehörende IKE-SA. Der IKE-SA-Manager implementiert das Checkin-Checkout-Pattern, d.h. die IKE-SA bleibt während dem Zugriff des Worker-Threads für andere Threads gesperrt.

Um ein Paket zu versenden, wird es der Send-Queue hinzugefügt, wo es später vom Sender-Thread abgeholt und versandt wird.

Mit der Event-Queue steht eine Timer-Facility zu Verfügung, quasi ein `charon` -interner `crond`. Ein Job kann zusammen mit einer Zeitangabe in die Event-Queue gespeichert werden, wo er zu gegebener Zeit vom Scheduler-Thread abgeholt und in die Job-Queue verschoben wird.

Die IKE-SA implementiert das State-Pattern. Die Zustände, welche von der IKE-SA durchlaufen werden, sowie die zugehörigen Exchanges sind in Abbildung 5.2 dargestellt.

Mehr Informationen zu diesen Patterns sind beispielsweise in [GHJV95] und [BMR+96] zu finden.

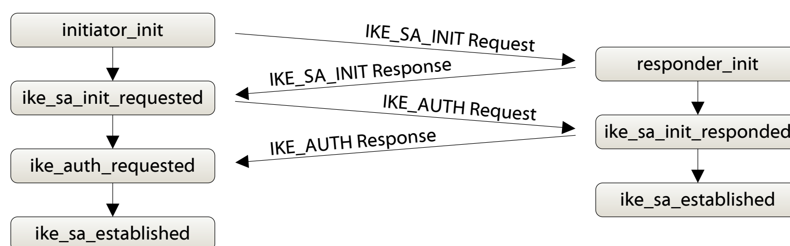


Abbildung 5.2.: Zustände der IKE-SA

5.3. Teilsysteme

5.3.1. Network Sockets

Bei der Implementation der Netzwerk Sockets gab es stets eines zu beachten, der IKEv2-Daemon `charon` durfte den IKE-Daemon `pluto` in keinster Weise beeinträchtigen – beide sollten zur gleichen Zeit betrieben werden können. Dazu sollten aber keine Änderungen an `pluto` nötig sein. Im folgenden beschreiben wir zuerst das Problem und die Situation in strongSwan II vor unseren Änderungen. Im Anschluss erläutern wir die vorgenommenen Anpassungen im Detail.

Situation

Die Problematik der Situation besteht darin, dass prinzipiell beide Daemons den gleichen Dienst anbieten wollen. Auf Ebene der Netzwerk Sockets unterscheiden sich IKE und IKEv2 nämlich

nicht. So verwenden beide UDP als Transportprotokoll unter Verwendung von Port 500 im "Normalbetrieb" und Port 4500 im Falle von NAT-T. Grundsätzlich ist es zwar möglich, dass mittels der Socket Option `SO_REUSEADDR` mehrere Prozesse einen Aufruf von `bind` auf dasselbe [IP-Adresse, Port]-Tupel machen können (dies gilt nur für `PF_INET/SOCK_DGRAM`-Sockets). Dies löst jedoch das Problem noch nicht. Zwar werden damit Broad- und Multicast-Datagramme beiden Prozessen ausgeliefert doch die erwarteten Unicast-Datagramme landen unglücklicherweise nur bei einem der Prozesse. Dieser müsste die empfangenen Pakete analysieren und solche die nicht für ihn bestimmt sind über irgendeine Form von IPC an den anderen Daemon weiterleiten. Eine solche Lösung würde ohne Zweifel erhebliche Modifikationen an `pluto` bedingen.

Die einzige saubere Lösung für dieses Problem stellen Raw-Sockets dar. Ein Raw-Socket ist ein spezieller Netzwerksocket, welcher den Zugriff auf die *rohen* IP-Datagramme ermöglicht. Wichtig ist diesbezüglich, dass jedes vom Kernel empfangene Datagramm an alle Raw-Sockets gleichermassen weitergegeben wird – sofern das gebundene Upper-Layer-Protokoll übereinstimmt (gültige Protokolle, siehe [RP94]). Erstellt wird ein Raw-Socket mit folgendem Systemaufruf:

```
1 raw_socket = socket(PF_INET, SOCK_RAW, int protocol);
```

Raw-Sockets ermöglichen einem Prozess den gesamten IP-Verkehr mitzuschneiden ohne dabei andere Prozesse und deren Sockets zu beeinträchtigen. Diese Eigenschaft können wir ausnutzen um in `charon` alle Pakete zu empfangen ohne den Empfang derselben in `pluto` zu beeinflussen. Der Code und das Verhalten von `pluto` ändern sich dabei nicht. IKEv2-Pakete welche er konsequenterweise empfangen wird, werden von ihm einfach ignorieren. Dasselbe gilt für `charon` und empfangene IKE-Pakete. Da aber ein Raw-Socket inhärent nicht auf einen einzelnen Port eingeschränkt ist, sondern wie in unserem Fall einfach alle UDP-Datagramme empfängt, wäre dieses Aussieben in `charon` um einiges aufwändiger. Aus diesem Grund wird auf dem Socket ein Filter installiert (siehe dazu Berkeley Packet Filter (BPF), Seite 33) welcher diese Arbeit an den Kernel delegiert.

Die Socket-Implementation von strongSwan II verfolgte bereits den beschriebenen Raw-Socket-Ansatz und ermöglichte damit `pluto` parallel zu betreiben. Die verwendeten Sockets wurden dabei in der Klasse `socket_t` gekapselt. Der interne Aufbau dieser Klasse war wie folgt – ein ähnlicher Ansatz verfolgt auch `pluto`:

- Über einen normalen, an [`INADDR_ANY`, 500] gebundenen, UDP-Socket wurden via `ioctl/SIOCGIFCONF` alle lokalen Interfaces enumieriert. An jedes dieser Interfaces wurde dann ein Raw-Socket gebunden. Ausserdem wurden die Interfaces intern in einer Liste aggregiert und konnten über die Methode `is_listening_on` abgefragt werden.
- Gesendet wurden die Packete über den oben erwähnten gewöhnlichen UDP-Socket.
- Empfangen wurden sie über die Raw-Sockets – da pro Interface einer vorhanden war, mit komplexer Logik um die Filedeskriptor-Sets für den Aufruf von `select` aufzubauen.

Implementation

Angesichts dieser Ausgangslage stellten wir einige Anforderungen an die angepasste Netzwerk-Schnittstelle. Erstens galt natürlich nach wie vor die Devise, das Zusammenspiel mit `pluto` nicht zu gefährden. Zweitens benötigten wir zusätzlich zu einem Socket auf Port 500 einen auf Port 4500. Drittens benötigten wir eine Liste der IP-Adressen aller Interfaces um die NAT-D Payloads zu erzeugen (siehe Abschnitt 5.3.3). Viertens sollte die Änderung möglichst ohne Anpassungen an bestehenden Implementationen von Sender- und Receiver-Threads auskommen. Im folgenden beschreiben wir nun unsere Implementation, welche diesen Anforderungen zu genügen versucht.

Die letzte der oben genannten Anforderungen erforderte eine konzeptuelle Neudefinition der Klasse `socket_t`. In strongSwan II wurde die Klasse als Abstraktion eines realen Sockets entwickelt. Dem Konstruktor wird der gewünschte Port übergeben und über die zwei Methoden `sender` und `receiver` werden dann Pakete gesendet bzw. empfangen. Um also auch Pakete auf Port 4500 zu empfangen, wäre die naheliegenste Lösung eine zweite Instanz der Socket-Klasse zu erzeugen. Diese Lösung wäre aber mit erheblichen Änderungen an den Threads oder sogar den Queues verbunden gewesen. Denn wie sollten wir die Pakete zentral empfangen, wenn der Aufruf von `select` innerhalb des Sockets gekapselt ist? Sollten wir dazu einen zweiten Receiver-Thread einführen? Wer entscheidet über welchen Socket ein Paket gesendet wird, der Sender-Thread? Aufgrund welcher Informationen? Sollten wir dazu eine zweite Send-Queue einführen? Aus welchem der beiden Sockets sollen wir die Liste der lokalen IP-Addressen beziehen? Macht es Sinn, diese Liste zweimal zu erstellen?

Solche Fragen führten letztlich zu folgendem Schluss: Wir kapseln diese ganze Logik in einer neu entwickelten Socket-Klasse und lagern die Enumeration der Interfaces in eine separate Klasse aus. Dabei änderten wir die Socket-Klasse konzeptuell so ab, dass sie nicht mehr einer Abstraktion eines einzelnen realen Sockets entspricht, sondern nunmehr zwei Sockets repräsentiert.

Um die Kompatibilität mit `pluto` zu gewährleisten, verwenden wir in unserer Implementation ebenfalls Raw-Sockets. Allerdings nur einen einzigen. Auf diesem installieren wir den weiter unten beschriebenen Berkeley Packet Filter (BPF). Verwendung findet dieser Socket ausschliesslich im Empfang von Paketen. Um Pakete zu versenden, verwenden wir zwei normale UDP-Sockets jeweils an einen der beiden Ports und `INADDR_ANY` gebunden. Um nicht mit `pluto` in Konflikt zu geraten setzen wir auf diesen Sockets die Socket Option `SO_REUSEADDR`.

Die zwei Send-Sockets unterscheiden sich von einem gewöhnlichen Socket in folgenden zwei Punkten:

Per Socket Policies Um zu verhindern, dass über die Sockets verschickter IKEv2-Traffic in einem IPsec-Tunnel verschwindet, werden auf den Sockets jeweils zwei sogenannte *Per Socket Policies* vom Typ `IPSEC_POLICY_BYPASS` installiert — eine ein- die andere ausgehend. Die Policies verhindern, dass über die Sockets verschickte Pakete durch systemweite IPsec-Policies beeinflusst werden.

UDP_ENCAP Die zweite Besonderheit betrifft nur den Send-Socket auf Port 4500. Auf diesem Socket setzen wir über einen Aufruf von `setsockopt` die Option `UDP_ENCAP` auf den Wert `UDP_ENCAP_ESPINUDP`. Nötig ist dies, damit der Kernel UDP-enkapsulierte ESP-Pakete, die über diesen Port eingehen, überhaupt entpackt. Die Option muss für mindestens einen an Port 4500 gebundenen Socket gesetzt werden (egal in welchem Prozess).

Die Funktion `sender` wurde analog zur Reduktion der Anzahl Sockets einfacher und übersichtlicher. So ist kein `select` (mit aufwändiger FD-Aggregation) mehr nötig, da wir nur noch auf dem einen einzigen Raw-Socket Pakete empfangen. Das Vorgehen beim Empfang eines Paketes ist folgendes:

1. Erzeugen der Source- und Destination-Host Objekte vom Typ `host_t` direkt aus den Informationen im IP- bzw. UDP-Header. Dies geschieht über eine zu `host_t` hinzugefügte Methode recht effizient.

2. Berechnen des Offsets, an dem die eigentlichen Paketdaten beginnen. Im Normalfall beträgt der Offset 28 Bytes (Länge von IP- und UDP-Header). Falls das Paket aber auf Port 4500 empfangen wurde, erhöht sich der Offset um die Grösse des Non-ESP-Markers um vier auf 32 Bytes.

3. Zuletzt werden die Paketdaten in einen `chunk_t` kopiert und als `packet_t` zurückgegeben.

Die Anpassungen in `receiver` sind nicht so gravierend wie in der eben beschriebenen Methode. Ausserdem ist das Vorgehen beim Senden eines Paketes nun leicht aufwändiger. Es lässt sich wie folgt zusammenfassen:

1. Als ersten Schritt prüfen wir den Source-Port des zu sendenden Pakets. Aufgrund des ermittelten Ports wählen wir den Socket um das Paket zu versenden.

2. Im Falle von Port 500 wird das Paket direkt versendet.

3. Falls das Paket von Port 4500 gesendet werden soll, müssen wir noch den Non-ESP-Marker vor den Paketdaten plazieren – allerdings nur wenn das Paket kein Keepalive ist (siehe dazu auch Abschnitte 4.2.4 und 4.2.4). Ist dies erledigt, senden wir das Paket über den zweiten Send-Socket.

Das schöne and dieser Lösung ist, dass keinerlei Änderungen an den Threads und den Queues vorgenommen werden mussten. So haben sich weder die Funktions-Signaturen noch die grundlegende Semantik – das Senden und Empfangen von Paketen – geändert.

Damit kommen wir zur letzten noch unerfüllten Anforderung – der Enumeration der lokalen IP-Adressen. Da innerhalb des Sockets keine solche Enumeration mehr nötig ist, steht uns die

Liste der Interfaces nicht mehr implizit zur Verfügung. Also haben wir diese Funktionalität in eine separate Klasse, `interface_t`, ausgelagert und gleich auch das Vorgehen zur Enumeration angepasst. Denn über die in `<ifaddrs.h>` definierte Funktion `getifaddrs` lassen sich die IP-Adressen der lokalen Interfaces viel einfacher in einer Liste aggregieren. Über die Methode `is_local_address` lässt sich die gleiche Information beziehen wie vorhin über die Methode `is_listening_on`. Ausserdem kann über die Funktion `get_addresses` die Liste der Adressen bezogen werden.

