Analyse und Vervollständigung der Unterstützung von ISATAP als IPv6 Migrationsstrategie unter Linux

Masterarbeit

2. November 2009

Sascha Hlusiak
Hochschule Ravensburg-Weingarten
Master Informatik

Betreut durch

Harald Usadel
Hochschule Ravensburg-Weingarten
Frank Fechter
Hochschule Ravensburg-Weingarten

Für Alina

Erklärung

Diese Arbeit wurde an der Hochschule Ravensburg-Weingarten als Abschlussarbeit zum Erlangen des Grades "Master of Science" angefertigt.

Hiermit versichere ich, die vorliegende Masterarbeit und Implementierung selbständig angefertigt und dabei nur die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Weingarten, 2. November 2009

Unterschrift

Danksagung

An dieser Stelle möchte ich die Gelegenheit nutzen, mich bei allen zu bedanken, die mich während meines Studiums und bei der Erstellung dieser Arbeit in vielfacher Art und Weise unterstützt haben.

Mein besonderer Dank gilt meinen Eltern, ohne deren große Unterstützung mein Studium und diese Arbeit nicht möglich gewesen wären.

Herrn Professor Usadel und Herrn Professor Fechter danke ich, dass mir das Thema zur Verfügung gestellt werden konnte.

Ich danke *Fred L. Templin* für das das Entwickeln neuer Techniken, speziell `rfc5214`, die als Grundlage für diese Arbeit dient und für seine Unterstützung und Zusammenarbeit. Dank gebürht *Bernhard Schmidt* (LRZ München) für seinen Einsatz zur Verbreitung von IPv6 im deutschsprachigem Raum.

Ebenso möchte ich mich bei allen Entwicklern von freier Software bedanken, durch die die Welt zu einem netteren Ort wird. Ohne unentgeltliche Nutzung deren Programme und Quelltexte wären Projekte wie dieses nicht realisierbar.

Kurzzusammenfassung

Das Internet Protokoll Version 4 (IPv4), auf dem nahezu das gesamte Internet aufbaut, gerät langsam in die Jahre und die zur Verfügung stehenden IPv4 Adressen sind bald aufgebraucht. Der Nachfolger IPv6 räumt zwar mit einigen Unzulänglichkeiten auf, hat aber durch Inkompatibilitäten noch Schwierigkeiten, sich durchzusetzen. Migrationsmechanismen, die beide Techniken vereinen und einen sanften Übergang versprechen, sind in dieser Phase der Entwicklung äußerst wichtig.

Über Tunneltechniken lässt sich die neue Technik über die alte Infrastruktur verwenden, ohne alle Netzkomponenten austauschen zu müssen. Über ISATAP können IPv6 Pakete über das bestehende IPv4 Internet übertragen werden und so Nutzer erschlossen werden, die noch nicht über die notwendige Ausstattung der neuen Technik verfügen. Bisher liegt mit `rfc5214` eine Spezifikation dieses Protokolls vor, die zwar von Microsoft Windows Vista und Cisco bereits unterstützt wird, unter Linux aber bisher nur mit Schwierigkeiten nutzbar ist.

Neben Grundlagen zu IPv6, IP-Tunneln und ISATAP, werden im Rahmen dieser Arbeit verschiedene ISATAP Implementierungen genau untersucht und insbesondere die Implementierung unter Linux analysiert und vorangetrieben. Nach Abschluss der Arbeit sollen umfangreiche Kenntnisse über die Verarbeitung von Tunnelverkehr im Linux Kernel sowie eine einfach zu nutzende Implementierung von ISATAP zur Verfügung stehen.

Inhaltsverzeichnis

Erklärung v

Danksagung vii

Kurzzusammenfassung ix

I. Einführung 1

1. Über diese Arbeit 3

2. Einführung IPv6 5

3. Tunnelmechanismen als Migrationsstrategie 9
3.1. proto-41 Tunnel . . . 9
3.1.1. IPv4-kompatible Adressen . . . 10
3.1.2. 6to4 . . . 13
3.1.3. 6over4 . . . 15
3.1.4. ISATAP . . . 17
3.2. Teredo . . . 19
3.3. Tunnelbroker . . . 21

II. Analyse von ISATAP 25

4. Definition des Protokolls 27
4.1. RFC 4214/5214 . . . 27
4.1.1. Adressformat . . . 27
4.1.2. Automatische Konfiguration von ISATAP Schnittstellen . . . 28
4.1.3. Stateless Autoconf . . . 28
4.1.4. Potential Router List . . . 29
4.1.5. ICMPv4 und ICMPv6 Fehler . . . 31
4.1.6. Absenderprüfung auf einem ISATAP Interface . . . 31
4.2. Sicherheitsmaßnahmen . . . 32
4.2.1. Wahl der Netztopologie . . . 32
4.2.2. Strenge Trennung des ISATAP Links . . . 32
4.2.3. Maßnahmen auf Clients . . . 36

4.2.4. ISATAP Spoofingszenarien . . . 37
4.2.5. Maßnahmen auf Routern . . . 39
4.2.6. Kreisrouting zwischen ISATAP Links . . . 40
4.2.7. Spoofing bei ICMPv4 Fehlern . . . 42

5. Unterstützung gängiger Betriebssysteme 43
5.1. Windows . . . 43
5.1.1. Windows XP . . . 43
5.1.2. Windows Vista . . . 48
5.1.3. Windows 7 . . . 49
5.2. Cisco IOS . . . 50
5.2.1. Konfiguration als ISATAP Router . . . 50
5.2.2. Prüfung von ISATAP Adressen . . . 51
5.2.3. ICMPv4 . . . 52
5.2.4. Fazit . . . 52
5.3. FreeBSD . . . 53
5.4. MacOSX . . . 53
5.5. Linux 2.6.30 . . . 54
5.5.1. isatapd aus dem Miredo Projekt . . . 54
5.5.2. Konfiguration SIT Tunnel . . . 56
5.5.3. Konfiguration ISATAP Tunnel . . . 57
5.5.4. Linux als ISATAP Router . . . 59
5.5.5. Potential Router List . . . 59
5.5.6. isatapd aus ndisc6-0.9.9 . . . 60
5.5.7. ICMPv4 Fehler . . . 62

III. Implementierung 63

6. Pflichtenheft 65
6.1. Stateless Autoconf im Kernel . . . 65
6.2. ICMPv4 Fehler im Kernel behandeln . . . 65
6.3. Unterstützung für PRL in iproute2 . . . 65
6.4. Userspace Daemon für ISATAP . . . 66

7. Linux Kernel 67
7.1. Analyse . . . 67
7.1.1. Schnittstelle zwischen Userspace und Tunnel . . . 68
7.1.2. Schnittstelle zwischen Userspace und PRL . . . 71
7.1.3. Konfiguration des Tunnels im Kernel . . . 73
7.1.4. Duplicate Address Detection (DAD) . . . 75
7.1.5. Stateless Autoconf . . . 77
7.1.6. Empfang von Datenverkehr . . . 79
7.1.7. Senden von Datenverkehr . . . 83

7.1.8. Zusammenfassung der ISATAP Unterstützung 93
7.2. Lösung 94
7.2.1. Stateless Autoconf 94
7.2.2. ICMPv4 Fehler des Tunnels 96

8. Implementierung Userspace 99
8.1. iproute2 99
8.2. isatapd-0.9.5 100
8.2.1. Initialisierung 101
8.2.2. PRL und Router Solicitations 101
8.2.3. Hauptschleife 103
8.2.4. Bedienung 105
8.2.5. Unzulänglichkeiten 106
8.3. isatapd-0.9.6 107
8.3.1. Vater Prozess 107
8.3.2. Kind Prozess 109
8.3.3. Bedienungsanleitung 116
8.3.4. Bekannte Probleme 117
8.4. Veröffentlichung und Wartung 118

9. Zusammenfassung und Ausblick 119

IV. Anhänge 121

A. Entwickelte Patches für Linux 123

Abbildungsverzeichnis 137

Listings 139

Literaturverzeichnis 141

Teil I.

Einführung

1. Über diese Arbeit

Diese Arbeit beschäftigt sich mit einer detaillierten Analyse der Implementierung von ISATAP unter Linux und anderen Betriebssystemen.

Teil I beschäftigt sich mit einer generellen **Einführung** in IPv6 und Tunnelmechanismen allgemein. Grundlegende Netzwerkkenntnisse von IPv4 und IPv6 werden vorausgesetzt.

In Teil II geht es eine detaillierte **Analyse** von ISATAP anhand `rfc5214` aber auch um Analyse und Bewertung von Angriffsszenarien. Zudem wird die Implementierung in gängigen Betriebssystemen, wie Microsoft Windows, Linux, MacOSX, Cisco-IOS und FreeBSD, zum Zeitpunkt dieser Arbeit untersucht.

Teil III stellt die gefundenen Erkenntnisse und die eigene **Implementierung** detailliert auf technischer Basis dar. Grundkenntnisse in C/C++ und des Linux Kernels werden empfohlen.

In Teil IV finden sich im **Anhang** eine detaillierte Auflistung der in dieser Arbeit entwickelten Patches, sowie Abbildungs-, Listings- und Literaturverzeichnisse.

Für Übersicht beim Lesen sorgen die folgenden Formatierungen:

- Quellcodeausschnitte werden in `nicht-proportionaler Schrift` dargestellt und wenn möglich farblich hervorgehoben.
- Befehle auf der Kommandozeile werden in einer `nicht-proportionaler Schrift` dargestellt.
- Dateinamen sind *kursiv* geschrieben.
- Namen von Modulen und Komponenten werden **halbfett** hervorgehoben.

2. Einführung IPv6

In Kürze IPv6 ist ein Layer 3 Protokoll, das auf lange Sicht das derzeit gängige IPv4 als Netzwerkprotokoll ablösen soll. IPv6 ist gänzlich inkompatibel zu IPv4, wird aber bereits von den meisten gängigen Betriebssystemen und Applikationen unterstützt.

Abbildung 2.1.: OSI Modell

...	...
Layer 4 (transport)	UDP/TCP
Layer 3 (network)	IPv4/IPv6
Layer 2 (link)	Ethernet
Layer 1 (physical)	(Kabel)

Adressraum Mit einer IPv4 Adresse von 32 Bit ergibt sich ein Adressraum von $2^{32} = 4,294,967,296$ möglichen Adressen. Dieser ist bald erschöpft und aufgrund dessen werden zunehmend Techniken wie *Network Address Translation* nötig um mehrere Geräte hinter mit einzelnen IPv4 Adresse anschließen zu können. Dies trifft besonders Privathaushalte, dessen Internet Provider meist nur eine öffentliche IPv4 Adresse pro Anschluss zuteilt. Dabei verletzt NAT das Ende-Zu-Ende Prinzip und ruft wiederum Techniken wie NAT-Traversal[1] hervor, um diese Einschränkungen zu umgehen.

Der IPv6 Adressraum umfasst theoretisch nutzbare $2^{128} = 3.4 \cdot 10^{38}$ Adressen, von denen ein Teil reserviert ist. Der Überfluss an Adressen ermöglicht es auch, Privathaushalten ganze Netze zuzuteilen und jedes Endgerät mit einer eigenen Adresse auszustatten. NAT und NAT-Traversal werden überflüssig und das Ende-Zu-Ende Prinzip ist wiederhergestellt.

Adressformat Der größte Unterschied für den Endanwender ist der wesentlich größere Adressraum. Eine IPv6-Adresse besteht aus 128 Bit und wird in 8 Blöcke mit je 16 Bit unterteilt. Hierbei können in den einzelnen Blöcken führende Nullen weggelassen werden, ein `::` steht für eine beliebige Anzahl von Null-Blöcken. Oft kann statt den letzten zwei Blöcken (=32 Bit) auch eine IPv4 Adresse geschrieben werden. Folgende Schreibweisen bezeichnen demnach dieselbe gültige IPv6 Adresse:

- `2001:7c0:1900:11:0:5efe:8d45:106`
- `2001:07c0:1900:0011:0000:5efe:8d45:0106`
- `2001:07c0:1900:0011::5efe:8d45:0106`

[1] NAT-Traversal benutzt spezielle UDP Pakete, um Firewalls und NAT-Routern eine bestehende Verbindung vorzutäuschen, um Ende-Zu-Ende Kommunikation zu ermöglichen: `http://en.wikipedia.org/wiki/NAT_traversal`

- `2001:07c0:1900:0011::5efe:141.69.1.6`

Prefix Die CIDR-Schreibweise aus IPv4 wurde übernommen und bezeichnet die Anzahl der unveränderlichen Bits eines Teiladressraums. Ein `::/8` bezeichnet zum Beispiel ein Subnetz oder *Prefix*, in dem die ersten 8 Bit fest sind und die verbleibenen 120 Bits verändert werden können (das Subnetz beinhaltet also 2^{120} IPv6 Adressen). Üblicherweise wird einer *Site*[2] ein `::/48` Prefix zugeteilt, welches diese weiter in physikalische Netzsegmente zu je `::/64` unterteilt. Beispielsweise wurde der Hochschule Ravensburg-Weingarten das global geroutete Prefix `2001:7c0:1900::/48` zugeteilt und verfügt somit über rund $2^{80} = 1.2 \cdot 10^{24}$ frei wählbare IPv6 Adressen.