Mehr über die Programmierung mit Berkeley Sockets kann in [Ste92] nachgelesen werden.

Berkeley Packet Filter (BPF)

Wie bereits erwähnt installieren wir auf dem Raw-Socket einen Filter zur Paketselektion im Kernel. Dabei sind wir vom bereits für strongSwan II entwickelte Filter ausgegangen und haben ihn so erweitert, dass er mit der UDP-Encapsulation kompatibel ist. Im Folgenden dokumentieren wir die verwendete Filter-Engine, Berkeley Packet Filter, und den erweiterten Filter selbst.

Die Berkeley Packet Filter (BPF) sind kurze Programme die als Arrays von Instruktion definiert werden. Der Instruktionssatz erinnert ein wenig an eine einfache Assemblersprache. So operiert jede Instruktion auf einem Pseudo-Rechner der aus einem Akkumulator, einem Index-Register, Scratch-Memory und einem impliziten Programm-Counter besteht. Um auch seinen Zweck als Paketfilter zu erfüllen, operiert der Filter ausserdem direkt auf den Paketdaten. Alle Verzweigungen innerhalb eines Filters sind stets nach vorne gerichtet und das Programm wird jeweils durch eine Return-Instruktion abgeschlossen. Jede Instruktion wird als Struct `bpf_insn` definiert, welches die folgenden Member hat:

`code` Die eigentliche Programmanweisung. Die verwendeten Opcodes sind dabei semi-hierarchisch geordnet wobei insgesamt acht Instruktionsklassen wie `BPF_LD`, `BPF_ALU` oder `BPF_JMP` definiert sind. Die eigentlichen Instruktionen werden als Kombination der Klassen mit zusätzlichen Operatoren gebildet. Wir werden unten einige Beispiele dazu sehen.

`jt` Ausschliesslich für Sprunganweisungen verwendet.

`jf` Ausschliesslich für Sprunganweisungen verwendet.

`k` Die Interpretation dieses Parameters variiert je nach verwendeter Instruktion.

Um die Definition der Instruktionen zu vereinfachen werden zwei Makros angeboten. `BPF_STMT` hat zwei Parameter `code` und `k` und setzt die zwei ausschliesslich für Sprunganweisungen verwendeten Member des Structs auf Null. `BPF_JUMP` erlaubt die Definition von Sprunganweisungen in einer etwas intuitiveren Syntax (`k` als zweiten und nicht als letzten Member). Wie erwähnt wird der Opcode einer Instruktion als Kombination aus einer Instruktionsklasse und verschiedenen Operatoren gebildet. Sehen wir uns dazu die erste Anweisung des in Listing 5.1 abgedruckten Filters an. Die Kombination von `BPF_LD+BPF_B+BPF_ABS` besagt folgendes:

`BPF_LD` Als Instruktionsklasse werden damit alle Instruktionen ausgezeichnet, die einen Wert in den Akkumulator kopieren.

`BPF_ABS` Legt die Quelle der Kopieroperation fest – in diesem Fall Paketdaten mit einem fixen Offset, welcher als Parameter `k` übergeben wird. Als Quelle können ausserdem Konstanten `BPF_IMM` und Paketdaten mit variablem Offset `BPF_IND` dienen. In letzterem Fall wird der Wert von `k` und der des Index-Registers `X` addiert um den Offset zu berechnen.

`BPF_B` Dieser Wert gehört zu einer Reihe von Operatoren, welche die Grösse des zu kopierenden Blockes angeben. Wie der Name schon andeutet wird in diesem Fall ein einzelnes Byte kopiert. Weitere gültige Werte sind `BPF_H` und `BPF_W` um zwei bzw. vier Bytes zu kopieren. Die Grösse muss übrigens für `BPF_IMM` nicht angegeben werden.

Wir laden also ein einzelnes Byte aus den Paketdaten in den Akkumulator. Der fixe Offset ist hier `IP + 9`, kopiert wird somit das zehnte Byte des IP-Headers. Dieses bezeichnet die Protokollnummer der im Paket transportierten Nutzlast. Dieser Wert befindet sich nun, bereit für die weitere Verarbeitung, im Akkumulator.

Als nächstes folgt eine Sprunganweisung. Die Semantik ist dabei folgende: Es wird der Wert im Akkumulator entweder mit einer Konstanten `BPF_K` oder dem Index Register `BPF_X` verglichen und anschliessend, falls das Resultat True oder Non-Null ist, entsprechend `jt` oder andernfalls entsprechend `jf` gesprungen. Diese Jump-Offsets werden dabei als Addition zum Program-Counter interpretiert. Im Beispiel wird also der Wert im Akkumulator (die Protokollnummer im IP-Header) mit der Konstante `IPPROTO_UDP` verglichen und falls diese gleich sind keine und ansonsten die nächsten 15 Zeilen übersprungen. Im Erfolgsfall fährt der Filter also gleich bei der nächsten Instruktion fort, andernfalls springen wir zur letzten Anweisung des Filters. Diese entstammt der Klasse `BPF_RET`, deren Instruktionen dazu dienen das Filterprogramm zu beenden. Dabei wird, entweder als Konstante oder über den Akkumulator (`BPF_A`), die Anzahl Bytes übergeben die der Filter vom Paket akzeptieren soll. Wird Null übergeben, wie in der letzten Anweisung des Filters in Listing 5.1, dann wird das Paket ignoriert. Der komplette Filter sollte sich mit Hilfe der Kommentare im Listing und obigen Erklärungen einigermassen nachvollziehen lassen.

Listing 5.1: Berkeley Packet Filter auf dem Raw-Socket

```
1 struct sock_filter ikev2_filter_code[] =
2 {
3     /* Protocol must be UDP */
4     BPF_STMT(BPF_LD+BPF_B+BPF_ABS, IP + 9),
5     BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, IPPROTO_UDP, 0, 15),
6     /* Destination Port must be either port or natt_port */
7     BPF_STMT(BPF_LD+BPF_H+BPF_ABS, UDP + 2),
8     BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, this->port, 1, 0),
```

```
9       BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, this->natt_port, 5, 12),
10      /* port */
11          /* IKE version must be 2.0 */
12          BPF_STMT(BPF_LD+BPF_B+BPF_ABS, IKE + 17),
13          BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x20, 0, 10),
14          /* packet length is length in
15           * IKEv2 header + ip header + udp header */
16          BPF_STMT(BPF_LD+BPF_W+BPF_ABS, IKE + 24),
17          BPF_STMT(BPF_ALU+BPF_ADD+BPF_K, IP_LEN + UDP_LEN),
18          BPF_STMT(BPF_RET+BPF_A, 0),
19      /* natt_port */
20          /* nat-t: check for marker */
21          BPF_STMT(BPF_LD+BPF_W+BPF_ABS, IKE),
22          BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0, 0, 5),
23          /* nat-t: IKE version must be 2.0 */
24          BPF_STMT(BPF_LD+BPF_B+BPF_ABS, IKE + MARKER_LEN + 17),
25          BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x20, 0, 3),
26          /* nat-t: packet length is length in
27           * IKEv2 header + ip header + udp header + non esp
               marker */
28          BPF_STMT(BPF_LD+BPF_W+BPF_ABS, IKE + MARKER_LEN + 24),
29          BPF_STMT(BPF_ALU+BPF_ADD+BPF_K, IP_LEN + UDP_LEN +
              MARKER_LEN),
30          BPF_STMT(BPF_RET+BPF_A, 0),
31      /* packet doesn't match, ignore */
32      BPF_STMT(BPF_RET+BPF_K, 0),
33  };
```

5.3.2. Kernel Interface

Das Kernel Interface von strongSwan II basiert, wie bereits in den Grundlagen (Kapitel 4.3) erwähnt, auf Netlink/XFRM. Im folgenden gehen wir auf unsere Erweiterungen an diesem Interface ein. So haben wir einerseits die in Abschnitt 4.3.3 beschriebenen Makros konsequent eingeführt, mit erheblichen Anpassungen am bestehenden Code. Auf diese Änderungen, so umfangreich sie auch sein mögen, gehen wir hier nicht noch einmal ein und verweisen auf den genannten Abschnitt. Andererseits haben wir auch neue Funktionalität hinzugefügt, die wir in den folgenden Unterabschnitten vorstellen möchten.

Änderungen an der Funktion `add_sa`

Die Funktion `add_sa` des Kernel Interfaces dient dazu, eine IPsec-SA im Kernel zu installieren. Dies kann grundsätzlich auf zwei Arten geschehen. So können mit einer Nachricht vom Typ

XFRM_MSG_NEWSA gleich alle Informationen einer SA an den Kernel geschickt werden oder aber es wird mit einer XFRM_MSG_UPDSA-Nachricht eine SA vervollständigt, die zuvor mit dem Bezug eines SPI über die Methode get_spi implizit erstellt wurde. Im Falle von strongSwan wird jeweils für die remote SA die erste und für die lokale SA die zweite Methode verwendet.

Die Hauptaufgabe der Methode liegt nun darin, alle Daten wie Protokoll, Modus, Timeouts und natürlich die Schlüssel in eine Netlink Nachricht zu verpacken und diese an den Kernel zu senden. Im Abschnitt 4.3.3 haben wir ja bereits beschrieben, wie dieser Packetaufbau abläuft. Analog zur vorgestellten Verschlüsselungsalgorithmus-Payload gehen wir vor, um im Kernel die UDP Encapsulation (siehe Abschnitt 4.2.4) zu aktivieren. Der Methode add_sa übergeben wir hierzu einen Pointer auf ein Struct, welches zwei UDP-Ports beinhaltet. Diese Ports spezifizieren die zwei Endpunkte der Verbindung welche benötigt werden. Um die UDP-Encapsulation zu deaktivieren, wird der Funktion ein NULL-Pointer übergeben. Das Listing 5.2 ist als Fortsetzung des Beispiels in Abschnitt 4.3.3 anzusehen.

Listing 5.2: Aktivierung der UDP Encapsulation im Kernel

```
1  if (natt)
2  {
3    rthdr->rta_type = XFRMA_ENCAP;
4    rthdr->rta_len = RTA_LENGTH(sizeof(struct xfrm_encap_tmpl));
5    hdr->nlmsg_len += rthdr->rta_len;
6
7    struct xfrm_encap_tmpl* encap = (struct xfrm_encap_tmpl*)
       RTA_DATA(rthdr);
8    encap->encap_type = UDP_ENCAP_ESPINUDP;
9    encap->encap_sport = ntohs(natt->sport);
10   encap->encap_dport = ntohs(natt->dport);
11   memset(&encap->encap_oa, 0, sizeof (xfrm_address_t));
12
13   rthdr = XFRM_RTA_NEXT(rthdr);
14 }
```

Wie im Listing ersichtlich prüfen wir zuerst ob NAT-T eingesetzt wird oder nicht. Im positiven Fall setzen wir dann Typ und Länge der aktuellen Teilpayload (ausserdem aktualisieren wir die Länge der gesamten Netlink-Nachricht). Die vom Nachrichtentyp XFRMA_ENCAP verlangten Daten vom Typ structxfrm_encap_tmpl werden als nächstes an die Nachricht angehängt. Anschliessend wird der Pointer rthdr auf die nächste Teilpayload verschoben und damit dieser Teilblock der Methode abgeschlossen.

Die Members des Structs xfrm_encap_tmpl sind die folgenden:

encap_type Dieser Member bezeichnet den Typ der UDP-Enkapsulierung. Da wie in Abschnitt

4.2.4 erläutert, erste Drafts vorschlugen, statt des heute üblichen NON-ESP-Markers einen NON-IKE-Marker einzuführen, definiert `linux/include/linux/udp.h` neben dem von uns verwendeten `UDP_ENCAP_ESPINUDP` auch `UDP_ENCAP_ESPINUDP_NON_IKE` als möglichen Wert.

encap_sport, encap_dport Hierin landen die UDP-Ports der Endpunkte des IPsec-Tunnels. Die eingekapselten ESP-Pakete werden dann mit diesen Ports addressiert.

encap_oa *oa* steht in diesem Fall für *Other Address* und bezeichnet jene IP-Adresse, welche die Gegenstelle im inneren des NAT hat. Die Idee dahinter ist, dass damit im Transport Mode die - mit Hilfe des *Pseudo-Headers* berechnete - Checksumme im Header von transportierten TCP/UDP-Paketen wieder korrigiert werden könnte, nachdem Teile des IP-Headers durch das NAT verändert worden sind (siehe [HSV+05, S. 4]). In Ermangelung einer separaten Payload (wie sie in IKE existierte), wird in [Kau05, S. 39] vorgeschlagen, dass die dazu benötigte IP-Adresse in IKEv2 aus dem Traffic Selector extrahiert werden soll. Wie oben im Listing aber ersichtlich ist, setzen wir den Wert dieses Members bedingungslos auf 0. Der Grund dafür liegt im Linux Kernel. Denn der Member wird in der aktuellen Kernelimplementation ohnehin nicht verwendet, dies ist auch zulässig, insofern der Linux Kernel die Checksumme von authentisierten ESP-Paketen schlichtweg ignoriert. Dieser Member ist also vorläufig nicht von Bedeutung.