Wichtige Prefixe Wichtige Adressräume von IPv6 sind:

`2000::/3` Momentan global geroutete IPv6 Adressen.

`::0.0.0.0/96` IPv4-kompatible IPv6 Adresse (Kapitel 3.1.1).
In `rfc4291, Abschnitt 2.5.5`[1] werden diese Adressen bereits als veraltet bezeichnet.

`2002::/16` Adressprefix für den 6to4 Tunnelmechanismus (Kapitel 3.1.2).

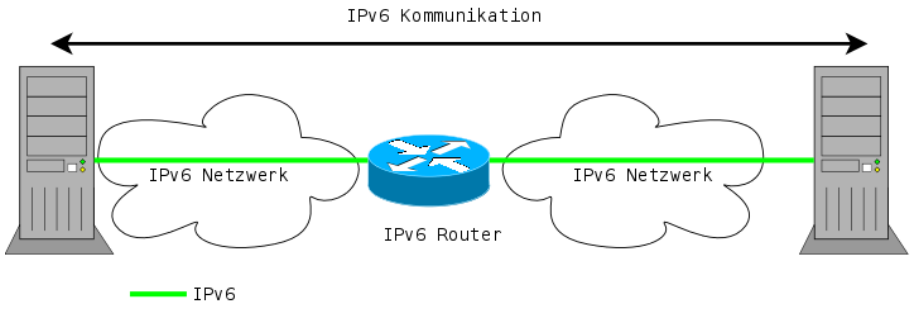
`2001:0000::/32` Reserviert für den Teredo Tunnelmechanismus (Kapitel 3.2).

`::ffff:0.0.0.0/96` IPv4-mapped IPv6 Adressen (Kapitel 3.1.1). Reine IPv4 Kommunikation mit einem Host über einen IPv6 Socket, *keine* echte IPv6 Kommunikation.

`fe80::/10` Link-lokale Adressen. Diese sind nur im Netzwerksegment gültig und werden nicht geroutet. Sie spielen eine wichtige Rolle bei *Stateless Autoconfiguration*.

`ff00::/8` IPv6 Adressen für Multicast.

Abbildung 2.2.: Natives IPv6



[2] Als *Site* wird die Organisation bezeichnet, die die IPv6 Endgeräte besitzt, z.B. eine Hochschule oder ein Privatanwender.

Natives IPv6 Da IPv6 nicht kompatibel zu IPv4 ist, müssen Netzwerksegment und Netzteilnehmer IPv6 als Layer 3 unterstützen. Von nativem IPv6 spricht man, wenn IPv6 direkt auf Layer 2 aufsetzt und als Transportschicht verwendet wird. Obwohl die meisten Betriebssysteme natives IPv6 unterstützen, ist der Betrieb oft wegen alter Hardware oder ungeeigneten Netzsegmenten nicht möglich. So ist eine IPv6 Unterstützung in gängigen Heimroutern fast nicht vorhanden, das DSL Segment zum Internetprovider bietet üblicherweise noch kein natives IPv6, aber auch alte Cisco-Hardware kann Probleme machen.

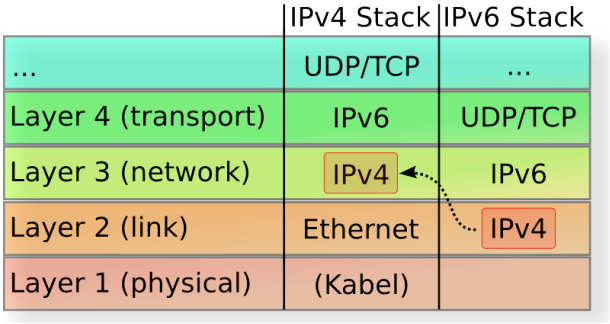
Die nachfolgenden Kapitel beschäftigen sich mit entsprechenden Tunnelmechanismen für IPv6, über die sich Netzwerksegmente überbrücken lassen, die noch nicht vollständig IPv6-fähig sind.

3. Tunnelmechanismen als Migrationsstrategie

3.1. proto-41 Tunnel

Die einfachste Art eines Tunnels ist ein in `rfc4213`[2] beschriebener, statischer IPv6-in-IPv4 Tunnel. Hierbei tritt IPv6 als Payload von IPv4 Paketen auf, im IPv4 Header wird dies durch Protokoll Nummer 41 gekennzeichnet. Da direkt nach dem IPv4 Header der IPv6 Header folgt, hat diese Technik den sehr geringen Overhead von nur 20 Bytes und ist verbindungslos.

Abbildung 3.1.: OSI Modell bei proto-41 Tunnel



In Bezug auf die IPv6 Verbindung befindet sich unterhalb von IPv6 (Layer 3) im OSI-Modell nun IPv4 als Link-Layer. Aus Sicht des IPv4 Stacks sind die Datenpakete des Tunnels wiederum Layer 3, die auf z.B. Ethernet als Layer 2 aufsetzen.

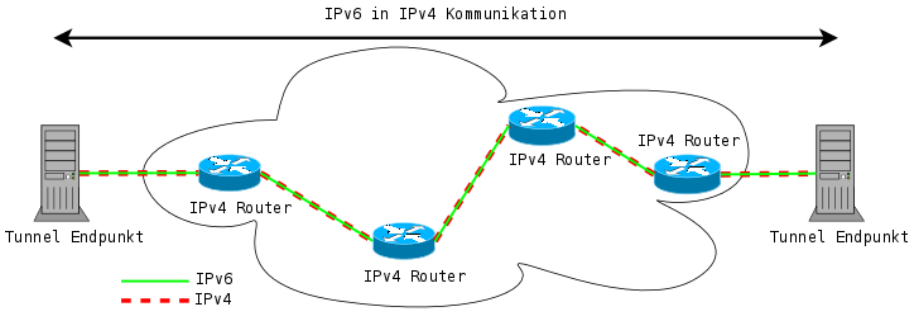
Die Endpunkte müssen sich bei einem solchen Tunnel direkt über IPv4 erreichen können, es wird kein zusätzliches Protokoll "gesprochen". Ein Betrieb über NAT ist nur mit viel Aufwand möglich, wenn als Endpunkt die öffentliche IPv4 Adresse des NAT-Routers verwendet wird und *Proto-41* entsprechend weitergeleitet wird.

Da es sich um eine Punkt-zu-Punkt Verbindung und damit um ein eigenes Netzsegment handelt, ist sogar Multicast möglich.

Sicherheit Da statische *proto-41* Tunnel verbindungslos sind, lassen sich leicht Pakete in den Tunnel einschleusen, indem ein Angreifer die Quelladresse des IPv4 Pakets auf die Adresse einer Gegenseite setzt, sodass der Empfänger des Pakets dieses annimmt. Abhilfe für *Source Address Spoofing* kann z.B. IPSec schaffen, die eine zusätzliche Schicht einfügt und kryptographisch sichert.

Für IPv6 Anwendungen ist der darunterliegende *proto-41* Tunnel komplett transparent aber mit geschickt gesetzten IPv4 Headern lässt sich die Präsenz solcher Tunnel auf

Abbildung 3.2.: Statischer proto-41 Tunnel



einem Pfad erkennen[1]

Einrichten unter Linux Die Technik wird unter Linux "Simple Internet Transition" (SIT) genannt und ist sehr schnell eingerichtet:

Listing 3.1: Einrichten eines statischen SIT Tunnels unter Linux

```
node1 ~ $ ip tunnel add name sit1 local 192.168.6.5 remote 192.168.6.6
node1 ~ $ ip link set sit1 up
node1 ~ $ ip -6 addr add fecd::1234/64 dev sit1

node2 ~ $ ip tunnel add name sit1 local 192.168.6.6 remote 192.168.6.5
node2 ~ $ ip link set sit1 up
node2 ~ $ ip -6 addr add fecd::5678/64 dev sit1

node2 ~ $ ping6 fecd::1234
PING fecd::1234(fecd::1234) 56 data bytes
64 bytes from fecd::1234: icmp_seq=1 ttl=64 time=3.16 ms
```

Eignung Ein statischer proto-41 Tunnel eignet sich für Systeme mit statischen IPv4 und IPv6 Adressen, insbesondere für Router, um einfach zwei IPv6 Netze über IPv4 zu überbrücken. Für Endsysteme ist die Technik nur bedingt geeignet, da sie mit nicht geringem Konfigurationsaufwand auf beiden Seiten verbunden ist und nicht für NAT geeignet ist.

Ein Tunnel kann später transparent durch native IPv6-Konnektivität ersetzt werden, sobald das Netzsegment es unterstützt. Ein Renumbering[2] ist dabei nicht nötig.

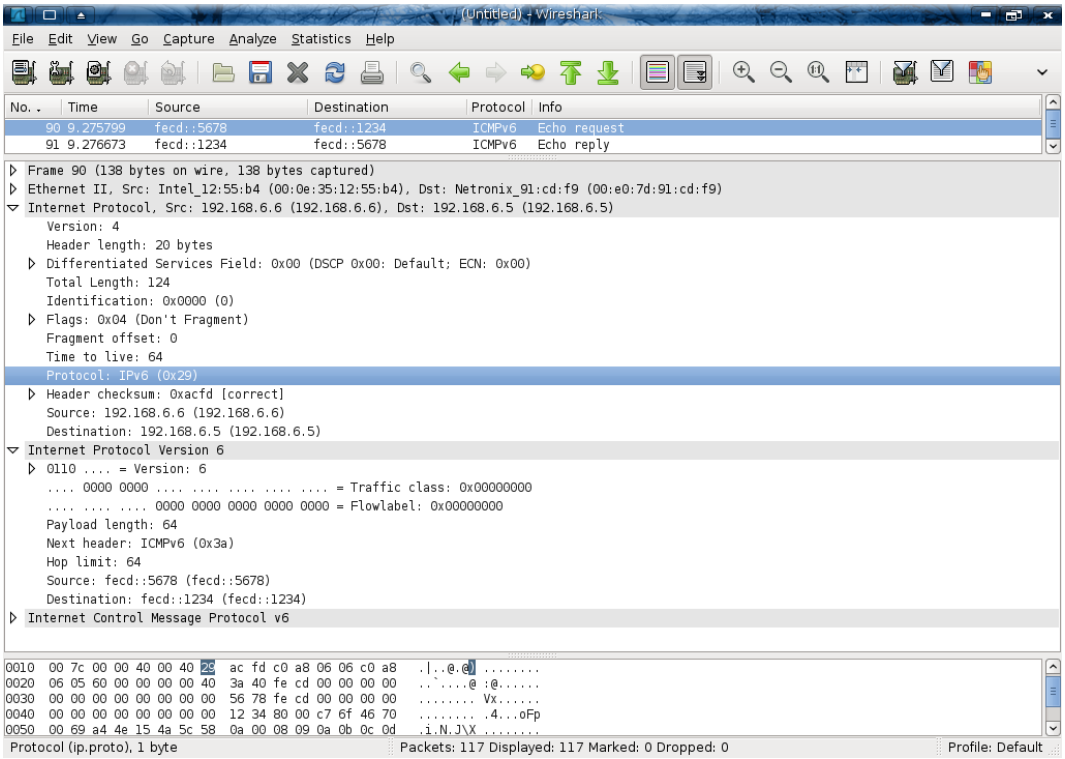
3.1.1. IPv4-kompatible Adressen

Ein halbautomatischer *proto-41* Tunnels ist durch Verwendung von IPv4-kompatiblen IPv6 Adressen. Hierbei wird grundsätzlich die IPv4 Adresse `aa.bb.cc.dd` eines Hosts bijektiv in eine IPv6 Adresse der Form `::aa.bb.cc.dd/96` abgebildet, d.h. in den niedrigsten 32 Bit ist die IPv4 Adresse eingebettet, der Rest ist mit 0 gefüllt; das Prefix

[1] IPv6 Tunnel Discovery: `http://www.dia.uniroma3.it/~compunet/tunneldiscovery`

[2] Renumbering: Ändert sich das IPv6 Prefix, müssen alle Hosts und DNS Einträge umkonfiguriert werden

Abbildung 3.3.: Mitschnitt einer proto-41 Kommunikation



hat dabei eine Länge von 96 Bit. Aus der IPv4 Adresse `192.168.6.6` kann so die IPv6 Adresse `::192.168.6.6/96` abgeleitet werden und andersherum.

Der IPv6 Stack schickt ein Paket an eine IPv4-kompatible IPv6 Adresse automatisch als IPv6-in-IPv4 Paket an die eingebettete IPv4 Adresse des Ziels. Weitere Konfiguration entfällt.

Vorteile Ein Tunnel ist sehr schnell eingerichtet, dynamisch und transparent. Jeder IPv4 Adresse wird eindeutig eine IPv6 Adresse zugeordnet. Die Technik skaliert, da keine Tunnel-Router zusätzlich belastet werden, sondern IPv6 Datenverkehr direkt über die vollständig bestehende IPv4 Infrastruktur geroutet wird.

Beim Tunnelmechanismus 6to4 wird als Default Router oft die IPv4-kompatible IPv6 Adresse `::192.88.99.1` konfiguriert, um IPv6 Pakete als *proto-41* über IPv4 an den Gateway senden zu können.

Nachteile Die Adressierungsart ist nach `rfc4291, Abschnitt 2.5.5`[1] bereits veraltet und braucht in neuen IPv6 Implementierungen nicht mehr unterstützt zu werden, da bessere Übergangslösungen zur Verfügung stehen.

Kommunikation ist nur möglich, wenn beide Hosts über IPv4-Kompatible Adressen verfügen und sich über IPv4 direkt erreichen können. Der Betrieb dieses Tunnels ist bei

Einsatz von NAT nur schwer möglich.

Es ist kein Routing über IPv6 vorgesehen. Da dadurch über diese Tunnelart keine neuen Nutzer erschlossen werden können, ist der Einsatz als Migrationsstrategie fraglich und vermutlich ist das ein Grund, warum die Tunnelart als veraltet gilt.

Einrichten unter Windows Unter Windows XP sind IPv4-kompatible Adressen standardmäßig deaktiviert, sollen aber mit folgendem Befehl aktiviert werden können:

```
1 netsh interface ipv6 set state v4compat=enable
```

Das Kommando hat in Tests allerdings keinen Effekt. Unter nachfolgenden Windowsversionen ist dieses Kommando nicht mehr vorhanden, sodass davon auszugehen ist, dass IPv4-kompatiblen Adressen von Windows *nicht mehr unterstützt* werden.

Einrichten unter Linux Die Konfiguration in Linux ist sehr einfach. Ist das sit-Modul geladen, sind für die Schnittstelle `sit0` bereits die entsprechenden Adressen konfiguriert und sofort einsatzbereit:

Listing 3.2: Verwendung IPv4-kompatibler IPv6 Adressen unter Linux

```
# modprobe sit
# ip link set sit0 up

# ip -6 addr show sit0
4: sit0: <NOARP,UP,LOWER_UP> mtu 1480
    inet6 ::192.168.6.6/96 scope global
       valid_lft forever preferred_lft forever
    inet6 ::127.0.0.1/96 scope host
       valid_lft forever preferred_lft forever

# ping6 ::192.168.6.5
PING ::192.168.6.5(::192.168.6.5) 56 data bytes
64 bytes from ::192.168.6.5: icmp_seq=1 ttl=64 time=0.955 ms
```

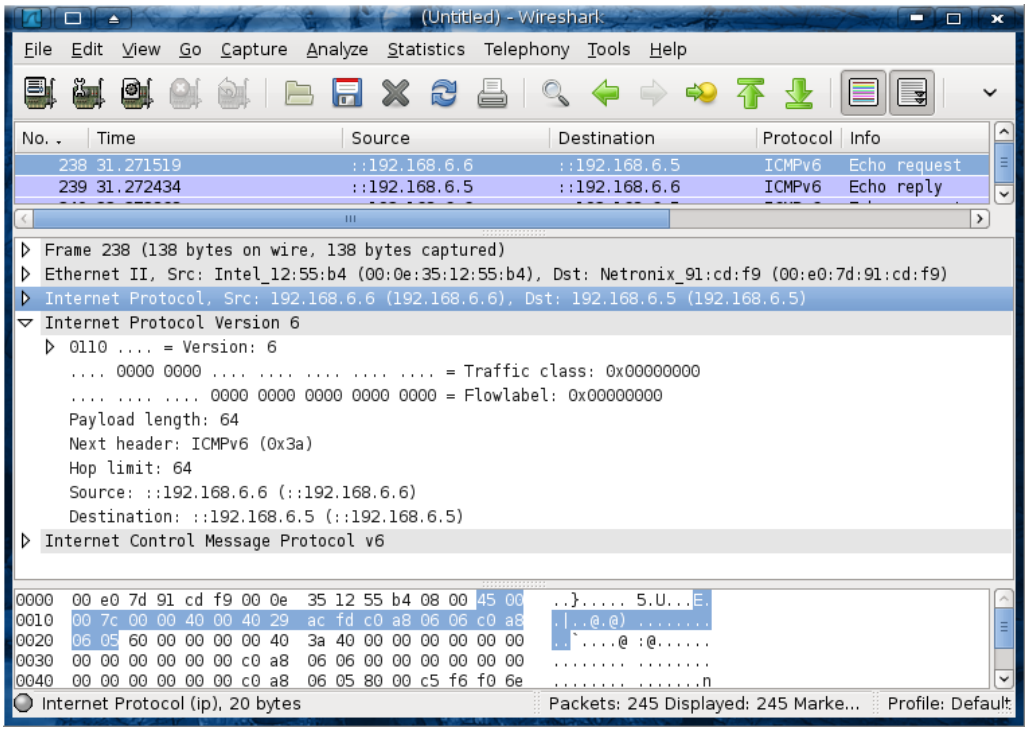
Zwar lassen sich jetzt an beliebige Hosts Pakete schicken, aber nur wer auch eine IPv4-kompatible IPv6 Adresse konfiguriert hat und direkt über IPv4 erreichbar ist, wird antworten.

IPv4-Mapped Adressen

IPv6-*kompatible* Adressen sind **nicht** zu verwechseln mit IPv4-*mapped* Adressen, die dazu verwendet werden, um IPv4 Kommunikation über die IPv6 API zu ermöglichen. Eine Anwendung kann so über den IPv6 Stack eines Hosts mit reinen IPv4 Hosts kommunizieren, ohne dass getrennt IPv4 und IPv6 API verwendet werden muss. Es werden weder native noch getunnelte IPv6 Datenpakete versendet, die Kommunikation außerhalb des Hosts ist rein IPv4. Dazu muss der Host natürlichüber eine IPv4 Adresse verfügen, es ist aber neben einem installiertem IPv6 Stack keine weitere Konfiguration notwendig, da die API diese Funktion transparent zur Verfügung stellt:

```
# ssh -6 ::ffff:127.0.0.1
```

Abbildung 3.4.: Mitschnitt von Kommunikation über IPv4-kompatible Adresse



So kann mit einem IPv6-fähigem Browser auch ohne konfigurierter IPv6 Adresse über die URL `http://[::ffff:d155:8168]/` die Hauptseite von www.google.de erreicht werden. Obwohl der Browser glaubt, über IPv6 zu kommunizieren, werden über das Netzwerk ausschließlich IPv4 Pakete an die eingebettete IPv4 Adresse `209.85.129.104` geschickt.

3.1.2. 6to4

Bei 6to4 Tunneln, beschrieben in `rfc3964` und `rfc3056`[3], handelt es sich ebenfalls um dynamische *proto-41* Tunnel, bei denen die Zieladressen der IPv4 Pakete aus den IPv6 Adressen abgeleitet werden. Hierfür ist der spezielle, global geroutete IPv6 Adressraum `2002::/16` vorgesehen. Jeder globalen IPv4 Adresse wird automatisch ein vollständiges Subnetz `2002:aabb:ccdd::/48` zugeordnet, wobei `aa.bb.cc.dd` die IPv4 Adresse des Tunnelendpunkts bezeichnet.

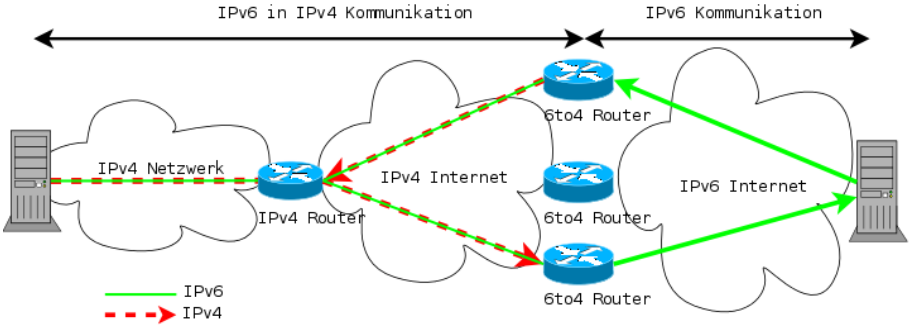
Aus der IPv4 Adresse `92.228.68.143` (=`0x5ce4448f`) wird z.B. die IPv6 Adresse `2002:5ce4:448f::1` gebildet, das `::/48` Subnetz kann intern weiter unterteilt und frei geroutet werden.

Ein Paket an eine 6to4 Adresse wird vom 6to4 Router in ein IPv4 Paket verpackt, welches an die IPv4 Adresse geschickt wird, die aus der 6to4 Adresse extrahiert wird. Der Router packt das IPv6 Paket aus und routet es native oder über weitere Tunnelmechanismen an den Empfänger weiter.

Wird ein Paket an eine native IPv6 Adresse geschickt, muss das IPv6 Paket über IPv4

in das native IPv6 Netz eingespeist werden. Hierfür werden sogenannte 6to4-Relays verwendet, die *proto-41* Pakete aller Art empfangen, auspacken und in das globale IPv6 Internet routen. Diese sind öffentlich und ohne Anmeldung nutzbar. Zur einfachen Adressierung des nächsten 6to4-Relays wurde die globale Anycast-Adresse[3] `192.88.99.1` reserviert. Ein natives IPv6 Paket verlässt das IPv6 Netzwerk über ein 6to4-Relay und

Abbildung 3.5.: 6to4-Tunnel Mechanismus



wird als IPv4 verpackt an den Tunnelendpunkt geschickt. Hier ist zu beachten, dass die Routen *asynchron* sein können da nicht zwangsweise dasselbe 6to4-Relay verwendet wird.

Listing 3.3: Einrichten von 6to4 unter Linux

```
# ip tunnel add tun6to4 mode sit remote any local 92.228.68.143
# ip link set dev tun6to4 up
# ip -6 addr add 2002:5ce4:448f::1/16 dev tun6to4
# ip -6 route add 2000::/3 via ::192.88.99.1 dev tun6to4
```

Das Prefix `::/16` der Tunnel Schnittstelle sorgt dafür, dass alle anderen 6to4 Hosts direkt über *proto-41* angesprochen werden, nicht über den Umweg eines Relays. Pakete an andere Adressen werden an die globale Anycast Adresse `::192.88.99.1` (oder auch `2002:c058:6301::`) geschickt. Zu beachten ist, dass Antwortpakete meist nicht *von* der Anycast Adresse kommen, sondern von der echten IP des zuständigen 6to4-Relays.

Auf dem Netzsegment des Tunnels wird kein Multicast unterstützt, da die IPv4 Empfängeradresse nicht aus der Multicast IPv6 Adresse (z.B. `ff02::2`) hergeleitet werden kann.

Sicherheit Durch die offene Architektur, muss ein 6to4 Router *proto-41* Pakete von *allen* Hosts entgegennehmen und verarbeiten. Dadurch können Pakete in den Tunnel injiziert werden, auch ohne IPv4 Adressen zu spoofen.

Durch asynchrone und nicht kontrollierbare Routen ist 6to4 nur wenig zuverlässig und Netzwerkprobleme sind nur sehr schwer zu debuggen.

[3] Anycast: Mehrere Hosts sind nach außen unter derselben IPv4 Adresse erreichbar; es ist Aufgabe des Routers, ein Paket sinnvoll zu einem dieser Hosts zu leiten, um die Last zu verteilen.

Vorteile Jeder IPv4 Adresse wird bereits eindeutig ein komplettes, geroutetes `::/48` Subnetz zugeordnet, welches ohne Registrierung sofort nutzbar ist. Dadurch eignet es sich vorzüglich als Migrationsstrategie für Privathaushalte, die Zugang zum IPv6-Internet haben möchten. Da eine globale IPv4 Adresse notwendig ist, funktioniert diese Technik nicht hinter NAT, aber auf dem Gateway und kann so ganze Haushalte über eine einzige IPv4 Adresse anbinden. Es ist mit zunehmender Unterstützung von 6to4 von Seiten der NAT-Router Hersteller zu rechnen (AVM[4] und Apple Airport Extreme[5] bieten bereits eine experimentelle Unterstützung).

Ein Tunnel kann leicht mit Boardmitteln erstellt werden und es ist keine Zusatzsoftware nötig, da kein gesondertes Protokoll gesprochen wird.

Die Technik wird von den meisten Betriebssystemen unterstützt und ist sehr einfach eingerichtet. Windows Vista und Windows 7 richten einen solchen Tunnel bereits automatisch ein, sobald eine globale IPv4 Adresse konfiguriert ist.

Nachteile Einzelne Computer hinter NAT können keinen Tunnel aufbauen, da eine globale IPv4 Adresse nötig ist.

Der Aufbau eines 6to4-Interfaces ist verbindungslos und völlig autonom, sodass Systeme eine 6to4-Schnittstelle konfigurieren können, ohne das eigentliche Konnektivität vorhanden ist. So richtet Windows Vista einen Tunnel auch dann ein, wenn die globale IPv4 Adresse von einer Firewall gefiltert wird, *proto-41* Pakete nicht ankommen und die IPv6 Verbindung dadurch unbrauchbar wird.