Die Funktion `update_sa_hosts`

Da aufgrund eines NAT sowohl die Adresse als auch der Port eines Tunnel-Endpunktes ändern kann, werden diese bei Erhatl einer IKE-Nachricht jeweils geprüft und falls nötig aktualisiert (siehe auch 5.3.4). Die Methode `update_sa_hosts` propagiert diese Änderungen an eine installierte SA im Kernel weiter. Wie weiter oben bereits erwähnt, ist es auch möglich mittels der Methode `add_sa` eine bestehende SA zu aktualisieren. Dazu wäre es aber erforderlich, alle Daten der SA, wie beispielsweise die verwendeten Schlüssel, irgendwo zu cachen. Dies war zum Zeitpunkt der Implementation nicht gegeben und ist wie wir sehen auch gar nicht nötig.

Da wir also die zur direkten Aktualisierung benötigten Daten nicht zur Verfügung haben, laden wir uns in einem ersten Schritt die installierte SA aus der SADB des Kernels. Dazu verwenden wir eine Nachricht vom Typ `XFRM_MSG_GETSA`, die zur Identifikation der SA ein Struct vom Typ `xfrm_usersa_id` verlangt. Die Informationen die der Kernel dabei als Primärschlüssel verwendet sind folgende:

1. Der SPI der SA.

2. Das verwendete Protokoll (AH oder ESP).

3. Die IP-Adresse (und Adressfamilie) des Remoteendes des IPsec-Tunnels.

Auf diese Liste werden wir in Kürze noch einmal zurück kommen. Wie in der Einführung über Netlink gezeigt (Listing 4.4) wird üblicherweise auf versendete Nachrichten, mit Hilfe des Flags NLM_F_ACK, eine Bestätigung des Kernel verlangt. In diesem Fall ist es jedoch wichtig, dass wir dieses Flag nicht setzen, da wir nun ja anstelle einer Bestätigung Daten (die SA) als Antwort erwarten. Diese Daten werden uns in Form einer XFRM_MSG_NEWSA-Nachricht geliefert - also in genau derselben Form in der wir in add_sa auch eine SA im Kernel installieren würden.

Listing 5.3: Aktualisierung einer SA

```
1 update->nlmsg_flags = NLM_F_REQUEST | NLM_F_ACK;
2 update->nlmsg_type = XFRM_MSG_UPDSA;
3
4 struct xfrm_usersa_info *sa = (struct xfrm_usersa_info*)
    NLMSG_DATA(update);
5 if (src_changes & HOST_DIFF_ADDR) {
6     sa->saddr = new_src->get_xfrm_addr(new_src);
7 }
8 if (dst_changes & HOST_DIFF_ADDR) {
9     update->nlmsg_type = XFRM_MSG_NEWSA;
10    sa->id.daddr = new_dst->get_xfrm_addr(new_dst);
11 }
```

Dieser Fakt erlaubt uns, die empfangene Nachricht unseren Wünschen entsprechend anzupassen und dann wieder zurück an den Kernel zu schicken - Daten die uns nicht “interessieren” belassen wir einfach so wie sie sind. Listing 5.3 zeigt die durchgeführten Modifikationen.

Kommen wir nun noch einmal auf obige Liste zurück. Wie in Punkt drei beschrieben, verwendet der Kernel unter anderem die IP-Adresse der Destination zur Identifikation einer SA. Falls sich diese nun geändert hat, ist es nicht mehr möglich, die SA direkt zu aktualisieren. Würden wir die Destinations Adresse analog zur Source Adresse einfach abändern, wüsste der Kernel nicht mehr, welche SA er eigentlich aktualisieren soll und würde das Update mit einer Fehlermeldung quittieren.

Stattdessen verwenden wir die gleiche Nachricht, um eine neue SA zu erstellen. Dazu setzen wir den Messagetyp von XFRM_MSG_UPDSA auf XFRM_MSG_NEWSA. Anschliessend können wir die Adresse bedenkenlos ersetzen. Nachdem uns der Kernel die Installation der neuen SA dann bestätigt hat, löschen wir die alte SA über die Funktion del_sa.

Falls sich der Port eines der beiden Endpunkte geändert haben sollte, müssen wir im Falle von NAT-T auch die Konfiguration der UDP-Encapsulation modifizieren. Dies geschieht wie in Listing 5.4 gezeigt durch Iteration über die angehängten Teilpayloads. Dazu verwenden wir die zwei Makros RTA_OK und RTA_NEXT welche uns den Pointer rthdr jeweils auf die nächste gültige Teilpayload ausrichten. Falls die Teilpayload dem gewünschten Typ, XFRMA_ENCAP, entspricht

Listing 5.4: Aktualisierung einer SA (Ports)

```
struct rtattr *rthdr =XFRM_RTA(update,struct xfrm_usersa_info);
size_t rtsize = XFRM_PAYLOAD(update, struct xfrm_usersa_info);
while (RTA_OK(rthdr, rtsize))
{
  if (rthdr->rta_type == XFRMA_ENCAP)
  {
    struct xfrm_encap_tmpl* encap = (struct xfrm_encap_tmpl*)
       RTA_DATA(rthdr);
    encap->encap_sport = ntohs(new_src->get_port(new_src));
    encap->encap_dport = ntohs(new_dst->get_port(new_dst));
    break;
  }
  rthdr = RTA_NEXT(rthdr, rtsize);
}
```

setzen wir die Ports und springen aus der Schleife. Anschliessend wir die modifizierte Nachricht an den Kernel gesendet.

Die Funktion `query_policy`

Aus der Entwicklung der DPD und der Keepalives entstand das Bedürfnis, zu erfahren, wann der letzte Traffic über eine bestimmte IPsec-Policy gelaufen ist. Zu diesem Zweck implementierten wir die Methode `query_policy`, welche uns in der aktuellen Implementation genau diese Information liefert. Die Methode ist aber leicht erweiterbar, um auch andere Daten bezüglich einer Policy abzufragen (zum Beispiel die Anzahl übertragener Pakete).

Vom Prinzip her verwenden wir das gleiche Vorgehen wie im ersten Teil der Funktion `update_sa_hosts`. Anstatt der SA laden wir hier aber eine spezifische Policy aus dem Kernel. Identifiziert wird eine Policy über das Struct `xfrm_userpolicy_id`. Dieses spezifiziert den Netzwerkverkehr, der durch diese Policy gematcht wird, also Informationen wie Netzwerkadresse/Netzmaske, Portrange, Higher-Level-Protokoll sowie die Richtung (IN, OUT, FWD).

Das diese Informationen nicht die eigentlichen Adressen der Endpunkte beinhalten – diese werden einer Policy mittels eines `structxfrm_user_tmpl` übergeben – können wir übrigens die Policies bei Adressupdates direkt über die Methode `add_policy` aktualisieren.

Analog zur SA antwortet uns der Kernel mit einer `XFRM_MSG_NEWPOLICY`-Nachricht, welche neben den kompletten Daten der Policy, die über die Methode `add_policy` an den Kernel geschickt wurden, auch dynamische Daten welche wir wie folgt auslesen.

```
1 struct xfrm_userpolicy_info *policy = (struct
    xfrm_userpolicy_info*)NLMSG_DATA(response);
2 *use_time = (time_t)policy->curlft.use_time;
```

Der Member `curlft` vom Typ `structxfrm_lifetime_cur` liefert neben dem hier ausgelesenen Zeitpunkt der letzten Benutzung auch die Anzahl übertragener Bytes (`bytes`) und Pakete (`packets`) sowie den Zeitpunkt der Installation im Kernel (`add_time`) – alle vom Typ `__u64`.

5.3.3. NAT Detection (NAT-D)

Unsere NAT Detection besteht aus vier verschiedenen Teilen, die wir im Folgenden näher erläutern werden:

- Die Methode `generate_natd_hash`, welche den Hash berechnet,
- Code in den Zuständen `initiator_init`, `responder_init` sowie `ike_sa_init_requested`, welcher die Payloads generiert respektive analysiert, und
- den Daten-Members der IKE-SA, welche den NAT-Status mitführen.

Die Methode `generate_natd_hash` auf der IKE-SA berechnet zu zwei SPIs und einem `host_t` den dazugehörigen NAT-D-Hash. Die relevanten Zeilen des Quellcodes sind in Listing 5.3.3 dargestellt. Gemäss den Anforderungen des Standards werden die SPIs, die IP-Adresse und der UDP-Port in binärer Form, Network Byte Order, konkateniert, und der Hashwert dieses Strings berechnet.

Die vorliegende Variante ist IPv4-spezifisch implementiert. Die Methode muss für IPv6-Unterstützung leicht erweitert werden. Je nachdem wie der IPv6-Support von `host_t` aussehen wird, könnten Teile dieser Methode auch in `host_t` ausgelagert werden, damit die Details des Adressierungsschemas für die Methode transparent werden.

```
1 natd_hash = chunk_alloc(this->nat_hasher->get_hash_size(this->
    nat_hasher));
2 natd_string = chunk_alloc(8 + 8 + 4 + 2);
3
4 sai = (struct sockaddr_in*)host->get_sockaddr(host);
5 p = natd_string.ptr;
6 *(u_int64_t*)p = spi_i;               p += sizeof(spi_i);
7 *(u_int64_t*)p = spi_r;               p += sizeof(spi_r);
8 *(u_int32_t*)p = sai->sin_addr.s_addr; p += sizeof(sai->
    sin_addr.s_addr);
9 *(u_int16_t*)p = sai->sin_port;       p += sizeof(sai->
    sin_port);
```

```
10
11 this->nat_hasher->get_hash(this->nat_hasher, natd_string,
     natd_hash.ptr);
12 this->nat_hasher->reset(this->nat_hasher);
```

Auf Seiten des Initiators werden im Zustand `initiator_init` mit Hilfe der Methode `generate_natd_hash` der IKE-SA die NAT-Detection-Payloads generiert und dem `IKE_SA_INIT`-Request angefügt. Im Zustand `ike_sa_init_requested` werden die vom Responder erhaltenen NAT-D Payloads schliesslich analysiert, und falls aufgrund der erhaltenen Payloads ein NAT detektiert wird, der IKE-SA mitgeteilt, welche Seite(n) sich hinter einem NAT befinden. Falls keine NAT-D-Payloads von der Gegenseite empfangen werden, bedeutet dies, dass die Gegenseite NAT-T nicht unterstützt. In diesem Falle wird NAT-T deaktiviert. Details zum Inhalt der NAT-D-Payloads haben wir in Abschnitt 4.2.4 dokumentiert.

Auf Seiten des Responders geschehen beide Tätigkeiten im Zustand `responder_init`, unterscheiden sich ansonsten aber nicht wesentlich von denjenigen des Initiators.

Die IKE-SA speichert den NAT-Zustand sowohl des lokalen Hosts als auch denjenigen des Gegenübers. Es kann von aussen abgefragt werden, ob und welche Hosts dieser IKE-Verbindung hinter NAT stecken.

5.3.4. Adress-Update und Port-Agilität

Eine wichtige Anforderung an eine mit NAT-Traversal ausgestattete Implementierung von IKEv2 ist die Fähigkeit, die Adresse und den Port des Peers im laufenden Betrieb aktualisieren zu können, wenn sie ändert. In jedem Zustand der IKE-SA rufen wir zu diesem Zweck die neue Methode `update_connection_hosts` auf der IKE-SA mit den Adressen und Ports des empfangenen Pakets auf. Dies darf aber erst geschehen, nachdem das eingehende Paket auf Authentizität und Integrität geprüft wurde.

Die Methode `update_connection_hosts`

In der Methode `update_connection_hosts` haben wir den kompletten Adress-Update-Algorithmus gekapselt. Als erstes prüft die Funktion die übergebenen neuen Adressen und Ports gegen diejenigen, welche in der Connection auf der IKE-SA gespeichert sind. Falls sich Adressen und/oder Ports geändert haben, wird die entsprechende Update-Methode auf der Connection aufgerufen, um in der Connection die neue Adress/Port-Kombination zu speichern.

Im speziellen Fall, wo wir uns selber hinter einem NAT befinden, darf nur der Port des Gegenübers, nicht aber die Adresse aktualisiert werden, um der in [Kau05] erwähnten DoS-Attacke vorzubeugen. In diesem Fall wird eine Warnung ins Log geschrieben und das Update verweigert.

Sofern ein Update stattgefunden hat, aktualisieren wir anschliessend die Adressen auf allen Child-SAs, indem wir dort die Methode `update_hosts` aufrufen. In der Child-SA wird das

Adress-Update schliesslich an das Kernel-Interface weitergegeben, um im Falle von relevanten Änderungen die IP-Adressen und Ports in den Security Associations zu aktualisieren, oder bei Adressänderungen auch die Policies. Wie dieses Update auf Kernel-Ebene implementiert ist, haben wir im Abschnitt 5.3.2 beschrieben.