Da das verwendete Subnetz direkt von der IPv4 Adresse abhängt, wechselt im Fall einer neuen IPv4 Adresse auch das IPv6 Subnetz. Das ist gerade bei DSL Anbietern mit 24-Stunden-Zwangstrennung der Fall, was durch ständiges Renumbering den Einsatz von DNS für 6to4 Adressen erschwert. Für statische IPv4 Adressen besteht die Möglichkeit, eine Reverse-DNS Zone zu delegieren[6].

Anhand der IPv6 Adresse ist der verwendete Tunnelmechanismus und die IPv4 Adresse des Tunnelendpunktes erkennbar. Mangels von Zuverlässigkeit, Kontrolle des Routings und der verwendeten Gateways ist der produktive Einsatz der Technik in Unternehmen ungeeignet.

3.1.3. 6over4

In `rfc2529`[4] ist ein weiterer Ansatz beschrieben, um IPv6 über *proto-41* in einem Netzwerk zu betreiben. IPv6 setzt wieder direkt auf IPv4 als Transportschicht auf, benutzt aber auch dessen Multicast Fähigkeiten und benötigt deshalb keine dedizierten Tunnelserver oder Relays.

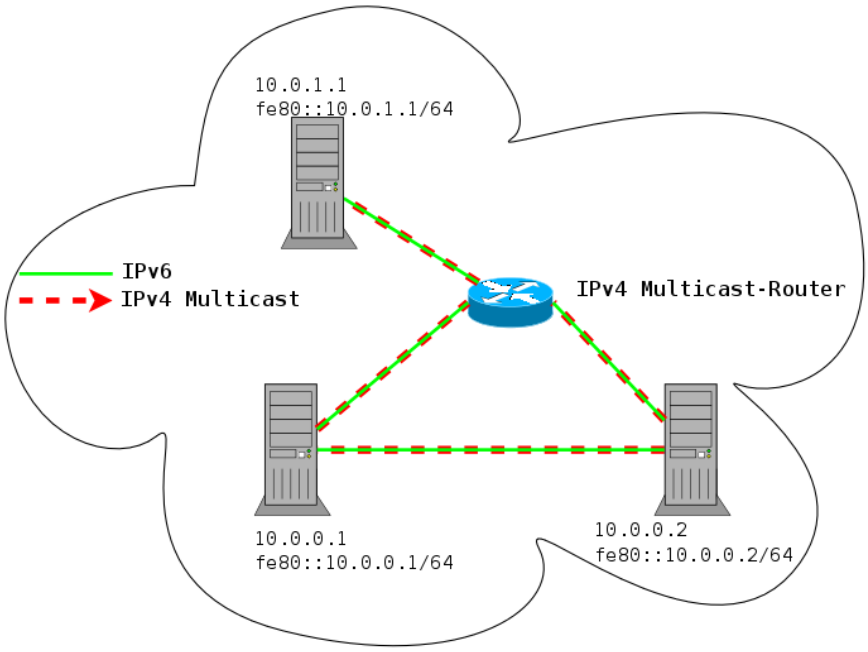
Die *linklocal*-Adresse eines *6over4*-Interfaces hat die Form `fe80::192.168.1.2/64`, d.h. die IPv4-Adresse des Hosts wird in den niederwertigen 32-Bit der IPv6-Adresse eingebettet.

[4] IPv6 über AVM Fritzbox: `http://www.avm.de/de/Service/Service-Portale/Service-Portal/Labor/7270_IPv6/labor_start_IPv6.php`

[5] IPv6 über Apple Airport Extreme: `http://www.apple.com/airportextreme/specs.html`

[6] Reverse DNS Delegation for 6to4: `http://6to4.nro.net/`

Abbildung 3.6.: 6over4 Tunnel Mechanismus über Multicast



Unicast Empfänger können dabei direkt erreicht werden, da sich die IPv4 Adresse aus der IPv6 Adresse ableiten lässt.

Multicast Empfänger werden über IPv4-Multicast erreicht. Dabei wird die IPv6 Multicastgruppe über eine spezielle Zuordnung in eine IPv4 Multicastgruppe gewandelt. Voraussetzung ist allerdings der Einsatz von multicastfähigen Netzkomponenten, weshalb *6over4* keine weite Verbreitung hat.

Durch den Einsatz von Multicast skaliert *6over4* annähernd wie natives IPv6. Zudem wird duch Multicast und linklokalen Adressen der transparente Einsatz von *Stateless Autoconf* möglich.

Vorteile Für einen Host fühlt sich *6over4* wie natives IPv6 an, da Router Solicitations und Neighbour Discovery über Multicast realisiert werden können. Es ist keine weitere Konfiguration eines Tunnelservers oder Tunnelclients nötig.

Sobald die Netzkomponenten IPv6 beherrschen, kann transparent die IPv4 Schicht entfallen und das Netz auf natives IPv6 umgestellt werden; es ist keine Umnummerierung notwendig.

Da der Schritt zu nativem IPv6 so klein ist, der Einsatz von IPv4-Multicast aber zusätzliche Komplexität bedeutet, ist der direkte Einsatz von nativem IPv6 in der Regel vorzuziehen.

Nachteile Die internen Router und Switche müssen IPv4-Multicast und ggf. auch Multicast-Routing beherrschen. Da kaum Implementierungen für gängige Betriebssysteme existieren, findet *6over4* kaum Beachtung. Die Technik bietet lediglich eine Transportschicht für IPv6 im Netzwerksegment, es ist immer noch ein global geroutetes IPv6 Prefix nötig, um Hosts vollständig an das IPv6 Internet anzuschließen.

3.1.4. ISATAP

Ähnlich *6over4* ist ISATAP (*Intra-Site Automatic Tunnel Addressing Protocol*), beschrieben in `rfc5214`[7], ein *proto-41* Tunnelmechanismus, um in einem reinen IPv4 Netzwerk mehrere Hosts über IPv6 zu verbinden. Ähnlich *6to4* wird hierbei eine spezielle IPv6-Adressierung verwendet, bei der die IPv4 Zieladresse in der IPv6 Adresse eingebettet wird, hier allerdings in den niederwertigsten 32 Bit. So kann leicht die Zieladresse des IPv4 Pakets aus der IPv6 Adresse abgeleitet werden. Das verwendete Subnetz muss ein Prefix der Größe `::/64` haben; eine gültige linklokale ISATAP Adresse wäre z.B. `fe80::5efe:10.0.0.1/64` (`fe80::5efe:a00:1/64`). Hierbei kann die IPv4 Adresse auch privat sein, es eignet sich deshalb für IPv4 Netze hinter einem NAT-Router. Wie bei *6to4* können sich Rechner im selben IPv6 Subnetz direkt UND ohne Einsatz eines Routers erreichen, indem das IPv6 Paket in ein entsprechendes IPv4 Paket verpackt wird.

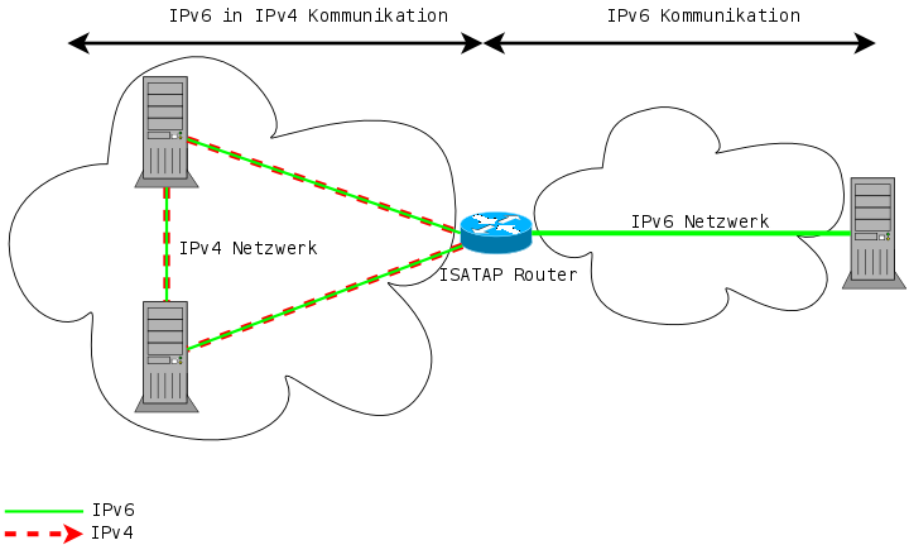
Pakete in ein anderes IPv6 Subnetz werden über einen ISATAP Router erreicht, wobei der ISATAP Router über IPv4 erreichbar und im selben IPv6 Subnetz sein muss. Da ISATAP kein Multicast beherrscht, kann *Stateless Autoconf* nur über Unicast erfolgen. Da ISATAP zudem verbindungslos ist und der Router keine Informationen über die Clients besitzt, müssen die Router vom Client gezielt abgefragt werden. Meist wird dabei der Domainname "ISATAP.DOMAIN" verwendet, um die verfügbaren ISATAP Router zu finden.

Der ISATAP Router muss über IPv4 als auch IPv6 Konnektivität verfügen, muss über IPv4 aber nicht von außen erreichbar sein. Es findet kein spezielles IPv6 Routing im Internet statt, wie bei *6to4*, da kein besonderes Prefix verwendet wird. Es kann sogar mit *6to4* kombiniert werden, da die Techniken für unterschiedliche Einsatzzwecke konzipiert wurden.

Autokonfiguration findet statt, indem der Client zuerst über DNS die IPv4 Adresse der möglichen ISATAP Router erfragt oder manuell mit diesen konfiguriert wird. Mit dem Wissen über die ISATAP Router, kann der Host IPv6 *Router Solicitation* Nachrichten als *proto-41* (Unicast) an die IPv4 Adressen des Routers senden. Dieser antwortet ebenfalls über Unicast und der Client erhält das Prefix des Netzes. Da kein Multicast verwendet wird, kann der ISATAP Router keine automatischen *Router Advertisements* senden sondern kann nur *Router Solicitations* beantworten.

Aus dem erhaltenen globalen Prefix des Subnetzes wird im Anschluss nach obigem Schema die globale IPv6 Adresse des Hosts gebildet. Der Host muss periodisch *Router Solicitation* Nachrichten an die ISATAP Router senden um die Gültigkeit der Routen und die IPv6 Adresse aufzufrischen.

Abbildung 3.7.: ISATAP Tunnel Mechanismus



Sicherheit Bei allen *proto-41* Tunneln kann die Quelladresse des IPv4 Pakets gefälscht und so jede Art IPv6 Paket in das Netzwerk eingeschleust werden. Da es sich bei ISATAP aber um ein lokales (*Intra-Site*) Protokoll handelt, ist es wesentlich einfacher, diese Attacken einzudämmen, insbesondere da der ISATAP Router nicht von außen über IPv4 erreichbar sein muss und sollte.

Hosts brauchen subnetzfremden Datenverkehr nur von ISATAP Routern zu akzeptieren und bei subnetzinternen Datenverkehr sollte die IPv6 mit der IPv4 Adresse verglichen werden um Spoofing-Angriffe zu erschweren. Insgesamt lässt sich Adressspoofing aber nicht verhindern.

Aufgrund des besonderen Adressformats bei ISATAP ist der Einsatz der *Privacy Extensions*[7] nicht möglich, allerdings ist die Identifikation eines IPv6 Teilnehmers für Administratoren dadurch wesentlich einfacher, da die IPv6 Adresse nicht mehr völlig frei gewählt werden kann, sondern direkt von der konfigurierten IPv4 Adresse abhängt.

Da die Größe des IPv4 Subnetzes nicht erkannt werden kann, kann der ISATAP Router nicht entscheiden, welche IPv4 Adressen *intern* und welche *extern* sind. über Ingress- und Egressfilter sollte stets verhindert werden, dass Subnetzeigener Datenverkehr über *proto-41* nach außen geroutet wird und sich so fremde Teilnehmer als Teil des ISATAP Netzes ausgeben können.

Betriebssystemunterstützung findet sich vor allem in Microsoft Windows und Ciscos-IOS, aber auch Linux lässt sich ab Kernel 2.6.25 in einem ISATAP Netz ansprechen. Für FreeBSD und MacOSX ist keine offizielle Unterstützung verfügbar.

[7] Bei eingeschalteten *privacy extensions* wird der Hostanteil der IPv6-Adresse, statt im EUI-64 Format, auf völlig zufällige Werte gesetzt, um die Identität und insbesondere die MAC Adresse des Hosts zu verschleiern.

Die nachfolgenden Kapitel werden sich ausführlicher mit ISATAP als Migrationsstrategie beschäftigen.

Vorteile ISATAP eignet sich, um in einem internen IPv4 Netzwerk alle nicht-IPv6-fähigen Geräte zu überbrücken und beliebige Hosts mit einem IPv6 Netz zu verbinden. An das IPv4 Netz werden keine weiteren Anforderungen gestellt. Der Protokolloverhead ist gering und Verbindungsprobleme sind leicht zu debuggen. Bei netzinternem Datenverkehr werden Router nicht zusätzlich belastet.

Eingehender und ausgehender Verkehr kann bequem auf dem ISATAP Router oder dem IPv4 Gateway gefiltert werden. Netzinterner Verkehr kann rudimentär über IPv4-Firewalls oder unterschiedliche ISATAP Domämen gefiltert werden. Alle Komponenten einer ISATAP Domäne befinden sich üblicherweise unter der Kontrolle des Betreibers.

Ein ISATAP Router ist mit sehr geringem Aufwand eingerichtet. Das Netz kann später ohne Renumbering auf nativen IPv6 Betrieb umgestellt werden.

Nachteile ISATAP ist unter Linux derzeit nur mit Aufwand nutzbar. Es ist ein IPv6 Adressraum notwendig, um eine Verbindung zum IPv6 Internet herzustellen. Die Tunnelart und interne Netzstruktur wird nach außen anhand der IPv6 Adresse sichtbar.

3.2. Teredo

Die meisten *Stateful Package Inspection*-Systeme wie NAT-Router können nur eingeschränkt mit *proto-41*-Traffic umgehen, weshalb sich die vorigen Tunneltechniken nicht für den Heimeinsatz eignen, wenn der Router nicht ausreichend konfiguriert werden kann.

Bei Teredo (`rfc4380`[9]) werden IPv6 Pakete in UDP eingepackt und über IPv4 verschickt, die *stateful* durch NAT-Router weitergeleitet werden. Damit muss der Teredo-Client nicht über eine öffentliche IPv4 Adresse verfügen, aber es handelt sich um eine Client-Server Architektur mit einem komplexeren Protokoll und erheblichen Overhead. Der Einsatz von Heartbeats (hier *Teredo-Bubbles* genannt) zum Teredo-Server und den Teredo-Relays ist notwendig, um die NAT-Tabelle des Routers aufzufrischen.

Verbindungsaufbau Der Verbindungsaufbau geschieht über eine Clientsoftware, die eine Verbindung zu einem bekannten Teredo-Server aufbaut und ein in `rfc4380` beschriebenes Protokoll spricht.

Eine Teredo Adresse wird folgendermaßen zusammengesetzt:

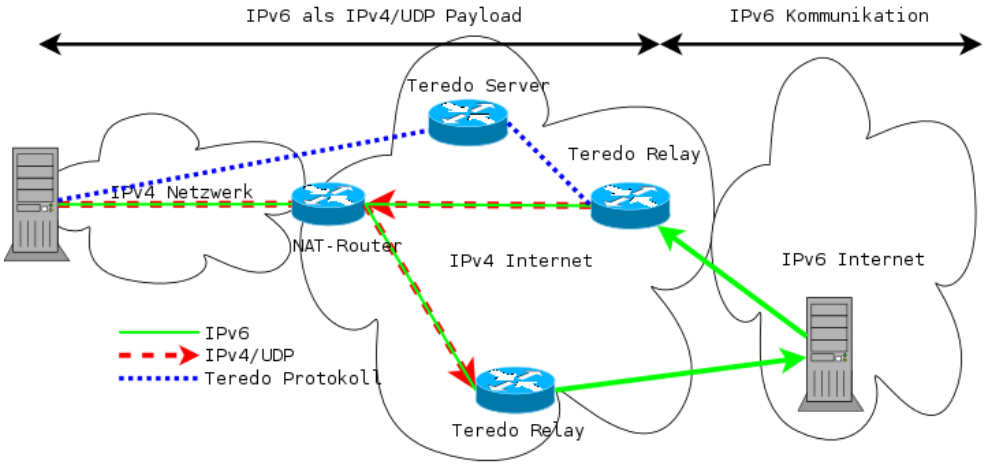
`Bits 00-31` Das spezielle Teredo Prefix `2001:0000`

`Bits 32-63` IPv4 Adresse des genutzten Teredo Servers

`Bits 64-79` Flags, die z.B. die Art des NAT kennzeichnen

`Bits 80-95` UDP Port der Verbindung vom NAT Gerät zum Server mit allen Bits invertiert.

Abbildung 3.8.: Teredo Tunnel über NAT



`Bits 96-127` IPv4 Adresse des NAT Geräts mit allen Bits invertiert.

So bezeichnet die IPv6 Adresse `2001:0000:0B0C:0D0E:8000:ABCD:69A3:5370` einen Teredo Client, dessen Adresse folgende Informationen enthält:

- Teredo Server `11.12.13.14` (`0x0B0C0D0E` Hex)
- Flags `0x8000=0100000000000000b`
- UDP Port `~0xABCD` = `0x5432` = 21554
- Öffentliche IPv4 Adresse des NAT `~0x69A35370=0x965CAC8F`=150.92.172.143

Der Teredo-Traffic wird über Teredo-Relays gesendet und empfangen. Bei einkommendem Datenverkehr ist wichtig, dass die Pakete vom Relay zum Client das NAT passieren können. Dazu werden eine Reihe Pakete verschickt, um von innen Löcher in den NAT Router zu bohren:

- Das Teredo-Relay sendet ein *Teredo-Bubble* an den Teredo-Server, der aus der IPv6 Adresse abgeleitet werden kann.
- Der Teredo-Server schickt ein *Teredo-Bubble* mit Informationen über das Teredo-Relay an den Client. Die Verbindung zwischen Teredo-Server und Teredo-Client ist dauerhaft vom Client über *Teredo-Bubbles* offengehalten.
- Der Teredo-Client schickt ein *Teredo-Bubble* an den anfragenden Teredo-Relay.
- Jetzt ist der Rückkanal vom Teredo-Relay zum Teredo-Client im NAT Router geöffnet und es können bidirektional Daten gesendet werden.

Eignung Teredo wird von gängigen Betriebssystemen unterstützt und findet weite Verbreitung. Für Linux und BSD existiert mit Miredo[8] eine freie Implementierung. Windows baut bei aktiviertem IPv6 automatisch eine Teredo-Verbindung zu einem bekannten Teredo-Server auf, solange keine native IPv6 Verbindung vorhanden ist (sogenannte *Sunset*-Prozedur[9]).

Teredo stellt jeweils nur eine einzelne IPv6 Verbindung zur Verfügung und eignet sich nicht, um ganze Netze aufzubauen oder zu verbinden.

Vorteile Teredo funktioniert auch hinter einem NAT-Router ohne spezielle Weiterleitung von Datenpaketen und es ist keine Registrierung für den Betrieb notwendig. Es eignet sich, um einzelne automatisch Rechner ans IPv6 Internet anzuschließen und wird von Windows von Haus aus unterstützt; unter Linux und BSD ist spezielle, frei verfügbare Client-Software nötig.

Nachteile Eine Unternehmensfirewall kann den IPv6-Verkehr, der über Teredo entsteht, nur schwer filtern. Durch übliche *stateful* Firewalls oder NAT werden zahlreiche Löcher gebohrt. Teredo-Adressen sind sehr dynamisch und es können keine Netze geroutet werden. Bei wechselnder IPv4 Adresse oder wechselndem UDP Port ändert sich stets auch die gebildete IPv6 Adresse. Das Protokoll ist relativ komplex und Verbindungsschwierigkeiten mit NAT sind schwer zu debuggen, da einkommender Datenverkehr über unterschiedliche Teredo-Relays geroutet werden kann.

Die Art des Tunnels, die IPv4 Adresse des NAT-Routers und die UDP Ports sind anhand der IPv6 Adresse öffentlich erkennbar.

3.3. Tunnelbroker

Eine sehr zuverlässige Art, eine IPv6 Verbindung ohne native Konnektivität zu erhalten, sind sogenannte Tunnelbroker. Sie routen mit teils proprietären Protokollen echte IPv6 Adressen und Subnetze über IPv4 zum Client. Ein eigenes Protokoll sorgt für den Verbindungsaufbau und Authentifizierung. Oft wird als Protokoll AYIYA[10] eingesetzt, was Tunnel auch hinter NAT Geräten und Firewalls erlaubt. Heartbeats sorgen dafür, dass die Verbindung bestehen bleibt oder bei Abbruch neu aufgebaut wird. Oft ist eine Registrierung beim Anbieter notwendig, dafür erhält man ein statisches IPv6 Subnetz, das nicht von einer verwendeten IPv4 Adresse abhängig ist und sich so vorzüglich auch für Heimanwender eignet.

Da der Datenverkehr stets über den selben Tunnel Server geroutet wird, lassen sich Verbindungsfehler einfach debuggen und Firewallregeln erstellen. Bekanntester Tunnelbroker ist SixXS[11], der nach einer erfolgreichen Authentifizierung ein `::/48` Subnetz zur

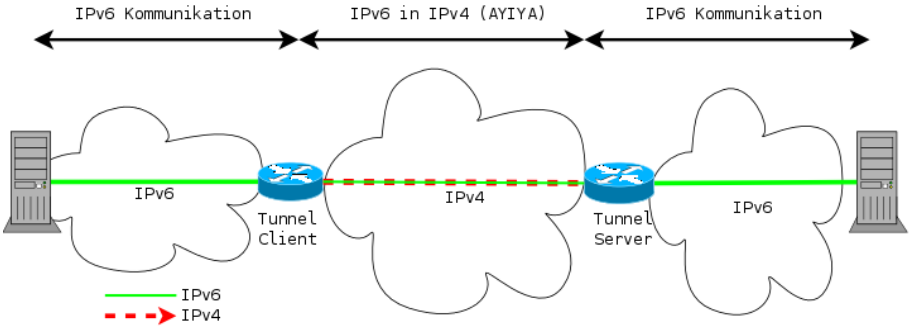
[8] Freie Teredo Implementierung *Miredo*: `http://www.remlab.net/miredo`

[9] Sunset Prozedur: Alle automatischen Tunnelmechanismen sollten sich automatisch selber deaktivieren, sobald natives IPv6 reif für den produktiven Betrieb ist

[10] Anything In Anything: `http://www.sixxs.net/tools/ayiya`

[11] SixXS: `http://www.sixxs.net`

Abbildung 3.9.: Tunnelbroker, z.B. SixXS



Verfügung stellt. Dieses kann intern wie natives IPv6 behandelt werden und ist nach außen nicht direkt als Tunnel erkennbar.

Weitere Tunnelbroker sind u.A. Hurricane Electric[12] und go6[13], die zum Teil eine Einwahl auch ohne vorherige Registrierung zulassen.

Vorteile Tunnelbroker überbrücken den ISP und ermöglichen es, einzelne Hosts oder ganze Netze mit dem IPv6 Internet zu verbinden. Das interne Netz kann eine native Anbindung an den Tunnel Client erhalten, oder es kann eine weitere Tunneltechnik eingesetzt werden.

Es findet übliches IPv6 Routing im Internet statt, da die Adresse eine normale IPv6 Adresse ist und nicht als Tunnel erkannt werden kann. Der Verbindungsaufbau funktioniert auch hinter NAT und man erhält meist komplette und statische Subnetze. Da Verkehr stets über dieselben Tunnelserver geroutet wird, sind Verbindungsprobleme sind leichter zu debuggen und die Verbindung ist allgemein stabiler, da weniger Komponenten beteiligt sind und der Tunnelserver geographisch günstig gewählt werden kann.

Selbst nach einer DSL Neueinwahl und damit wechselnder IPv4 Adresse bleiben IPv6-Verbindungen bestehen.

Es bieten sich dieselben Firewallmöglichkeiten wie bei nativem IPv6, was es auch für Unternehmen sehr attraktiv macht, deren ISP noch keine native IPv6 Konnektivität bereitstellt.

Nachteile Meist ist eine Registrierung und *Eignungsprüfung* nötig. Spezielle Einwahlsoftware ist nötig, um den Tunnel aufzubauen und zu überwachen (aiccu bei SixXS, freenet6/tspc bei go6). Diese ist meist für gängige, aber nicht immer alle Betriebssysteme verfügbar.

Manchmal liegen die Tunnelserver geographisch weit entfernt, was die Latenzen stark erhöht. So hatte go6 bis vor kurzem lediglich einen einzigen Tunnelserver in Montreal/-Kanada, was den Dienst für Europäische Nutzer wegen sehr geringem Datendurchsatz

[12] Hurricane Electric: `http://tunnelbroker.net`
[13] go6: `http://www.go6.net`

unbrauchbar machte. Auch ist der Datendurchsatz abhängig vom Durchsatz aller Nutzer des jeweiligen Tunnelservers.

Da das erhaltene Subnetz dem Tunnelbroker zugeordnet ist, ist ein Renumbering des Netzwerkes nötig, sobald IPv6 nativ verfügbar wird. Bis dahin ist das verwendete IPv6 Subnetz allerdings unabhängig vom IPv4 Internet Provider.

Teil II.

Analyse von ISATAP

4. Definition des Protokolls

4.1. RFC 4214/5214

Der erste Entwurf von ISATAP stammt von März 2001 und ist unter draft-ietf-ngtrans-isatap-00[5] abrufbar. Nach Version 24 des Entwurfs im Januar 2005 folgte die erste offizielle Veröffentlichung von `rfc4214`[6] im Oktober 2005. An der Entwicklung von ISATAP waren maßgeblich Cisco, Microsoft und Fred L. Templin (Boeing) beteiligt, seit März 2008 liegt in `rfc5214`[7] eine überarbeitete und stabile Beschreibung von ISATAP vor, die den experimentellen Status verlassen hat. Grundlage dieser Analyse ist ausschließlich diese überarbeitete Version.