5.3.5. NAT Keepalives

Sobald detektiert wurde, dass wir uns hinter einem NAT befinden, schicken wir periodisch NAT Keepalives. Unsere Implementation besteht nur aus der Job-Klasse `send_keepalive_job_t`, welche in der `execute`-Methode prüft, ob die IKE-SA in ausgehender Richtung lange genug idle war, und falls ja, direkt ein NAT-Keepalive-Paket generiert und in die Send Queue einfügt. Der Job fügt sich anschliessend selber automatisch wieder in die Event Queue ein, da keine Antwort abgewartet werden muss. Ab dem Zeitpunkt wo im State `responder_init` respektive `ike_sa_init_requested` ein NAT detektiert wurde, ist also immer ein `send_keepalive_job_t` in der Event Queue.

Die Information, wie lange die IKE-SA idle war, wird vom Kernel Interface wie in Abschnitt 5.3.2 erläutert abgefragt, und bezieht sich sowohl auf (enkapsulierte) IKE-Messages, welche von `charon` gesendet wurden, als auch auf (enkapsulierte) ESP-Pakete, welche vom Kernel gesendet worden sind. Beides reicht aus, um die NAT-Mappings in den NAT-Routern aktiv zu halten.

5.3.6. Dead Peer Detection (DPD)

Um Dead Peer Detection zu realisieren, haben wir beschlossen, jeden IKEv2-Exchange als für DPD relevant zu behandeln. Wird also ein Request irgendeines Exchanges auch nach mehreren Retransmits nicht von der Gegenseite quittiert, so wird angenommen, dass der Peer nicht mehr aktiv ist, und wenn DPD aktiviert ist, entsprechend gehandelt. Es macht in unseren Augen keinen Sinn, DPD-Pakete (also `INFORMATIONAL`-Exchanges ohne Payloads) gesondert zu behandeln, einerseits weil auf Protokollebene kein spezieller Payload-Typ existiert, andererseits weil jeder IKEv2-Exchange die Aktivität des Gegenübers bestätigt. Schliesslich weiss man ja bereits nach Nicht-Erhalt der Response eines normalen Requests, dass das Gegenüber nicht mehr aktiv ist. Nach unserer Interpretation des Standards [Kau05] ist dies auch so korrekt und "im Sinne des Erfinders".

Wir haben also DPD im Rahmen der bereits existierenden Retransmission-Funktionalität implementiert. Die Default-Timings von strongSwan II genügten aber den Anforderungen nicht, da erstens der Retransmission-Mechanismus nie abgebrochen hat, und zweitens die Timings exponentiell grösser wurden, bis schliesslich ein Integer Overflow stattfand. Der exponentielle Backoff-Algorithmus berechnet die Wartezeit t_d für die n-te Retransmission aus der Formel:

$$t_d(n) = \text{Multiplikator} \times \text{Basis}^n$$

Für die Lösung, welche unseren Ansprüchen an die Reaktionszeit genügt, gleichzeitig aber auch den exponentiellen Backoff-Algorithmus sinnvoll beibehält, wählten wir eine Basis von 1.5 und einen Multiplikator von 6000 Millisekunden. Nach 6 Retransmits wird abgebrochen. Dies resultiert in einer Reaktionszeit von etwas über 3 Minuten, was unseren Bedürfnissen gerecht wird. Wir haben die Timings der ersten 20 Retransmissions mit den alten und den neuen Default-Werten in Tabelle 5.1 dargestellt.

Retransmission n	Delay alt $3000 \cdot 2^n$ [ms]	Delay neu $6000 \cdot 1.5^n$ [ms]	Delay neu [s]	$\sum$ neu [min]
0	3000	6000	6.0	0.10
1	6000	9000	9.0	0.25
2	12000	13500	13.5	0.48
3	24000	20250	20.3	0.81
4	48000	30375	30.4	1.32
5	96000	45563	45.6	2.08
6	192000	68344	68.3	3.22
7	384000	102516	102.5	4.93
8	768000	153773	153.8	7.49
9	1536000	230660	230.7	11.33
10	3072000	345990	346.0	17.10
11	6144000	518985	519.0	25.75
12	12288000	778478	778.5	38.72
13	24576000	1167717	1167.7	58.19
14	49152000	1751575	1751.6	87.38
15	98304000	2627363	2627.4	131.17
16	196608000	3941045	3941.0	196.85
17	393216000	5911567	5911.6	295.38
18	786432000	8867351	8867.4	443.17
19	1572864000	13301026	13301.0	664.85
20	3145728000	19951540	19951.5	997.38

Tabelle 5.1.: Retransmission Timings

Unsere Implementation von Dead Peer Detection umfasst:

- die Methode `send_dpd_request` auf der IKE-SA, welche einen leeren `INFORMATIONAL`-Request als DPD-Request losschickt;
- die Job-Klasse `send_dpd_job_t`, welche in der `execute`-Methode prüft, ob die IKE-SA in eingehender Richtung lange genug idle war, und falls ja, `send_dpd_request` auf der IKE-SA aufruft, und sich sonst selber wieder in die Event Queue einfügt;

- der Methode schedule_dpd_job auf dem State ike_sa_established, welche send_dpd_job_t instanziert und der Event Queue hinzufügt, sofern DPD aktiviert ist.

Wenn der State ike_sa_established erstmals betreten wird, wird schedule_dpd_job aufgerufen. Sobald eine DPD-Response empfangen wurde, wird wieder schedule_dpd_job aufgerufen, um nach dem konfigurierten Intervall wieder ein DPD zu schicken. Es ist also immer entweder ein DPD-Exchange unterwegs, oder ein send_dpd_job_t in der Event Queue.

Die Information, wie lange die IKE-SA idle war, wird vom Kernel Interface wie in Abschnitt 5.3.2 erläutert abgefragt, und bezieht sich sowohl auf IKE-Messages, welche charon empfangen hat, als auch auf ESP-Pakete, welche vom Kernel empfangen worden sind. Beides signalisiert uns, dass die Gegenstelle noch aktiv sein muss.

5.4. Sonstige Resultate

Neben unserer Arbeit an NAT-Traversal und Dead Peer Detection haben wir auch hier und dort Änderungen oder Verbesserungen an charon angebracht. Nennenswert sind:

- UML Test Framework um die Möglichkeit erweitert, mit tcpdump Pakete zu zählen.
- UML Test Framework um die Möglichkeit erweitert, generische Befehle auszuführen und den Rückgabewert zu prüfen.

5.5. Angeregte Verbesserungen

Über unser Ticketing-System haben wir dem Lead Developer von strongSwan II, Martin Willi (mwilli) diverse Probleme und Anregungen gemeldet, welche im Laufe unserer Arbeit an charon entstanden sind. Diese Tickets haben wir in Tabelle 5.2 zusammengefasst. Einige Punkte, welche uns wichtig oder interessant erscheinen, erläutern wir ausführlicher.

5.5.1. Integer Overflows in event_queue

Bei unserer Arbeit haben wir mehrere Integer-Overflows in der Event-Queue entdeckt. Die problematische Version von time_difference ist in Listing 5.5.1 zu sehen:

```
1 static long time_difference(struct timeval *end_time, struct
    timeval *start_time)
2 {
3     long seconds, microseconds;
4
5     seconds = (end_time->tv_sec - start_time->tv_sec);
6     microseconds = (end_time->tv_usec - start_time->tv_usec);
```

```
7     return ((seconds * 1000000) + microseconds);
8 }
```

Hier wird eine Zeitdifferenz in Mikrosekunden in einem `long` zurückgegeben. Da ein `long` auf 32-Bit-Architekturen 32 Bit fasst, und ein Bit für das Vorzeichen benötigt wird, überläuft der Rückgabewert bei Zeitdifferenzen über ca. 36 Minuten:

$$\frac{2^{32-1}}{10^6}\,\mathrm{s} = \frac{2^{32-1}}{60 * 10^6}\,\mathrm{min} \approx 35.8\,\mathrm{min}$$

Ein zweiter Integer-Overflow findet sich in folgendem Code-Snippet:

```
1 static void add_relative(event_queue_t *this, job_t *job,
     u_int32_t ms)
2 {
3     timeval_t current_time;
4     timeval_t time;
5     int micros = ms * 1000;
6
7     gettimeofday(&current_time, NULL);
8
9     time.tv_usec = ((current_time.tv_usec + micros) % 1000000);
10    time.tv_sec = current_time.tv_sec + ((current_time.tv_usec
         + micros)/ 1000000);
11
12    this->add_absolute(this, job, time);
13 }
```

Wir sehen schön, dass ein `u_int32_t` (Millisekunden) mit 1000 multipliziert wird, und das Resultat in einem `int` gespeichert wird. Wieder werden also Mikrosekunden in einem 32 Bit breiten, vorzeichenbehafteten Integer gespeichert, was wie vorhin berechnet zu einem Overflow ab ca. 36 Minuten führt.

Diese Probleme wurden behoben, und es wird jetzt mit den dafür vorgesehenen Datentypen gerechnet: `time_t` und `suseconds_t`

5.5.2. SPI zufällig erzeugen

Die Security Parameter Indizes der IKE-Verbindung wurden von `charon` nicht zufällig erzeugt, sondern fortlaufend nummeriert. Dadurch wurde die Eindeutigkeit der SPI nur für die Dauer der Laufzeit des Daemons garantiert; sobald der Daemon neu gestartet wird, wiederholten sich die gleichen SPIs. Ausserdem ist es nicht erwünscht, dass die SPIs vorhersehbar sind. Es entsteht dadurch zwar nicht direkt ein Sicherheitsproblem, es könnte aber einem Angreifer trotzdem einen Vorteil bescheren. Aus diesen Gründen wurde schliesslich auf unsere Anregung hin beschlossen, die SPI auch in `charon` zufällig zu generieren.

5.5.3. OOD: Job-Logik in Jobs statt Thread Pool

Die Programmlogik der Jobs in der Job Queue, welche von den Threads im Thread Pool abgearbeitet werden, war in den Methoden des Thread Pools implementiert. Ein neuer Job-Typ fügte man also hinzu, indem man einerseits die neue Job-Klasse erzeugte, eine neue Konstante für den Job-Typ definierte, und die Logik zur Abarbeitung dieses Job-Typs zum Thread Pool hinzufügte.

Aus der Sichtweise des objektorientierten Designs ist dies nicht wünschenswert. Viel schöner wäre die Kapselung, wenn die Jobs über eine `execute`-Methode verfügten, und somit die komplette Logik des Jobs in der Job-Klasse selber implementiert wäre. Der Thread Pool würde so zu dem, was er eigentlich sein sollte: eine generische Implementation eines Thread Pools. Diese Anregung wurde so umgesetzt.

Ticket	Subject	Resolution
#55	Memory Leak in `child_sa::install`	`invalid`
#54	Timeout in `kernel_interface::send_message`	`open`
#48	`strftime` in `timetoa` nutzen	`open`
#46	Retransmission auf ein Paket beschränkt	`wontfix`
#45	Charon Unit Tests nicht in Build	`fixed`
#44	OOD: Job-Logik in Jobs statt Thread Pool	`fixed`
#43	Integer Overflows in `event_queue`	`fixed`
#39	Duplicate Packet Detection fehlt	`wontfix`
#35	SPI nicht zufällig erzeugt	`fixed`
#33	Hasher: Interface irreführend	`fixed`
#32	Design von `chunk_from_buf`	`wontfix`
#31	`SIGSEGV` wenn Stroke-Socket open failed	`invalid`
#30	`SIGSEGV` bei bereits existierenden Policies	`invalid`
#28	make test fails	`worksforme`
#24	Makefiles überarbeiten	`fixed`

Tabelle 5.2.: Zusammenfassung unserer Tickets an `mwilli`

6. Tests

6.1. Methodik

Tests haben wir sowohl über ein reales Netz zwischen unseren Entwicklungssystemen durchgeführt, als auch als automatisierte Tests in die bestehende UML-Testumgebung integriert.

6.1.1. Reale Tests

Abbidlung 6.1 zeigt die von uns verwendete Netzwerktopologie. Die umrahmte Teilkonfiguration hatten wir dabei stets aufgebaut um während der Entwicklung jeweils möglichst schnell die Funktionsweise zu überprüfen. Der Cisco 2611 Router zwischen den beiden PCs wurde so konfiguriert, dass er Zugriffe aus dem "privaten" Netz 192.168.0.0/24 ins "öffentliche" Netz 10.0.0.0/24 auf seine öffentliche Adresse 10.0.0.1 nattet (NAT Overload, in Cisco-Terminologie). Diese Konfiguration erlaubte uns jederzeit eine **Host to Host** Verbindung aufzubauen und zu testen. Die erweiterte Netztopologie, mit zwei zusätzlichen Laptops (einer unter FreeBSD einer unter Windows), verwendeten wir um auch eine **Net to Net** Konfiguration in einer realen Umgebung zu testen. In beiden Testkonfigurationen wurde der IPsec-Tunnel jeweils zwischen den beiden PCs aufgebaut.

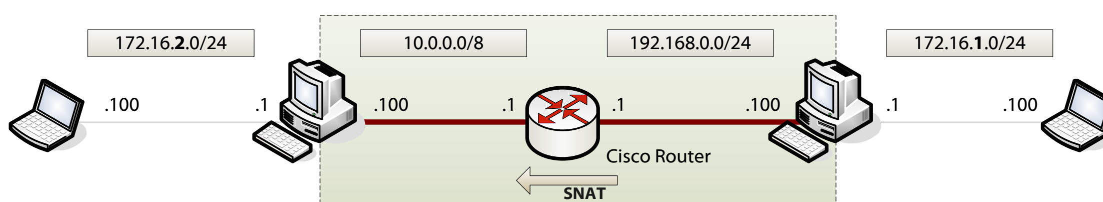


Abbildung 6.1.: "Realer" Testaufbau

6.1.2. UML Tests

Um auch ausgefallenere Konfigurationen testen zu können, ohne jedesmal die Verkabelung und die Konfiguration unsere Systeme und der des Cisco Routers ändern zu müssen, verwendeten wir die für strongSwan entwickelte UML-Testumgebung. Die automatisierten Tests lassen sich auch

jederzeit unter gleichen Bedingungen wiederholen. Durch die Integration in die bisherige Testsuite ist auch gewährleistet, dass durch zukünftige Arbeit am Projekt die entwickelte Funtionalität zerbricht. Die Netzwerktopologie der UML-Testumgebung ist in Abbildung 6.2 abgebildet. Sie besteht im Prinzip aus drei Netzen wobei das mittlere Netz (192.168.0.0/24) in den meisten Tests die Rolle des "öffentlichen" Netzes übernimmt. Aufgrund der starren Netzwerktopologie konnten wir gewisse NAT-Konfigurationen teilweise nicht befriedigend testen (siehe auch Kapitel 7.1).

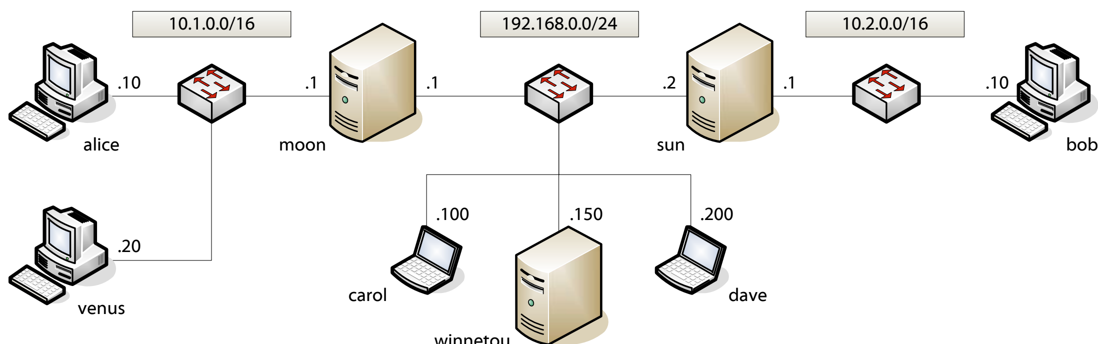


Abbildung 6.2.: UML-Testumgebung

Während dem Testen in der UML-Umgebung stiessen wir auf ein interessantes Problem, welches im Zusammenhang mit der *Connection Tracking State Table* der Linux Firewall `netfilter` zu Tage trat. Als *stateful*-Firewall verwendet `netfilter` diese *conntrack*-Tabelle um zu jeder Verbindung Informationen zu speichern, dies sind beispielsweise die IP-Adressen und Ports der Endpunkte, der Zustand und das verwendete Protokoll der Verbindung sowie Timeouts. Stateful Firewalls sind inhärent sicherer als ihre zustandslosen Gegenstücke, da sie etwa eingehende Pakete nur als Antwort auf ausgehende Pakete zulassen können.

Wird in einem Test beispielsweise von Host **carol** ein IPsec-Tunnel zu Gateway **moon** aufgebaut, dann werden die Daten der zugehörigen IKE/UDP-Verbindung wie folgt in der *conntrack*-Tabelle auf **moon** aufgezeichnet (leicht umformatiert):

```
udp   17 56 src=192.168.0.100 dst=192.168.0.1 sport=500 dport=500
            src=192.168.0.1 dst=192.168.0.100 sport=500 dport=500
            [ASSURED] mark=0 use=1
```

Diese Einträge bleiben je nach Zustand und Art der Verbindung unterschiedlich lange in der Tabelle erhalten. Für bestätigte UDP-Connections ist der Timeout in `/usr/src/linux/net/ipv4/netfilter/ip_conntrack_proto_udp.c` standardmässig auf 180 Sekunden festgelegt.

Ein nachfolgender Test der Testsuite möchte nun auf **moon** Port-Forwarding (siehe Abschnitt 4.1) simulieren. Dazu installiert er eine Firewall-Regel, welche Anfragen auf Port 500 und 4500 aus dem Netz 192.168.0.0/24 auf den Host **alice** weiterleitet. Ein Verbindungsaufbau von **carol**

auf **moon** (**carol** weiss nicht, dass **moon** nicht selbst der IPsec-Gateway ist) verleitet nun **moon**, aufgrund des bestehenden Eintrags in der *conntrack*-Tabelle, zum Trugschluss, dass diese Verbindung bei ihm selbst terminiert. Dies lässt den Test in der Folge scheitern. In Abschnitt 7.1 schlagen wir eine Lösung für dieses Problem vor.

6.2. Testfälle

Im Folgenden möchten wir einige auftretende NAT-Situationen vorstellen und jeweils sowohl die real durchgeführten als auch die zur UML-Testsuite hinzugefügten Testfälle beschreiben. Die einzelnen NAT-Konfigurationen sind in Abschnitt 4.1 beschrieben.

6.2.1. SNAT

Die SNAT-Situation ist wohl am häufigsten im realen Einsatz anzutreffen. Unzählige Heim- und Firmennetzwerke werden mit einfachen ADSL-Routern ans Internet angeschlossen wobei die meisten dieser Router auf die eine oder andere Art SNAT unterstützen.

Reale Tests

Wie bereits in der Einführung dieses Kapitels beschrieben, basierte unser realer Netzaufbau auf einem SNAT zwischen unseren beiden Arbeitsplätzen. In dieser Konfiguration haben wir sowohl *Host to Host*-Verbindungen als auch, unter Einbeziehung unserer Laptops, eine *Net to Net*-Verbindung getestet.

Den *Host to Host*-Tunnel zwischen unseren Systemen verwendeten wir hauptsächlich, um während der Entwicklung möglichst rasch ein Feedback zu erhalten. Somit ist dies die wohl am ausgiebigsten getestete Konfiguration.

Zu zwei Gelegenheiten konfigurierten wir jeweils das zweite Interface unserer Arbeitsstationen, das ansonsten ans HSR-Netz angeschlossen war, so dass wir auch eine *Net to Net*-Verbindung testen konnten. Auch in dieser Konfiguration haben wir den IPsec-Tunnel zwischen den beiden PCs aufgebaut, wobei jeder jeweils “sein” 172.16.x.0/24-Netz anbot.

UML Tests

nat-rw-one Dieser UML-Test konfiguriert auf **moon** ein SNAT welches das Netz 10.1.0.10/16 ins “öffentliche” Netz 192.168.0.0/24 nattet (siehe Abbildung 6.3). Anschliessend baut Roadwarrior **alice** einen IPsec-Tunnel zum Gateway **sun** auf. **alice** pingt danach den Rechner **bob**, wobei diese ICMP-Pakete als UDP-encapsulated ESP auf Moon aufgezeichnet werden.

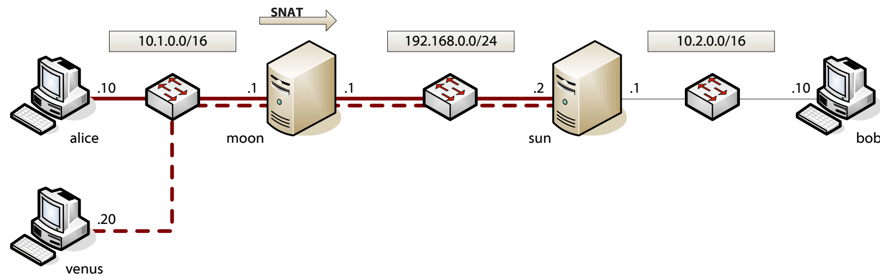


Abbildung 6.3.: UML-Tests zu SNAT

nat-rw-two Als Erweiterung des obigen Tests baut hier neben **alice** zusätzlich auch **venus** einen Tunnel zu **sun** auf.

nat-rw-mixed Dies ist eine weitere Variante des obigen Tests. In diesem Fall verwendet aber **venus** nicht `charon` um die Verbindung aufzubauen, sondern baut die Verbindung mit `pluto` auf.

6.2.2. Double SNAT

In dieser Konfiguration befindet sich der Initiator hinter zwei NAT-Router. Dies lässt sich etwa mit folgender Situation veranschaulichen: Privates Netzwerk mit ADSL-Router und einem Wireless LAN mit separatem Subnetz innerhalb des LANs, wobei dieses vom WLAN-Accesspoint ins LAN genattet wird. Diese Situation haben wir mangels Hardware ausschliesslich als UML-Test getestet.

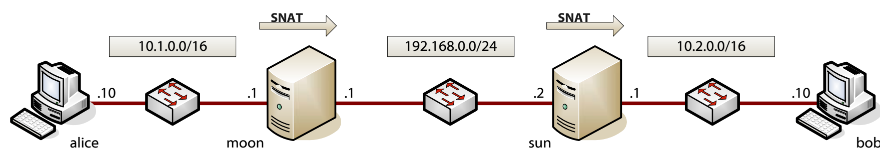


Abbildung 6.4.: UML Test zu Double SNAT

ikev2-nat-double-snat Abbildung 6.4 zeigt wie dieser Test innerhalb der Testsuite abgebildet wurde. Die beiden Hosts **moon** und **sun** sind so konfiguriert, dass sie jeweils das ihnen zur Linken liegende Netz ins ihnen zur Rechten liegenden Netz natten. Anschliessend baut **alice** eine Verbindung zum Gateway **bob** auf und pingt diesen an. Anschliessend wird auf **moon** geprüft, ob die Pings auch in UDP eingekapselt übertragen werden.

6.2.3. DNAT

Diese Situation liegt dann vor, wenn sich ein IPsec-Gateway hinter einem NAT-Router befindet. Dies trifft man beispielsweise bei kleineren Privat- und Firmentnetzen an, die über einen ADSL-Router ans Internet angeschlossen sind. Diese Situation haben wir nicht real getestet.

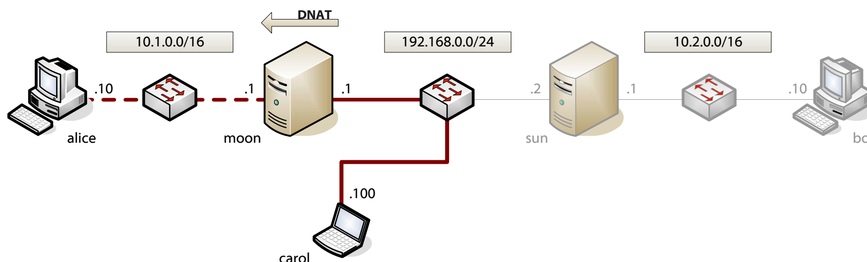


Abbildung 6.5.: UML Test zu DNAT

ikev2-nat-pf Für diesen Test konfigurieren wir auf **moon**, in Form eines Port Forwardings, ein Destination NAT, welches die Ports 500 und 4500 auf den IPsec-Gateway **alice** weiterleitet. Anschliessend öffnet **carol** einen IPsec-Tunnel auf Host **moon** — **carol** ist nicht bekannt dass sie eigentlich mit **alice** kommuniziert (gestrichelte Linie). Wie auch in den anderen Tests verifizieren wir mit einem Ping von **carol** auf **alice**, dass die Verbindung korrekt, mit Ekapsulierung, zustande gekommen ist.

6.2.4. Double NAT (SNAT/DNAT)

Dieses Scenario kombiniert die beiden NAT-Varianten SNAT und DNAT. Diese Situation ist durchaus häufig anzutreffen. Roadwarrior welche nur aus genatteten Netzwerken auf das Internet zugreifen können und Firmen welche über DNAT einen IPsec-Gateway im Internet publizieren, sind wohl eine geläufige Kombination. Wie schon bei der *Double SNAT*-Situation fehlte uns die nötige Hardware um diese Kombination real zu testen. Dies übernimmt ein UML-Test.

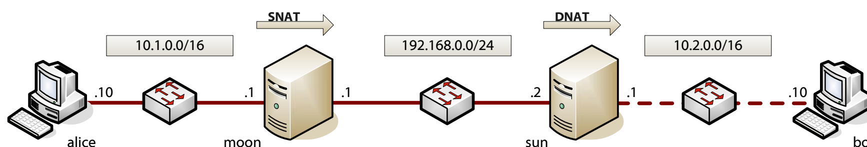


Abbildung 6.6.: UML Test zu Double NAT

ikev2-nat-double Dieser Test verlangt die Konfiguration von zwei NAT. Ein Source NAT auf **moon** sorgt für **alice**'s Zugriff auf das "öffentliche Netz" 192.168.0.0/24, ein Destination

NAT in Form eines Port Forwardings auf **sun** publiziert den IPsec-Gateway **bob** im gleichen Netz. Abbildung 6.6 illustriert diese Situation. Host **alice** baut nun einen Tunnel zu **sun** auf, ohne zu Wissen, dass sie eigentlich mit **bob** kommuniziert (gestrichelte Linie). Auch hier testen wir mit einem Ping ob die Verbindung korrekt zustande gekommen ist.