4.1.1. Adressformat

Das Adressformat von ISATAP wird in `rfc5214, Abschnitt 6.1`[7] beschrieben. Ein ISATAP Subnetz hat stets ein Prefix von `::/64`, die niederwertigen 64 Bit der Adresse sind wie folgt definiert:

```
|0              15|16             31|32                               63|
+----------------+----------------+--------------------------------+
|000000ug00000000|0101111011111110|mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm|
+----------------+----------------+--------------------------------+
```

Bit 6 ist das "u"-Bit (*universal/local*) und soll auf 1 gesetzt sein, wenn die IPv4 Adresse global eindeutig ist, auf 0, wenn die eingebettete IPv4 Adresse privat[1] ist. Das "g" Bit (*individual/group*) zur Kennzeichnung von Unicast/Multicast wird bei ISATAP überlicherweise stets auf 0 gesetzt. Die niederwertigsten 32 Bit stellen die IPv4 Adresse des Hosts dar (Big-Endian).

Zwar ist das "u"-Bit bei privaten oder öffentlichen Adressen entsprechend zu setzen, `rfc5214` sieht aber von einer strengen Überprüfung der Korrektheit dieses Bits ab. Leider ist dadurch die Zuordnung nicht mehr bijektiv, da für jede IPv4 Adressen mindestens zwei gültige ISATAP IPv6 Adressen existieren. Mit einer gegebenen IPv4 Adresse kann somit nicht auch auf die IPv6 Adresse geschlossen werden, wenn sich eine Implementierung nicht an die Empfehlung hält. Folgende zwei Adressen sind zwar beide korrekt, aber nur die Erste ist vollständig `rfc5214`-konform:

- `fe80::5efe:10.0.0.1`
- `fe80::200:5efe:10.0.0.1`

[1] Eine Liste mit privaten IPv4 Adressblöcken findet sich in `rfc3330, Abschnitt 3`[8]

Da die IPv4 Adresse in dem Beispiel privat ist, sollte das "u"-Bit auf 0 gesetzt werden. `rfc5214` beschreibt keine weitere Verwendung für das "u"-Bit, im Kapitel 4.2.6 wird jedoch ein Verfahren beschrieben, das "u"-Bit zu verwenden um mögliches Kreisrouting zu verhindern.

4.1.2. Automatische Konfiguration von ISATAP Schnittstellen

Jeder IPv4 Adresse eines Hosts wird bereits automatisch eine linklokale ISATAP IPv6 Adresse zugeordnet, die automatisch eingerichtet werden kann, ohne dass eine weitere Konfiguration notwendig ist.

Über die vorgesehenen ISATAP linklocal Adressen kann bereits kommuniziert werden, wobei IPv6 als Transport- und IPv4 als Verbindungsschicht benutzt wird.

Duplicate Address Detection Normalerweise muss für jede neu konfigurierte IPv6 Adresse ein *Duplicate Address Detection* Mechanismus ausgeführt werden, bevor sie genutzt werden darf (`rfc4862, Abschnitt 5.4`[11]). Da man bei ISATAP allerdings davon ausgehen kann, dass die IPv4 Adresse im Netz und damit die speziell zugewiesene ISATAP IPv6 Adresse bereits eindeutig sind, ist der DAD-Algorithmus nicht mehr notwendig.

Neighbour Discovery Auch Neighbour Discovery entfällt, da die Zieladresse des IPv4 Pakets bereits aus der IPv6 Adresse abgeleitet werden kann. Natürlich muss für das IPv4 Paket ARP ausgeführt werden, bevor es über Ethernet versendet werden kann. *Neighbour Discovery über IPv6* ist in `rfc4861`[19] definiert.

Neighbour Unreachability Detection wird empfohlen, ist aber nicht notwendig, da die darunterliegende Schicht mit ARP bereits einen vergleichbaren Mechanismus besitzt.

4.1.3. Stateless Autoconf

Multicast `rfc5214` beschreibt ausdrücklich nicht die Verwendung von Multicast über ISATAP, da nicht davon ausgegangen werden kann, dass die darunterliegende IPv4 Schicht Multicastfähig ist. Dabei wird nicht klar, wie weit Multicast Adressen benutzt werden dürfen oder ob die Technik in einer folgenden rfc beschrieben wird. Es ist damit zu rechnen, dass sich in diesem Punkt verschiedenen Implementierungen unterscheiden werden.

Es ist nicht spezifiziert, wie Broadcast- oder Multicastziele wie `ff02::1` (allnodes) oder `ff02::2` (allrouters) über ISATAP zu erreichen sind, daher sollte davon ausgegangen werden, dass diese nicht nicht verfügbar sind.

Unicast Das Fehlen von Multicast bei ISATAP führt dazu, dass Router keine automatischen *IPv6 Router Advertisement*-Nachrichten senden können, da das Ziel Unicast sein muss und dem Router noch nicht bekannt ist. Daher liegt die Zuständigkeit beim Client,

der periodisch den Router mit *IPv6 Router Solicitation*-Nachrichten abfragen muss; dieser kann auf die Anfrage nun gezielt mit Unicast antworten, da der Absender bekannt ist. Clients müssen demnach mit Informationen über potentielle Router konfiguriert werden, damit das IPv4-Ziel der *Router Solicitation* bekannt ist.

Über die IPv6 Zieladresse des *ICMPv6 Router Solicitation*-Pakets wird keine Aussage gemacht, sie kann Unicast (z.B. `fe80::5efe:10.0.0.1`) oder Multicast (`ff02::2`) sein. Auch bei einer IPv6 Multicastadresse als Zieladresse muss das IPv4 Paket Unicast sein, wobei *Router Solicitation*-Nachricht lediglich vom einen Router empfangen wird (kein echtes Multicast). Wird eine IPv6 Unicast Zieladresse verwendet, muss diese bekannt sein und aus der IPv4 Adresse des Routers "geraten"werden. Da Multicast in `rfc5214` nicht sauber beschrieben wird, sollte Stateless Autoconf stets an die Unicast Adresse gerichtet sein.

ISATAP Link Der Client konfiguriert für das von einem ISATAP Router erhaltene Prefix eine globale IPv6 Adresse im ISATAP Adressformat. Als ISATAP Link wird im Folgenden das Netzsegment bezeichnet, in dem direkter *proto-41* Netzverkehr zwischen ISATAP Teilnehmer gesendet und empfangen werden kann (Layer 2, hier IPv4), sowohl über globale als auch linklokale IPv6 Adressen. Datenverkehr verlässt und betritt den ISATAP Link ausschließlich über die konfigurierten ISATAP Router.

Ein ISATAP Link ist dabei nicht notwendigerweise nur auf ein Teilnetz von IPv4 beschränkt, sondern kann sich mit globaler Gültigkeit auch über den gesamten IPv4 Adressraum erstrecken. Dabei verwenden alle Benutzer des ISATAP Links dasselbe IPv6 Prefix, gleichzeitig kann ein Host Mitglied mehrerer ISATAP Links gleichzeitig sein.

Privacy Extensions Bei *Stateless Autoconf* wird üblicherweise das EUI-64 Format verwendet, bei dem über eine bestimmte Zuordnung die MAC Adresse in die niederwertigen 64 Bit der IPv6 Adresse eingebettet wird. Die resultierende IPv6 Adresse ist eindeutig, da die MAC Adresse bereits eindeutig ist, dadurch wird der Host aber auch eindeutig identifizierbar, selbst wenn er das Netz wechselt. Das ist besonders bei mobilen Geräten ein Problem, da damit Nutzer über Subnetze hinweg verfolgt werden können.

Abhilfe schaffen die in `rfc4941`[10] beschriebenen *Privacy Extensions* von IPv6, indem bei *Stateless Autoconf* statt der MAC Adresse die niederwertigen 64 Bit der IPv6 Adresse bei rein zufällig gewählt werden.

Durch die spezielle Bedeutung der niederwertigen 64 Bit der IPv6 Adresse bei ISATAP sind aber weder das EUI-64 Format noch Privacy Extensions möglich und daher deaktiviert. Ein ISATAP Host verwendet immer die gleiche IPv6 Adresse, abhängig von der verwendeten IPv4 Adresse.

4.1.4. Potential Router List

Potential Router List Ein Host wird mit einer *Potential Router List* (*PRL*) konfiguriert. Diese Liste enthält die IPv4 Adressen von potentiellen Routern und dient insbesondere zwei Zwecken:

- Subnetzfremder Datenverkehr wird nur von IPv4 Adressen angenommen, die in der PRL aufgeführt sind.
- Der Host sendet periodisch *Router Solicitation* Nachrichten als *proto-41* Pakete an mögliche Router, um *Router Advertisement* Nachrichten zu provozieren, die Informationen über das globale Prefix des Subnetzes und Routen enthält.

  `rfc5214, Abschnitt 8.3.3` empfiehlt zudem, *Router Advertisements* ebenfalls nur von konfigurierten Routern anzunehmen.

Die Einträge der PRL kann der Host automatisch über z.B. DNS beziehen oder auch statisch konfiguriert werden.

Für Redundanz und Lastverteilung kann ein DNS Name mehrere A-Records für die PRL enthalten. Das RFC sieht vor, dass jeder Eintrag der PRL mit einem Timer versehen wird, über den gesteuert wird, wann *Router Solicitations* für diesen Eintrag gesendet werden sollen. Der Abstand zweier *Router Solicitations* darf höchstens so groß sein, wie die kleinste TTL der DNS Einträge. Zudem sollte der Zeitabstand die Lifetime des Prefixes in dem *Router Advertisement* unterschreiten, damit die konfigurierten Adressen und Routen nicht ihre Gültigkeit verlieren. Nachfolgende *Router Solicitations* sollen nicht näher als 2 Minuten beieinander liegen.

Automatische Konfiguration der PRL Um die Initialisierung der PRL auf dem Client zu automatisieren, wird empfohlen die Router im DNS stets über isatap.DOMAIN.TLD bekannt zu machen. Implementierungen sollten diesen FQDN[2] auflösen und eine automatische Konfiguration vornehmen. Da zum Zeitpunkt der Host noch nicht über IPv6 verfügt, muss die Namensauflösung über IPv4, statisch (z.B. über */etc/hosts*) oder einem anderen Weg erfolgen. Im Gegenzug entfällt jegliche Konfiguration auf dem Client.

Es wird empfohlen, den DNS Namen alle 3600 Sekunden neu abzufragen und die PRL neu aufzubauen, für den Fall, dass sich die darunterliegende Netztopoligie ändert.

Abhängigkeit von höherer Netzwerkschicht Kritisch wird bei der automatischen Konfiguration von ISATAP die Abhängigkeit von DNS angesehen. Hier hängt eine niedrige Netzwerkschicht (ISATAP (Layer 2), IPv6 (Layer 3)) von einer höheren Netzwerkschicht ab (UDP (Layer 4), DNS (Layer 5)). Um zirkuläre Abhängigkeit zu vermeiden, darf die Namensauflösung *nicht* über IPv6 erfolgen.

Mehrere ISATAP Links gleichzeitig Da ein Host Mitglied mehrerer ISATAP Domänen gleichzeitig sein kann, sind Einträge der PRL nicht notwendigerweise der selben ISATAP Domäne zugehörig. ISATAP Router, die in der PRL konfiguriert sind, können unterschiedliche Informationen zu Prefix und Routen zurückliefern, weshalb der *Stateless Autoconf*-Mechanismus mit jedem Router einzeln durchgeführt werden muss und nicht beendet werden darf, sobald das erste gültige *Router Advertisement* von einem Router empfangen wurde.

[2] Der *Fully Qualified Domain Name* (FQDN) setzt sich aus dem Hostname und der Suchdomain zusammen, der der Host angehört und meist über DHCP bezogen oder statisch konfiguriert wird.

Sofern die Router über *proto-41* erreichbar sind, werden Clients bei Erfolg mit mehreren IPv6 Adressen und Routen konfiguriert, die sich unabhängig von einander nutzen lassen.

4.1.5. ICMPv4 und ICMPv6 Fehler

`rfc5214, Abschnitt 7.2`[7] empfiehlt, dass ISATAP Implementierungen auf ARP- und ICMPv4-Fehler reagieren sollten, um über einen fehlerhaften Pfad zu informieren. Hier ist man direkt auf die Nutzdaten des ICMPv4 Pakets angewiesen, um die Verbindung zu ermitteln, auf die sich die Nachricht bezieht. Viele Router senden in der Payload gemäß `rfc792, Seite 4`[12] nur den original IPv4-Header plus die ersten 8 Byte der Folgedaten.

Folgt dem IPv4 Header direkt ein UDP oder TCP Header, reichen diese Informationen aus, die Verbindung zu identifizieren. Im Falle von *proto-41*-Tunneln folgt aber nach dem IPv4 Header der 40 Byte große IPv6 Header. Die ersten 8 Byte des IPv6 Headers enthalten nicht die nötigen Informationen, um die entsprechende IPv6-Verbindung zu identifizieren.

Kapitel 4.2.7 enthält weitere Gedanken zur Sicherheit bei ICMPv4 und ICMPv6.