6.2.5. Spezialfälle

Die folgenden Tests decken einige Spezialfälle ab, die zu testen es sich lohnt. Augrund der bereits angesprochenen *Conntrack*-Problematik konnten diese noch nicht vollständig in die Testsuite integriert werden.

ikev2-nat-portswitch Bei diesem Test wird ein rebootender NAT-Router simuliert. Nach Aufbau des IPsec-Tunnels, analog zum Test `nat-rw-one`, wird auf **moon** die Conntrack-Tabelle geflusht. Beim nächsten DPD sollte auf dem Gateway **sun** ein Adressupdate durchgeführt werden.

ikev2-dpd In diesem Test sollte die Funktionsweise der DPD geprüft werden. Am einfachsten geschieht dies, in dem auf einem der Hosts die Firewall so eingestellt wird, dass alle Pakete geblockt werden.

7. Projektstand

Als Resultat unserer Arbeit ist strongSwan II heute in der Lage, Verbindungen mit IKEv2 in diversen NAT-Szenarien aufzubauen. Allfällige NATs werden detektiert, und alle nötigen Massnahmen getroffen, damit sowohl die IKEv2-Verbindung als auch die darüber ausgehandelten IPsec-Nutzverbindungen trotz NAT zustande kommen und betrieben werden können.

Zusätzlich haben wir die wesentlichen Bestandteile von Dead Peer Detection implementiert. Tote Verbindungen werden detektiert, und soweit es der derzeitige Stand von strongSwan II zulässt, wird darauf reagiert.

Als Nebenprodukt der Arbeit im Umfeld von strongSwan II haben wir diverse Verbessungen und Korrekturen an nicht direkt mit NAT-T oder DPD in Zusammenhang stehendem Code vorgenommen oder vorgeschlagen.

Unsere Arbeit ist in Form von Patches bereits in das strongSwan-Projekt eingeflossen, und wird dort von Martin Willi und Prof. Dr. Andreas Steffen weiterentwickelt werden.

7.1. Zukunftvisionen

7.1.1. Verifikation von NAT-T/NAT-D

Da weder Testvektoren noch andere Implementationen von NAT-T für IKEv2 existieren, konnte unsere Implementation nicht gegenüber anderen Implementationen verifiziert werden. Ein solcher Test wäre aber wertvoll, um die Interoperabilität in der Praxis zu prüfen, denn Tests gegen sich selber testen nur, ob die eigene Implementierung im Sinne des Erfinders funktioniert, nicht aber, ob alle Details und Feinheiten des Standards korrekt umgesetzt wurden. Tests mit anderen Implementationen verschiedener Art wären daher sinnvoll, sobald andere NAT-T-Implementationen für IKEv2 existieren.

7.1.2. DPD mit konfigurierbaren Actions

Dead Peer Detection kennt momentan nur eine einzige Action, wenn detektiert wird, dass der Peer nicht mehr reagiert: die Security Association (SA) wird terminiert, der Eintrag in der SADB wird gelöscht.

Wünschenswert wäre die Unterstützung von weiteren Actions, wie z.B. `hold`, was aber grundsätzlichere Erweiterungen an strongSwan bedingt, weshalb wir das im Rahmen dieser Arbeit nicht realisiert haben.

7.1.3. `conntrack` im UML Test Framework

Weil die UML-Instanzen zwischen Tests nicht neu gestartet werden, sind im Connection Tracking Code des Linux Kernels noch Conntrack-Entries vergangener Tests aktiv, was zu Verfälschungen von Testfällen führen kann (siehe dazu Abschnitt 6). Deshalb ist es wichtig, dass vor jedem Test die Conntrack-DB des Kernels geleert wird. Dies ist mit dem Userspace-Utility `conntrack` realisierbar, welches im Umfeld des Netfilter-Projektes entwickelt wird.

Es wäre wichtig, dieses Utility in den UML-Systemen zu integrieren, und vor jedem Test die Conntrack-DB zu leeren.

`http://www.netfilter.org/`

7.1.4. Netzwerktopologie im UML Test Framework

Die bestehende Netzwerktopologie ist vom UML Test Framework fest vorgegeben, und kann nicht für einzelne Tests verändert oder erweitert werden. Für vollumfängliche Tests aller möglichen NAT-Szenarien aber genügt eine Topologie mit nur 2 Gateways und 3 Segmenten nicht. Will man beispielsweise kaskadierte Double NATs testen, so ist dies mit dem bestehenden Framework nicht möglich.

Leider ist die Topologie fest in der Implementation des Test Frameworks verankert und nicht zentral definiert. Da in jedem Script fest von dieser Topologie ausgegangen wird, ist es ein grösseres Unterfangen, die Topologie zu verändern.

Schön wäre, wenn die Topologie zentral in einer Markupsprache wie XML oder YAML definiert und erweitert werden könnte. Wir sind uns aber bewusst, dass dies eine grössere Änderung am Test Framework bedeuten würde, und deshalb weder ein einfaches noch ein kurzes Unterfangen wäre.

7.1.5. XFRM Address Change Notification Kernel Patch

Wenn der Kernel UDP-enkapsulierte ESP-Pakete empfängt, und dabei eine Adressänderung des Peers feststellt (vgl. Abschnitt 5.3.4), so wäre grundsätzlich vorgesehen, dass der Kernel einen Userspace-IKE-Daemon wie `charon` darüber informieren sollte.

Der Linux Kernel unterstützt zur Zeit (2.6.17) diese Notification nur über die `PF_KEY` -Schnittstelle (`SADBX_X_NAT_T_NEW_MAPPING`-Message), nicht aber über die von `charon` derzeit verwendete `XFRM` -Schnittstelle. Es wäre mit vertretbarem Aufwand möglich, vergleichbare Funktionalität zur `XFRM` -Schnittstelle hinzuzufügen, und anschliessend über einen Patch im Linux Kernel einfliessen zu lassen. Da die eigentliche Funktionalität für `PF_KEY` bereits implementiert ist, kann dort bereits auf die Implementation des eigentlichen Problems zurückgegriffen werden.

7.1.6. Unterstützung für `subnetwithin`

Für die vollständige Unterstützung von Szenarien wie Host zu Host Verbindung über NAT wird es notwendig sein, auch in `charon` den Konfigurationsparameter `subnetwithin` zu unterstützen. Dieser Schritt ist zwar im Rahmen des strongSwan-Projektes geplant, aber noch nicht realisiert worden.

7.1.7. IPv6-Support

strongSwan II wird langfristig nicht um eine vollständige IPv6-Unterstützung herumkommen, weshalb wir grundsätzliche darauf geachtet haben, nicht IPv4 spezifisch zu implementieren. Kleine Erweiterungen werden aber insbesondere im NAT-Detection-Code (Erzeugung der NAT-D-Hashes) nötig sein. Grösserer Aufwand wird unserer Meinung nach die saubere Erweiterung des von uns lediglich unwesentlich erweiterten Typs `host_t` für IPv6 darstellen.

7.1.8. Timeout in Kernel-Interface

Um die Kernel-Kommunikation robuster zu machen, sollte das Kernel-Interface einen Timeout-Mechanismus implementieren. Ansonsten kann es passieren, dass in Fehlersituationen alle Worker-Threads im Kernel-Interface auf Antworten des Kernels warten, die nie kommen.

A. Projektmanagement

A.1. Projektplan

Abbildung A.1.: Projektplan

A.2. Zeitabrechnung

Das folgende Diagramm zeigt den Verlauf des Aufwandes in Stunden, über die gesamte Projektlaufzeit hinweg. Projektwoche 03 war Daniel Röthlisberger abwesend. Die Schwankungen in den Stundenzahlen sind teilweise durch die vielen Feiertage bedingt (Tag der Arbeit, Ostern, Auffahrt, Pfingsten).

Wer	Ø	$\sum$
Tobias Brunner	15.5 h	217 h
Daniel Röthlisberger	15.1 h	212 h
Minimum Modulbeschr.	14.0 h	196 h

Tabelle A.1.: Zeitauswertung: Wochenschnitt und Total

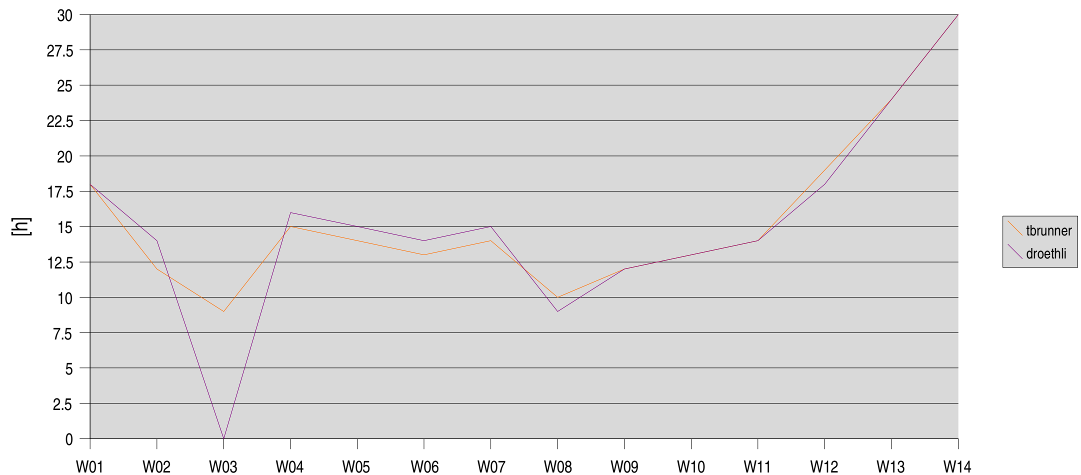


Abbildung A.2.: Zeitabrechnung

A.3. Qualitätssicherung

A.3.1. Massnahmen

Um einen gewissen Qualitätsstandard des von uns entwickelten Source Code zu garantieren, haben wir folgende Massnahmen umgesetzt.

Source Control Management Selbstverständlich haben wir unseren Code auschliesslich unter der Kontrolle eines SCM-Systems entwickelt, um kollaboratives Arbeiten zu ermöglichen, eine komplette History zu Verfügung zu haben, und um Datenverlust zu verhindern. Wir haben hierzu einen privaten Subversion-Server mit Backup-Infrastruktur genutzt.

Tests Soweit sinnvoll und möglich haben wir für unseren Code Unit Tests geschrieben, und ansonsten wie im Kapitel 6 dokumentiert getestet.

Code Reviews Wir führten regelmässig informelle Code Reviews durch. Ein sehr wertvolles Mittel für informelle Code Reviews waren die Commit-Notifications, welche es dem jeweils anderen Team-Mitglied ermöglichten, offensichtliche Fehler und Probleme, welche durch Commits eingeführt wurden, sofort zu entdecken.

Pilot/Kopilot-System In besonders kniffligen Situationen haben wir Fallweise im Pilot/Kopilot-System gearbeitet. Hierbei programmiert das eine Team-Mitglied (Pilot) und erklärt den Code und die Konzepte fortlaufend, während das andere Team-Mitglied (Kopilot) das Gesagte und den Code kritisch hinterfragt, Einwände einbringt und sich an der Lösungsfindung beteiligt.

Programmierrichtlinien Wir haben keine eigenen Richtlinien aufgestellt, und uns stattdessen an die Code-Richtlinien von strongSwan II gehalten.

A.3.2. Effektivität

Wir haben keine statistischen Daten gesammelt, welche eine quantitative Auswertung der Effektivität erlauben würden. Subjektiv gesehen aber haben die umgesetzten Massnahmen sehr viel bewirkt. Fehler wurden innert kürzester Zeit behoben, sei es dank fehlschlagenden Tests oder in Folge einer Code Review. Das Pilot/Kopilot-System hat sich in schwirigen Situationen als Mittel zur sicheren Lösungsfindung bewährt. Bei gewissen Teilen unseres Codes sind wir an die Grenzen des mit sinnvollem Aufwand Machbaren gestossen.