Erweiterung In `rfc1812, Abschnitt 4.3.2.3`[13] werden Tunnel mit berücksichtigt und empfohlen, so viele Informationen wie möglich als ICMPv4 Payload zu senden, ohne eine Grenze von 576 Byte zu überschreiten. In Tests halten sich bereits viele Implementationen, wie Linux und Windows, an diese Erweiterung.

Dennoch ist es für einen ISATAP Stack oft nicht möglich, zuverlässig auf ICMPv4 Fehler reagieren zu können und der IPv6 Stack wird nicht über entsprechende Fehler informiert. Paketverlust und lange Timeouts bei unerreichbaren Nachbarn sind die Folge.

Eine ISATAP Implementierung sollte im Fehlerfall eine ICMPv6 Nachricht an den Absender schicken, welche laut `rfc4443, Abschnitt 3.1`[14] so viele Informationen wie möglich enthalten sollte, ohne die minimale MTU bei IPv6 von 1280 Bytes zu überschreiten.

Kapitel 7.2.2 beschäftigt sich genauer mit der Implementierung unter Linux.

4.1.6. Absenderprüfung auf einem ISATAP Interface

Der Empfänger muss bei Datenempfang auf einer ISATAP Schnittstelle gemäß `rfc5214, Abschnitt 7.3`[7] die Quelladresse des IPv6 und IPv4 Headers auf Konsistenz und Gültigkeit prüfen. Ein empfangenes Paket auf einer ISATAP Schnittstelle ist gültig,

- wenn die IPv6 Adresse eine korrekte ISATAP Adresse nach `rfc5214` ist und die IPv4 Quelladresse des IP Headers mit den niederwertigsten 32 Bit der IPv6 Quelladresse übereinstimmt. Es sollten nur ISATAP Pakete angenommen werden, die aus demselben Subnetz stammen.

- wenn die Nachricht ein IPv6 *Router Advertisement* ist, es sich bei der Quelladresse um eine ISATAP Adresse handelt und die IPv4 Adresse des Absenders in der PRL aufgeführt ist.

  Es sollten keine *Router Advertisements* von normalen Hosts angenommen und verarbeitet werden.

- wenn das IPv4 Paket von einem konfigurierten ISATAP Router kommt. Datenverkehr von konfigurierten ISATAP Routern muss immer angenommen werden, da diese Subnetzfremden IPv6 Datenverkehr weiterleiten können.

4.2. Sicherheitsmaßnahmen

4.2.1. Wahl der Netztopologie

Ein ISATAP Netz ist ein IPv6 Subnetz der Größe `::/64`, unabhängig der verwendeten IPv4 Adressen, es umspannt also stets den kompletten IPv4 Adressraum. Während die IPv4 Topologie Firewalls und Router mit ACLs enthalten kann, sind alle Hosts einer ISATAP-Domäne untereinander direkt über IPv6 erreichbar, da sie sich im selben Subnetz befinden. Der Versuch, Zugriffskontrolle in einer ISATAP-Domäne auf Grund von IPv4 Filterregeln zu regeln, ist mühsam und nicht flexibel, da IPv4 Firewalls Schwierigkeiten haben, Pakete aufgrund der IPv6 Payload zu filtern.

Um viele Angriffsszenarien bereits ausschließen zu können, empfiehlt es sich, abgeschottete and sicherheitskritische IPv4 Netze nicht in dieselbe ISATAP Domäne einzubinden, sondern pro IPv4-Netz ein eigenes IPv6 Subnetz zu planen. Eine IPv4 Firewall trennt die Netze, indem allein der Fluss von *proto-41*-Verkehr reguliert wird. Zwischen den IPv6 Netzen kann eine reine IPv6 Firewall für die notwendige Zugriffskontrolle sorgen.

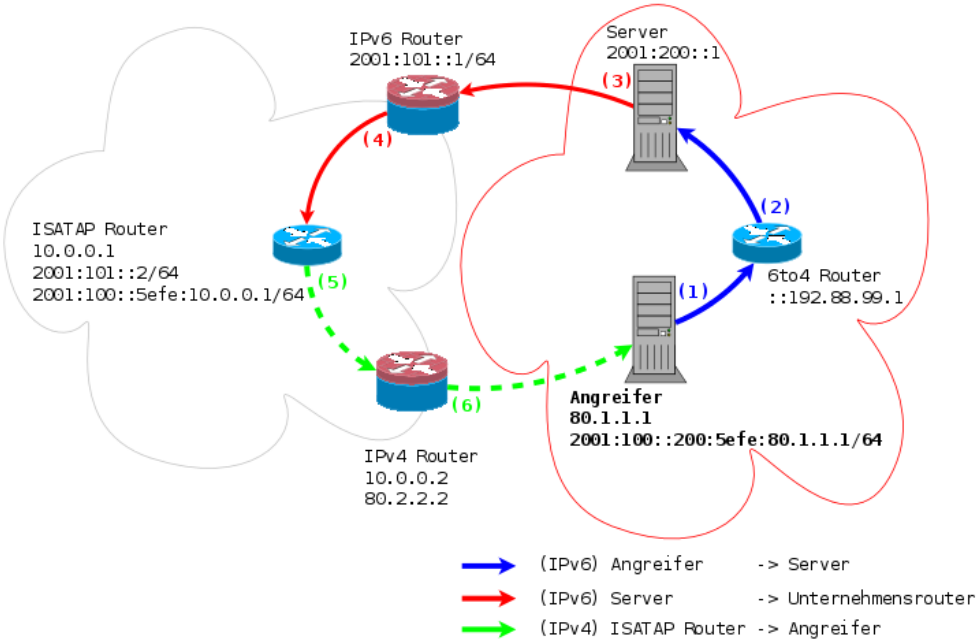
4.2.2. Strenge Trennung des ISATAP Links

Es reicht nicht aus, einen ISATAP Router lediglich vor Angriffen von außen zu schützen. Sobald ein ISATAP Router ungefiltert Pakete ins IPv4-Internet senden kann, besteht die Möglichkeit von Mißbrauch.

Werden ausgehende *proto-41* Pakete nicht geblockt, kann sich ein Angreifer über asynchrone Routen als Teil des ISATAP Netzes ausgeben und es wird folgende Kommunikation möglich (Abbildung 4.1):

1. Der Angreifer gibt sich eine gültige ISATAP Adresse in der seine globale IPv4 Adresse eingebettet ist; er muss dafür das verwendete Prefix des Netzes kennen. Direkte ISATAP Kommunikation mit dem ISATAP Router ist nicht möglich, wenn dieser durch eine IPv4 Firewall oder NAT nicht von außen erreichbar ist. Er kann IPv6 Pakete aber über einen beliebigen IPv6-Router in das globale IPv6 Netz einschleusen, z.B. über einen *6to4*-Router (`::192.88.99.1`).

Abbildung 4.1.: Adressklau bei ISATAP ohne Egressfilter



2. Sofern der (*6to4*-)Gateway die Absendeadresse nicht überprüft, sendet er das Paket weiter an die Zieladresse. Dieser kann die Herkunft des Pakets nicht mehr überprüfen.

3. Die Antwort des Servers wird entsprechend den IPv6 Routingtabellen zum ISATAP Netzwerk geroutet.

4. Nimmt der Router des Unternehmens netzeigenen Verkehr von außen an, wird das IPv6 Paket an den internen ISATAP Router weitergeroutet.
   - Wird eine statefull Firewall eingesetzt, werden die Antwortpakete verworfen, da sie nicht zu einer von innen initiierten Verbindung gehören.
   - Der Host kann von der Firewall als *extern* eingestuft werden.
   - Der Host kann als *intern* eingestuft werden und dabei vollen Zugriff auf das interne Netz erhalten. Zudem wird er von der Unternehmensfirewall vor Angriffen von "außen" geschützt wird.

5. Der ISATAP Router packt das IPv6 Paket in ein IPv4 Paket (*proto-41*) und schickt es an die in der Zieladresse eingebetteten IPv4 Adresse.

6. Der IPv4 Router oder eine Firewall des Unternehmens sollte das ausgehende *proto-41*-Paket verwerfen, da es sonst an den Angreifer geroutet wird.

Der Angreifer kann nun mit dem Server über eine IPv6 Adresse des Unternehmens kommunizieren. Zwar ist der Angreifer durch seine eingebettete IPv4 Adresse identifizierbar,

aber es könnten Adressbereichsfilter umgangen werden. Für den Angriff spielt es keine Rolle, ob für das ISATAP Netzwerk private (NAT) oder öffentliche IPv4 Adressen verwendet werden, solange der Angreifer vom ISATAP Router erreicht werden kann.

Unter Umständen ist auch direkte Kommunikation mit internen ISATAP Hosts des Unternehmens möglich, auch wenn der ISATAP Router nicht von außen erreichbar ist. Das kann passieren, wenn IPv6 Pakete ohne Prüfung des Absenders zum Client geroutet werden. Dieser glaubt dann dass sich der Absender auf demselben ISATAP Link befindet und sendet seine Antworten direkt über *proto-41* an den Absender zurück und nicht über den ISATAP Router. Die Unternehmensfirewall sollte ausgehende IPv4 Pakete grundsätzlich auch von Clients filtern.

Anfrage- und Antwortpaket nehmen asynchrone Pfade, was in der Praxis aber nicht weiter von Bedeutung ist.

Praxisbeispiel

Mit etwas Glück kann innerhalb weniger Minuten ein offenes ISATAP Netzwerk gefunden werden. Der ISATAP Router muss dazu weder öffentlich noch bekannt sein.

ISATAP Netz Über Google wurden die folgenden, weltweit offenen ISATAP Netze gefunden:

- `2001:638:500:11::/64`, JOIN, Universität Münster
  ISATAP Router `128.176.191.4` (isatap.uni-muenster.de)
  weltweit offen, beantwortet keine *Router Solicitations*

- `2001:4c50:0:fe00::/64`, Kabelfernsehen München ServiCenter
  ISATAP Router `85.233.58.56` (isatap.cablesurf.de)
  weltweit offen

- `2001:da8:8000:3::/64`, Shanghai Jiaotong University
  ISATAP Router `202.112.26.254`
  weltweit offen

- `2001:da8:200:900e::/64`, Tsinghua University, Beijing
  ISATAP Router `59.66.4.50` (isatap.tsinghua.edu.cn)
  weltweit offen

- `2001:250:4000:8fff::/64`, HuaZhong University of Science and Technology
  ISATAP Router `202.114.1.186`
  weltweit offen

Wie der Test zeigt, senden alle Router *proto-41*-Pakete ungefiltert ins Internet. Zudem nehmen die Router auch ISATAP Pakete von außen an, sodass sich der ISATAP Link über den ganzen Globus erstreckt und die Router weltweit offen sind. Ob das Absicht oder eine Fehlkonfiguration ist, lässt sich nicht erkennen; völlig anonym ist ein Angreifer nicht, da stets die verwendete IPv4 Adresse in die IPv6 Adresse eingebettet wird.

Reguläre Nutzung Alle oben gefundenen ISATAP Router haben eine öffentliche IPv4 Adresse und nehmen *proto-41*-Pakete von außen an, was sie zu öffentlichen ISATAP Routern macht. Bei den Tests fällt auf, dass die linklokale IPv6 Adresse der jeweiligen Routers das *u*-Bit nicht gesetzt hat, obwohl es sich um eine öffentliche IPv4 Adresse handelt. Vermutlich handelt es um ältere Cisco-IOS Router, die ein ähnliches Verhalten zeigen (siehe Kapitel 5.2). Eine Benutzung mit einem standardkonformen ISATAP Client (Windows Vista, Windows 7) schlägt daher fehl.