A.4. Protokolle

A.4.1. Sitzung Woche 1

Datum	03.04.2006
Zeit	10:00
Ort	HSR 6.110
Anwesend	as, mw, dr, tb

Aufgabenstellung

- Gemäss schriftlicher Aufgabenstellung.
- Ausführliche Tests sind wichtig!
  - MASQUERADING
  - SNAT
  - DNAT
  - Kombinationen
- Testen mit UML und realer Hardware
- NAT-T Pflicht, DPD optional

Architektur Charon

- Gemäss handschriftlichem Schema von mw
- Multi-Threaded: Worker Threads mit Thread Pool
  - Worker Threads arbeiten Job Queue ab
- Timed Events:
  - Scheduler arbeitet Event Queue ab und füllt Job Queue
- Socket IO: Separate Receiver und Sender Threads
  - Receiver Thread füllt Job Queue
  - Sender Thread arbeitet Send Queue ab
- IKE SA Manager: Checkout/Checkin Verfahren
  - Worker Threads holen sich SA über SA Manager Fassade (synchronisiert)
  - IKE SA: schützt die IKE-Verbindung selber
  - Child SA: schützt Nutzverbindung (ESP, AH)
- Kernel Interface
  - XFRM/Netlink statt PF_KEY

  - UDP Encapsulation muss noch implementiert werden
- Message Processing
  - Parser transformiert flaches Paket in Header/Payload-Objekte
  - Generator transformiert Header/Payload-Objekte in flaches Paket
  - Verify-Methoden auf Header/Payload-Objekten
- Protokoll: Objektorientierte Finite State Machine
  - State == Object
  - State Transitions == Methoden

Administratives

- Wir arbeiten mit Snapshots von mw
- Zurückmergen unseres Codes über Patches
- Evtl. später read-only Zugang zum Subversion-Repo von mw (?)
- Impliziert dass wir eigene SCM-Infrastruktur verwenden
- Entscheid von dr + tb: Wir verwenden Gentoo-Linux

A.4.2. Sitzung Woche 2

Datum	11.04.2006
Zeit	13:10
Ort	HSR 6.110
Anwesend	as, mw, dr, tb

Review Projektplan

- Phasen, Milestones, Arbeitspakete, Zeitplan

Information Infrastruktur

- Trac
  - Logins für as, mw?
  - Arbeiten mit Ticketing
- Subversion
  - Vendor Branches

Konfliktsituationen Tunnel Mode / Transport Mode

- Siehe #21 - nicht Bestandteil der Arbeit!

Formelle Anforderungen

- Projektmanagement: minimal, sinnvoll
- Doku: optimal, sinnvoll
- Testprotokolle: Spezifizieren und protokollieren ja

Diverses

- Es gibt einen neuen IKEv2 Draft (Clarification), unbedingt lesen!

A.4.3. Sitzung Woche 3

Datum	18.04.2006
Zeit	13:10
Ort	HSR 1.206
Anwesend	as, mw, tb

Hinweise

Daniel ist diese Woche abwesend.

Stand der Arbeiten

- Initialisierung abgeschlossen
- Beginn der Arbeit am Kernel Interface

Informationen zu neuem Kernel

- Es gibt neue iptables-Regeln für IPSec im Kernel 2.6.16

A.4.4. Sitzung Woche 4

Datum	25.04.2006
Zeit	13:10
Ort	HSR 1.206
Anwesend	as, mw, dr, tb

Interface/Adressen

- #29
- source:/trunk/charon/network/socket.c
- source:/trunk/charon/network/socket.h
- Neue Methode get_addresses or some such
- Initiator: alle ifaces verwenden
- Responder: über Paket-Infos richtige Addr wählen

Makros in Kernel-Interface

- #25
- mwilli kümmert sich darum

Probleme in Charon

- Unit Test: SIGSEGV in Socket #28
- Fehlerhandling: SIGSEGV in Daemon #26, #30, #31
- mwilli kümmert sich darum

Build-Infrastruktur

- #24
- Es werden nicht alle Objects kompiliert bei Änderungen in Header-Files
- Es wird immer gelinkt auch wenn alle Binaries up to date sind
- Makefiles *sehr* unorthodox aufgebaut
- Build-Infrastruktur wird mit 4er-Release von strongSwan neu entwickelt werden. Bis dann leben wir so gut es geht mit dem, was vorhanden ist.

A.4.5. Sitzung Woche 6

Datum	09.05.2006
Zeit	13:10
Ort	HSR 1.206
Anwesend	as, mw, dr, tb

Stand der Arbeiten

- NAT-D steht und funktioniert, schlecht getestet: #6, #7
- Wechsel auf Port 4500 funktioniert grundsätzlich: #9
- IKE-Encapsulation und Kernel-Config funktioniert: #10, #11
- Socket-Code umgeschrieben: #5, #29
- *Noch offen*: Dyn. Addr. Update: #12

Ausblick

- Beginn mit Tests (UML, Real-World): #16, #17, #18
- Dyn. Addr. Update: #12
- Logging, generell Cleanup unseres Codes: #34

SPI Generation

- Random oder Counter? #35, #36

States Architektur

- Code dupliziert?
- Dyn. Addr. Update
- Refactoring?

Stroke

- 0.0.0.0 resp. %any

Repo

- Merge mit strongSwan
- trunk/Source gibts nicht mehr

A.4.6. Sitzung Woche 7

Datum	16.05.2006
Zeit	13:10
Ort	HSR 1.206
Anwesend	as, mw, dr, tb

Stand der Arbeiten

- Dyn. Addr. Update: #12
  - Kernel Notification fehlt
  - Rest ok

Ausblick Kernel-Update

- Netlink meldet Addr Change Event #40
  - Testen, nutzen
- TS / Policies #41
  - Child SA Update nötig in State IKE_SA_ESTABLISHED
  - Update-Methode implementieren

Ausblick DPD / Keepalives

- Kernel Policies abfragen wann letztes ESP Paket gesendet / empfangen wurde #8
- strongSwan Source: `kernel.c`, `kernel_netlink.c`

Tests

- Für UML-Tests mergen #42
- Mit Merge warten bis Autotools-Migration abgeschlossen

A.4.7. Sitzung Woche 8

Datum	23.05.2006
Zeit	13:10
Ort	HSR 1.206
Anwesend	as, mw, dr, tb

Stand der Arbeiten

- Child SA refactored
  - alloc(), add(), update() unifiziert
- Kernel Interface auf Makros umgebaut #25
- Dyn. Addr. Update: #12
  - Kernel Notification in Arbeit
- Update oder Del/Add
  - Mail an Herbert Xu

Noch offen

- Netlink meldet Addr Change Event #40
  - Nutzen
- TS / Policies #41
  - Child SA Update nötig in State IKE_SA_ESTABLISHED
  - Update-Methode implementieren
- DPD/Keepalives
  - Keepalive Zeiten konfigurieren (Variabeln statisch verdrahten (Zeit in Sek, 0 == disabled), wird bei Merge mit Config verhängt)
- UML-Merge #42

A.4.8. Sitzung Woche 9

Datum	30.05.2006
Zeit	13:10
Ort	HSR 1.206
Anwesend	as, mw, dr, tb

Stand der Arbeiten

- Merge mit aktueller strongSwan Codebase, Tests #42
- Herbert Xu: Del/New statt Update wenn die Primärschlüssel ändern ist korrekt.
  - Kernel Update der SA entsprechend geändert, Update-Chain fertiggestellt #41
- Keepalives vorbereitet (Config, Job/Event) #8

Fragen

- OOD Thread Pool / Jobs #44
  - Wieso Programmlogik der Jobs in prozeduraler Manier im Thread-Pool (switch/case) statt in den Jobs?
- Event Queue Integer Overflows #43
- Charon Unit Tests nicht in Autotools Buildmaschinerie – wer machts? #45 -> mwilli

Noch offen

- Policy-Update fertigstellen analog zu SA #41
- Netlink meldet Addr Change Event #40
- Keepalives fertigstellen #8
- DPD implementieren #15
- Tests, Tests, Tests...

Absprachen

- Aquire/Expire: mwilli merged unseren Kernel-Code.

A.4.9. Sitzung Woche 10

Datum	06.06.2006
Zeit	13:10
Ort	HSR 1.206
Anwesend	as, mw, dr, tb

Stand der Arbeiten

- Nochmals Merge wegen defektem Zertifikats-Lagecode
- OOD Thread Pool / Jobs von mwilli refactored, unser Job entsprechend angepasst #44
- Keepalives fertiggestellt #8
- Policy-Update fertiggestellt #41

Probleme

- is_local_address() für %any
  - TRUE für %any
  - FALSE für %any, explizite Abfrage im stroke_interface
  - *%any immer remote! starter löst auf*
- IKE_AUTH reply decryption failed.
  - Bereits gelöst

Noch offen

- Netlink meldet Addr Change Event #40
  - Aquire/Expire: mwilli merged unseren Kernel-Code.
- DPD implementieren #15
- Tests, Tests, Tests...

Neu zu beachten

- IPv6-Verträglichkeit (kein Bestandteil der Aufgabenstellung)

A.4.10. Sitzung Woche 11

Datum	13.06.2006
Zeit	13:10
Ort	HSR 1.206
Anwesend	as, mw, dr, tb

Stand der Arbeiten

- DPD implementiert #15
  - Timer-Kopplung? (ja)
  - Timer-Parameter? (max 3–5 Min., 6 sek timeout, 6 retries, base 1.5)
  - Config: auto/restart/...; dpd timers pro Connection; keyingtries (0, 1, n)

Probleme (gelöst)

- Problem gelöst: IKE_AUTH reply decryption failed (Fehler bei Merge).
- TS-Problem in Host2Host Setup analysiert
  - Nach Absprache mit mw/as ignoriert

Noch offen

- Kleinere Arbeiten am Kernel Interface
  - Netlink meldet Addr Change Event #40
  - get_policy um last used Timer zu extrahieren

A.4.11. Sitzung Woche 12

Datum	20.06.2006
Zeit	13:10
Ort	HSR 1.206
Anwesend	as, mw, dr, tb

Stand der Arbeiten

- DPD implementiert #15 inkl. Änderungen Sitzung 11
- Doku angefangen
  - Outline an as senden
- UML Problem mit NAT

Noch offen

- Kleinere Arbeiten am Kernel Interface
  - Netlink meldet Addr Change Event #40
  - get_policy um last used Timer zu extrahieren

A.4.12. Sitzung Woche 13

Datum	27.06.2006
Zeit	13:10
Ort	HSR 1.206
Anwesend	as, mw, dr, tb

Stand der Arbeiten

- Idle-Time Refactorings erledigt
  - Unterscheidung Incoming/Outgoing (DPD/Keepalives)
  - Kernel Policies abfragen
  - DPD #15
- UML Problem mit NAT gelöst
  - Fehlende obskure, undokumentierte Socket Option für UDP Encapsulation fehlte noch
  - Nicht klar, wieso Socket Option auf einem Socket nötig ist, damit der Kernel ESP in UDP korrekt auspacken kann
  - Jetzt tuts
- Tests
  - Neue UML-Tests #52
  - UML-Framework:
    - tcpdump Test erweitert (nicht nur j/n sondern Count)
    - eval Test erweitert (Befehle auf Shell ausführen, Exit Status testen)
  - Alle Szenarien inkl. Double-NAT usw funktionieren
- Netlink meldet Addr Change Event #40 machen wir nicht
  - Nur mit PFKEY möglich, in XFRM nicht implementiert
- Merge, hin und her, erledigt
- Doku #23 work in progress

Noch offen

- 1. Prio: Doku
- 2. Prio: Timing-Problem DPD #53
- 3. Prio: Noch mehr Tests (UML, Cisco Pix)

Doku Hints

Nicht vergessen:

- XFRM Patch Address Change von Kernel
- conntrack in UMLs

Vorgaben:

- Kein Glossar, aber Abk-Verz.
- Kap 4: umbenennen in “Grundlagen”
- Abstract nicht in letzter Minute!

A.4.13. Sitzung Woche 14

Datum	04.07.2006
Zeit	13:10
Ort	HSR 1.206
Anwesend	as, mw, dr, tb

Stand der Arbeiten

- Timing-Problem gelöst (Mini-Patch Kernel-Interface)
- Mehr UML-Tests
- Doku...

Noch offen

- Doku

B. Erfahrungsberichte

B.1. Tobias Brunner

Es freut mich ein weiteres mal auf eine spannende und äusserst lehrreiche Studienarbeit zurückblicken zu können.

Die intensive Arbeit in einem komplexen Themengebiet erforderte, dass wir uns innert kürzester Zeit enorm viel Wissen zu den verschiedensten Themen aneigneten. Einerseits galt es sich natürlich in strongSwan II einzuarbeiten, andererseits in Technologien wie IKEv2, UDP-ENCAP oder NETLINK/XFRM. So studierten wir RFCs, befragten Google und begaben uns, aufgrund mangelnder Dokumentation, nicht selten in die Niederungen der Kernel-Sourcen. Viele dieser Entdeckungsreisen liessen mich zur durchaus positiven Erkenntnis gelangen, dass auch die Linux Gurus nur mit Wasser kochen.

Die Arbeit mit Linux und der Programmiersprache C war für mich in dieser Intensität auch eine neue Erfahrung. Als langjähriger Windows-Benutzer muss ich aber mittlerweile zugeben, dass "the unix way" für gewisse Arbeiten ausserordentlich produktiv sein kann.

Interessant war für mich ausserdem die Mitarbeit an einem grösseren Open Source Projekt. Die Gewissheit an etwas mitzuwirken, das nicht in irgendeiner Schublade verschwinden, sondern tatsächlich eingesetzt wird, lässt mich auch mit einem Gefühl von Zufriedenheit und Stolz auf die abgeschlossene Arbeit zurückblicken.

Wie bis anhin zeigten wir unsere Schwächen in der Zeitplanung. Die eingeplanten Reserven ermöglichten uns aber alle gesteckten Ziele termingerecht zu erreichen.

Die Zusammenarbeit mit Daniel hat wiederum ausgezeichnet geklappt. Durch seine langjährige Erfahrung mit Open Source Software konnte ich einmal mehr von seinem breiten Wissen auf diesem Gebiet profitieren. Die produktiven Diskussionen im Team und mit unseren kompetenten Betreuern, Prof. Dr. Andreas Steffen und Martin Willi, waren auch von unschätzbarem Wert.

Mit einem erfolgreichen Abschluss des Projektes einher geht meine persönliche Gewissheit, sowohl in technischer als auch menschlicher Hinsicht noch länger von den gemachten Erfahrungen profitieren zu können.

Darüber hinaus lassen mich die mit viel gemeinsamem Einsatz überwundenen grösseren und kleineren Herausforderungen motiviert auf die kommende Diplomarbeit blicken.

B.2. Daniel Röthlisberger

Einmal mehr kann ich nach vielen investierten Stunden auf ein sehr interessantes Projekt zurückblicken. Unzählige kleine und grosse Probleme — Herausforderungen — haben sich uns in den Weg gestellt, und ich habe dadurch im Verlauf des Projektes viel für mich persönlich profitieren können.

Wir haben uns innert kürzester Zeit in die komplexe Welt von IPsec eingearbeitet, uns vertieft mit den RFCs und Drafts zu IKEv2, IKEv1, NAT-T und DPD auseinandergesetzt, und schliesslich uns in den Quellcode und die Architektur von strongSwan II eingearbeitet. Hinzu kam die Arbeit mit der Kernel-Schnittstelle von Linux, welche nicht sonderlich gut bis gar nicht dokumentiert ist, getreu dem Motto “Read the source, Luke”. Die Erfahrung und das Fachwissen unserer Betreuer, Prof. Dr. Andreas Steffen und Martin Willi, waren uns hierbei sicher eine wesentliche Hilfe, doch am Ende half oft nur noch der Blick in die Kernel-Sourcen.

Unser Team hat während dieser Zeit wie gewohnt sehr gut harmoniert, was ein wesentlicher Faktor für den Erfolg eines jeden Projektes ist. Wir haben uns im Team gegenseitig bestens ergänzt. Tobias überraschte mich regelmässig mit frisch angeeignetem Wissen oder neuen Aspekten zu Themen, die ich zuvor geglaubt hatte, vollständig verstanden zu haben.

Unser Zeitmanagement war sicher nicht perfekt, wir haben aber im Vergleich zur letzten Studienarbeit Fortschritte gemacht. Ein Grund hierfür war eine realistischere Planung. Wir konnten die geplanten Ziele dank Reserven fristgerecht umsetzen, und ich bin deshalb selber zufrieden mit unserer Leistung und dem erreichten Resultat.

Am Ende des Projektes angelangt bin ich froh, dass wir dieses Projekt in Angriff genommen haben. Es hat mich auf jeden Fall für die Diplomarbeit motiviert, und ich bin überzeugt, dass sich die vielfältigen gemachten Erfahrungen nicht nur dort auszahlen werden. Und zu guter Letzt bleibt die Gewissheit, nicht für den Papierkorb gearbeitet zu haben, sondern einen Beitrag an ein Open-Source-Projekt geleistet zu haben, welches ausserhalb des akademischen Elfenbeinturms auch effektiv eingesetzt wird und sich bewährt.

Abkürzungsverzeichnis

AH Authentication Header.

DNAT Destination Network Address Translation.

DPD Dead Peer Detection.

DRY Don’t Repeat Yourself.

ESP Encapsulating Security Payload.

FD File Descriptor.

FreeS/WAN Free Secure Wide Area Network.

FTP File Transfer Protocol.

HMAC (Keyed) Hash Message Authentication Code.

IANA Internet Assigned Numbers Authority.

ICMP Internet Control Message Protocol.

IETF Internet Engineering Task Force.

IKE Internet Key Exchange.

IP Internet Protocol.

IPC Inter-Process Communication.

IPsec Internet Protocol Security.

ISAKMP Internet Security Association and Key Management Protocol.

ISP Internet Service Provider.

ITU International Telecommunications Union.

KLIPS KerneL Internet Protocol Security.

LAN Local Area Network.

NAPT Network Address Port Translation.

NAT Network Address Translation.

PAT Port Address Translation.

PGP Pretty Good Privacy.

SA Security Association.

SAD Security Association Database.

SADB Security Association Data Base.

SHA-1 Secure Hash Algorithm 1.

SNAT Source Network Address Translation.

SPD Security Policy Database.

SSH Secure SHell.

SSL Secure Sockets Layer.

SUA Single User Account.

TCP Transmission Control Protocol.

TLS Transport Layer Security.

UDP User Datagram Protocol.

VPN Virtual Private Network.

WAN Wide Area Network.

X.509 X.509 (Zertifikatsformat, ITU-Standard).

Tabellenverzeichnis

5.1. Retransmission Timings 43
5.2. Zusammenfassung unserer Tickets an `mwilli` 46

A.1. Zeitauswertung: Wochenschnitt und Total 57

Abbildungsverzeichnis

4.1. NAT in der Übersicht 8
4.2. NAT: Beispiel 9
4.3. AH/ESP in Transport Mode 13
4.4. AH/ESP in Tunnel Mode 13
4.5. IKEv2 Meldungsaustausch mit Payloads 15
4.6. NAT-Detection Payloads 16
4.7. Aufbau Notify-Payload für NAT-Detection 17
4.8. UDP-Encapsulation 17
4.9. Non-ESP Marker 18
4.10. NAT-Keepalive 18
4.11. IPsec in Linux 19
4.12. Messageaufbau mit Makros 23
4.13. Payload mit `rtnetlink` Makros 23
4.14. `XFRM` Nachricht um eine SA zu erstellen 25

5.1. Architektur von `charon` 28
5.2. Zustände der IKE-SA 29

6.1. “Realer” Testaufbau 47
6.2. UML-Testumgebung 48
6.3. UML-Tests zu SNAT 50
6.4. UML Test zu Double SNAT 50
6.5. UML Test zu DNAT 51
6.6. UML Test zu Double NAT 51

A.1. Projektplan 57
A.2. Zeitabrechnung 58

Listings

4.1. Netlink Socket . . . 21
4.2. Netlink Header . . . 22
4.3. Netlink Adressen . . . 22
4.4. Message Aufbau in `add_sa` . . . 24
4.5. Message Aufbau in `add_sa` . . . 25
5.1. Berkeley Packet Filter auf dem Raw-Socket . . . 34
5.2. Aktivierung der UDP Encapsulation im Kernel . . . 36
5.3. Aktualisierung einer SA . . . 38
5.4. Aktualisierung einer SA (Ports) . . . 39

Literaturverzeichnis

[BA97] Baker, F. ; Atkinson, R.: *RIP-2 MD5 Authentication*. RFC 2082 (Proposed Standard). http://www.ietf.org/rfc/rfc2082.txt. Version: Januar 1997 (Request for Comments)

[BMR+96] Buschmann, Frank ; Meunier, Regine ; Rohnert, Hans ; Sommerlad, Peter ; Stal, Michael: *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996. – ISBN 0–471–95869–7

[CSFP06] Collins-Sussman, Ben ; Fitzpatrick, Brian W. ; Pilato, C. M.: *Version Control with Subversion, Version 1.2*. O'Reilly Media http://svnbook.red-bean.com/

[EF94] Egevang, K. ; Francis, P.: *The IP Network Address Translator (NAT)*. RFC 1631 (Informational). http://www.ietf.org/rfc/rfc1631.txt. Version: Mai 1994 (Request for Comments). – Obsoleted by RFC 3022

[FLYV93] Fuller, V. ; Li, T. ; Yu, J. ; Varadhan, K.: *Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy*. RFC 1519 (Proposed Standard). http://www.ietf.org/rfc/rfc1519.txt. Version: September 1993 (Request for Comments)

[GHJV95] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. – ISBN 0–201–63361–2

[HBR04] Huang, G. ; Beaulieu, S. ; Rochefort, D.: *A Traffic-Based Method of Detecting Dead Internet Key Exchange (IKE) Peers*. RFC 3706 (Informational). http://www.ietf.org/rfc/rfc3706.txt. Version: Februar 2004 (Request for Comments)

[Hor04] Horman, Neil: *Understanding And Programming With Netlink Sockets* http://people.redhat.com/nhorman/

[HSV+05] Huttunen, A. ; Swander, B. ; Volpe, V. ; DiBurro, L. ; Stenberg, M.: *UDP Encapsulation of IPsec ESP Packets*. RFC 3948 (Proposed Standard). http://www.ietf.org/rfc/rfc3948.txt. Version: Januar 2005 (Request for Comments)

[HW05] Hutter, Jan ; Willi, Martin: *strongSwan II, Eine IKEv2-Implementierung für Linux*. Hochschule für Technik Rapperswil, 2005

[KA98a] Kent, S. ; Atkinson, R.: *IP Authentication Header*. RFC 2402 (Proposed Standard). http://www.ietf.org/rfc/rfc2402.txt. Version: November 1998 (Request for Comments). – Obsoleted by RFC 4302

[KA98b] Kent, S. ; Atkinson, R.: *IP Encapsulating Security Payload (ESP)*. RFC 2406 (Proposed Standard). http://www.ietf.org/rfc/rfc2406.txt. Version: November 1998 (Request for Comments). – Obsoleted by RFCs 4303, 4305

[KA98c] Kent, S. ; Atkinson, R.: *Security Architecture for the Internet Protocol*. RFC 2401 (Proposed Standard). http://www.ietf.org/rfc/rfc2401.txt. Version: November 1998 (Request for Comments). – Obsoleted by RFC 4301, updated by RFC 3168

[Kau05] Kaufman, C.: *Internet Key Exchange (IKEv2) Protocol*. RFC 4306 (Proposed Standard). http://www.ietf.org/rfc/rfc4306.txt. Version: Dezember 2005 (Request for Comments)

[KSHV05] Kivinen, T. ; Swander, B. ; Huttunen, A. ; Volpe, V.: *Negotiation of NAT-Traversal in the IKE*. RFC 3947 (Proposed Standard). http://www.ietf.org/rfc/rfc3947.txt. Version: Januar 2005 (Request for Comments)

[Mas98] Masinter, L.: *Hyper Text Coffee Pot Control Protocol (HTCPCP/1.0)*. RFC 2324 (Informational). http://www.ietf.org/rfc/rfc2324.txt. Version: April 1998 (Request for Comments)

[MMP98] McDonald, D. ; Metz, C. ; Phan, B.: *PF_KEY Key Management API, Version 2*. RFC 2367 (Informational). http://www.ietf.org/rfc/rfc2367.txt. Version: Juli 1998 (Request for Comments)

[Moy98] Moy, J.: *OSPF Version 2*. RFC 2328 (Standard). http://www.ietf.org/rfc/rfc2328.txt. Version: April 1998 (Request for Comments)

[MR04] Marchionni, Eric ; Rayo, Patrik: *User-Mode-Linux Test Suite für Linux strongSwan*. Zürcher Hochschule Winterthur, 2004

[MSST98] Maughan, D. ; Schertler, M. ; Schneider, M. ; Turner, J.: *Internet Security Association and Key Management Protocol (ISAKMP)*. RFC 2408 (Proposed Standard). http://www.ietf.org/rfc/rfc2408.txt. Version: November 1998 (Request for Comments). – Obsoleted by RFC 4306

[Pip98] Piper, D.: *The Internet IP Security Domain of Interpretation for ISAKMP*. RFC 2407 (Proposed Standard). http://www.ietf.org/rfc/rfc2407.txt. Version: November 1998 (Request for Comments). – Obsoleted by RFC 4306

[Pos81] Postel, J.: *Internet Protocol*. RFC 791 (Standard). http://www.ietf.org/rfc/rfc791.txt. Version: September 1981 (Request for Comments). – Updated by RFC 1349

[RL93] Rekhter, Y. ; Li, T.: *An Architecture for IP Address Allocation with CIDR*. RFC 1518 (Proposed Standard). http://www.ietf.org/rfc/rfc1518.txt. Version: September 1993 (Request for Comments)

[RP94] Reynolds, J. ; Postel, J.: *Assigned Numbers*. RFC 1700 (Historic). http://www.ietf.org/rfc/rfc1700.txt. Version: Oktober 1994 (Request for Comments). – Obsoleted by RFC 3232

[SE01] Srisuresh, P. ; Egevang, K.: *Traditional IP Network Address Translator (Traditional NAT)*. RFC 3022 (Informational). http://www.ietf.org/rfc/rfc3022.txt. Version: Januar 2001 (Request for Comments)

[Ste92] Stevens, W. R.: *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992. – ISBN 0–201–56317–7

[wik] *Wikipedia*. WWW. http://www.wikipedia.org/