Windows XP:

```
> netsh interface ipv6 isatap set router isatap.cablesurf.de
```

Linux (Für Programm siehe Kapitel 8.2):

```
$ sudo ./isatapd -v isatap.cablesurf.de

isatapd: Adding internal PDR 85.233.58.56
isatapd: is0 created (local 78.51.67.231, pmtudisc)
isatapd: interface is0 up
isatapd: Soliciting fe80::200:5efe:55e9:3a38
isatapd: Soliciting fe80::5efe:55e9:3a38
isatapd: Advertisement from fe80::5efe:55e9:3a38
isatapd:   Prefix 2001:4c50:0:fe00::/64, lifetime 604800 sec
```

Konfiguration des Angreifers Der ISATAP Router braucht nicht öffentlich sein. Eine bestehende 6to4 Verbindung auf dem Client vorausgesetzt, reicht bereits ein einzelner Linux Befehl um eine ISATAP Adresse zu "stehlen":

Listing 4.1: Stehlen einer ISATAP Adresse über 6to4

```
------------- Konfiguration Angreifer ---------------
localhost:~# ip tunnel show
 [...]
 6to4: ipv6/ip  remote any  local 92.230.128.99  ttl 64

localhost:~# ip -6 route show
 [...]
 default via ::192.88.99.1 dev 6to4  metric 1200  mtu 1420 advmss 1360

localhost:~# ip -6 addr add 2001:da8:8000:3:0:5efe:92.230.128.99/128 \
    dev 6to4

------------- Test von externen Server ---------------

remotehost:~# traceroute6 2001:da8:8000:3:0:5efe:92.230.128.99
traceroute to 2001:da8:8000:3:0:5efe:5ce6:8063, 30 hops max
[...]
11  cernet-RGE.hkix.net (2001:7fa:0:1::ca28:a1be)  285.096 ms
12  2001:252:0:101::1 (2001:252:0:101::1)  297.117 ms
13  2001:252:0:1::1 (2001:252:0:1::1)  284.698 ms
14  tjn-bj-v6.cernet2.net (2001:da8:1:16::2)  286.943 ms
15  jnn-tjn-v6.cernet2.net (2001:da8:1:15::2)  290.747 ms
16  hefei-core-h-pos-01-v6.cernet2.net (2001:da8:1:24::2)  300.246 ms
17  nj-hef-v6.cernet2.net (2001:da8:1:13::1)  302.460 ms
18  sh-nj-v6.cernet2.net (2001:da8:1:4::2)  309.960 ms
```

```
19  2001:da8:1:205::2 (2001:da8:1:205::2)  310.646 ms
20  cernet2.net (2001:da8:a4:2::2)  308.927 ms
21  cernet2.net (2001:da8:8000:102::206)  308.591 ms
22  cernet2.net (2001:da8:8000:100::2)  308.888 ms
23  cernet2.net (2001:da8:8000:3:0:5efe:5ce6:8063)  445.075 ms
```

Das Ziel wird nach 23 IPv6 Hops erreicht. Zu beachten ist, dass zwischen Hop 22 und 23 das IPv4 Paket das ganze Internet vom ISATAP Router zur IPv4 Adresse des Angreifers durchquert. Das dargestellte Traceroute stellt jedoch nur einen Pfad dar. Antworte des Zielhosts werden über den 6to4-Gateway und haben mit ISATAP nichts mehr zu tun.

Die hier gestohlene IPv6-Adresse `2001:da8:8000:3:0:5efe:5ce6:8063` wird über DNS kurioserweise zu `cernet2.net` aufgelöst.

Der Angreifer erhält vom ISATAP Router *proto-41*-Pakete. Durch Betrachtung des einkommenden Netzwerkverkehrs erhält man die öffentliche IPv4 Addresse des ISATAP Routers (oder dessen NAT Routers):

```
localhost:~# iptables -I INPUT 1 -p 41 -j LOG
localhost:~# dmesg
[...]
IN=ppp0 OUT= MAC= SRC=202.112.26.254 DST=92.230.128.99 LEN=124 TOS=0x00
   PREC=0x00 TTL=235 ID=57886 PROTO=41
```

Als ISATAP Router wurde hier `202.112.26.254` identifiziert.

4.2.3. Maßnahmen auf Clients

Empfang von Nachbarn Zusätzlich zum IPv6 Stack bietet bei ISATAP Clients der IPv4 Stack zusätzliche Angriffsfläche. Die in `rfc5214` beschriebene, strenge Überprüfung der Absendeadresse bietet nicht immer Schutz vor Spoofing Attacken.

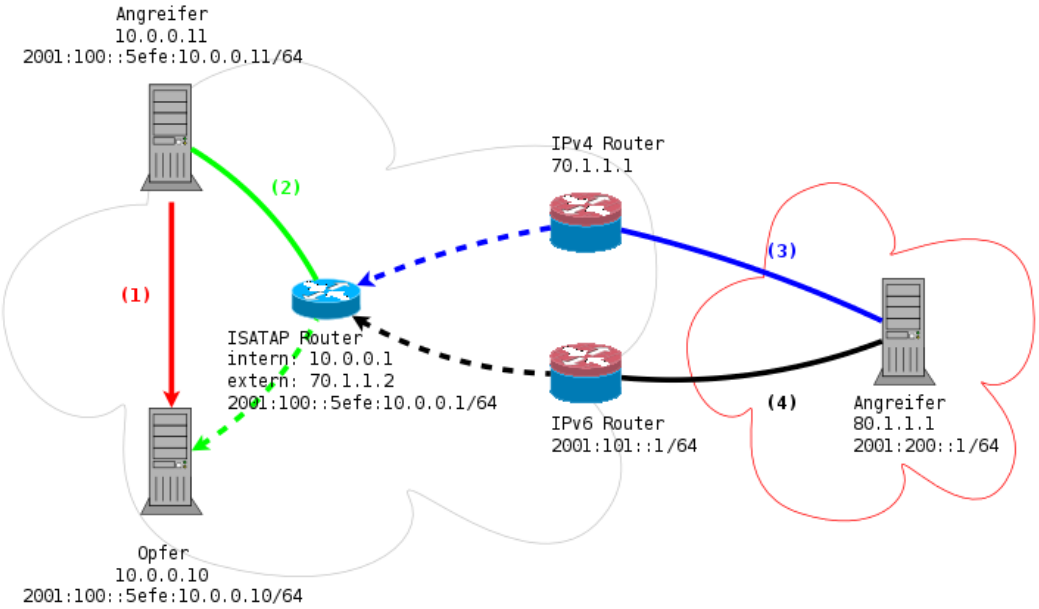
Zwar muss die Quelladresse des IPv4 Pakets mit den niederwertigen 32 Bit der IPv6 Quelladresse übereinstimmen, doch lassen sich beide leicht fälschen. Dabei muss es sich allerdings um einen Angriff innerhalb des ISATAP Links handeln und unterscheidet sich nicht von üblichen Attacken gegen IPv4. ISATAP ist weiterhin anfällig für alle Attacken gegen den IPv4 Stack, wie *ARP-Poisoning* oder *MAC-Flooding*.

Empfang von Routern Datenempfang von einem Router ist kritisch, da hier eine Vertrauensstellung herrscht und jede Art von IPv6 Daten angenommen und verarbeitet werden. Es ist empfehlenswert, eine IPv6 Firewall zu betreiben, um netzfremden und netzinternen Datenverkehr zu filtern. Der Einsatz einer IPv4 Firewall bringt bei ISATAP nur wenig zusätzliche Sicherheit.

Sind die Clients mit öffentlichen IPv4 Adressen konfiguriert, sollten *proto-41* weder das interne Netzwerk verlassen, noch von außen in das Netzwerk gelangen. Andernfalls werden Spoofing-Angriffe auch von außerhalb möglich.

Verschlüsselung Wird der IPv4 Datenstrom verschlüsselt, ist automatisch auch der IPv6 Datenstrom bis zum ISATAP Router verschlüsselt, verlässt das ISATAP Netz allerdings unverschlüsselt.

Abbildung 4.2.: ISATAP Spoofing Szenarien



Wird lediglich der IPv6 Datenstrom verschlüsselt (z.B. durch IPsec/VPN), können abgehörte IPv4 Datenpakete noch immer Informationen über die Kommunikationspartner preisgeben.

4.2.4. ISATAP Spoofingszenarien

Abbildung 4.2 illustriert mögliche Angriffsszenarien beim Einsatz von ISATAP.

Szenario 1 Ein interner Angreifer schickt ein Paket über *proto-41* direkt an das Opfer. Sofern sich zwischen Angreifer und Opfer keine weitere IPv4 Firewall befindet, muss das Opfer das Paket untersuchen.

```
src: 10.0.0.11, 2001:200::1/64
dst: 10.0.0.10, 2001:100::5efe:10.0.0.10/64
```

Das *proto-41*-Paket wird angenommen, aber das IPv6 Paket wird verworfen, da der IPv6-Sender keine netzinterne ISATAP Adresse ist und es sich um den IPv4-Sender nicht um einen vertrauenswürdigen Router handelt.

```
src: 10.0.0.11, 2001:100::5efe:10.0.0.20/64
dst: 10.0.0.10, 2001:100::5efe:10.0.0.10/64
```

Das Paket muss verworfen werden, da die eingebettete IPv4 Adresse nicht mit der IPv4 Adresse des Senders übereinstimmt.

```
src: 80.1.1.1,  2001:100::5efe:80.1.1.1/64
dst: 10.0.0.10, 2001:100::5efe:10.0.0.10/64
```

Das IPv6 Paket wird angenommen, da es sich um eine gültige ISATAP Adresse des Subnetzes handelt und anhand der IPv4 Adresse nicht die Größe des Subnetzes abgelesen werden kann. Eine Firewall auf dem Client könnte den Adressraum einschränken, für den *proto-41* Pakete angenommen werden und das Paket verwerfen. Das Antwortpaket würde vom Opfer über *proto-41* direkt an die Zieladresse gesendet. Hier tritt Szenario 3 ein, in dem *proto-41* Verkehr, der das Netzwerk verlässt, vom IPv4 Gateway gefiltert werden sollte.

```
src: 10.0.0.20, 2001:100::5efe:10.0.0.20/64
dst: 10.0.0.10, 2001:100::5efe:10.0.0.10/64
```

Das IPv6 Paket wird angenommen, da es sich um eine gültige ISATAP Adresse des lokalen Subnetzes handelt. Es kann nicht festgestellt werden, ob die IPv4 Adresse des Absenders existiert; der Angriff unterscheidet sich nicht von einer IPv4 Spoofing Attacke. Antwortpakete verlassen das Subnetz nicht, der Angriff würde sich nur für verbindungslose Protokolle (z.B. UDP) lohnen.

```
src: 10.0.0.1,  2001:200::1/64
dst: 10.0.0.10, 2001:100::5efe:10.0.0.10/64
```

Hier wird ein netzfremdes Paket direkt an das Opfer geschickt, und zudem die IPv4 Adresse auf die eines gültigen Routers gesetzt. Da die IPv4 Adresse in der PRL des Clients aufgeführt ist, wird das Paket angenommen, verarbeitet und die IPv6 Adresse nicht weiter überprüft.

Szenario 2 Da IPv4 Adressen von ISATAP Routern bei den Clients besonderes Vertrauen genießt, kann es sich für einen Angreifer lohnen, Datenverkehr über diesen einzuschleusen. Der ISATAP Router ist zudem anfällig für dieselben Möglichkeiten wie in Szenario 1.

```
src: 10.0.0.11, 2001:100::5efe:10.0.0.20/64
dst: 10.0.0.1,  2001:100::5efe:10.0.0.10/64
```

Das Paket sollte vom Router geprüft und verworfen werden, da Absenderadressen nicht übereinstimmen. Leitet der Router das Paket an das Opfer weiter, tritt der nachfolgende Fall ein:

```
src: 10.0.0.1,  2001:100::5efe:10.0.0.20/64
dst: 10.0.0.10, 2001:100::5efe:10.0.0.10/64
```

Das Paket sollte verworfen werden, auch wenn es von einem Router stammt, da netzinterne Pakete üblicherweise direkt gesendet werden und nicht über den Router. `rfc5214` geht auf diesen Fall nicht ein und die meisten Implementierungen nehmen solche Pakete an.

```
src: 10.0.0.11, 2001:200::1/64
dst: 10.0.0.1,  2001:100::5efe:10.0.0.10/64
```

Das Paket sollte vom Router verworfen werden, da netzfremder Datenverkehr von innen kommt und der Absender kein Router ist. Leitet der Router das Paket an das Opfer weiter, kann dieses das gefälschte Paket nicht von einem Echten unterscheiden und wird das Paket verarbeiten.

Szenario 3 Ist der ISATAP Router von außen über IPv4 erreichbar, bietet sich die Möglichkeit, diesen über *proto-41* anzugreifen.

```
src: 80.1.1.1, 2001:100::5efe:80.1.1.1/64
dst: 70.1.1.2, 2001:100::5efe:10.0.0.10/64
```

Grundsätzlich sollte *proto-41*-Verkehr zum ISATAP Router nur von innen zugelassen werden und von außen geblockt werden. Zusätzlich ist netzwerkinterner IPv6 Verkehr von außen abzulehnen, da das auf eine spoofing Attacke hindeutet.

Auch sollte *proto-41*-Verkehr vom ISATAP Router nicht nach außen gesendet werden, da sich so ein externer Angreifer als netzintern ausgeben kann (Siehe Kapitel 4.2.2 und Abbildung 4.1).

Szenario 4 Über IPv6 ergeben sich weniger Angriffsmöglichkeiten, da es nur noch die IPv6 Quelladresse zu fälschen gibt.

```
src: 2001:100::5efe:80.1.1.1/64
dst: 2001:100::5efe:10.0.0.10/64
```

Der Ingressfilter auf dem IPv6 Gateway sollte grundsätzlich jeden netzinternen Datenverkehr verwerfen, der von außen kommt.

```
src: 2001:200::2/64
dst: 2001:100::5efe:10.0.0.10/64
```

Das Paket wird vom IPv6 Gateway an den ISATAP Route weitergeleitet, da die Absenderadresse gültig erscheint und nicht weiter geprüft werden kann.

4.2.5. Maßnahmen auf Routern

Da dem ISATAP Router generell vertraut werden muss, sollte dieser das Risiko von Spoofing Angriffen (Abbildung 4.2) minimieren.

- Clients sollten netzfremden Datenverkehr nicht von benachbarten Clients annehmen (1)
- Der ISATAP Router sollte netzfremden Datenverkehr von innen ablehnen und bei Bedarf auch netzinternen Datenverkehr von innen nach innen ablehnen können (2)

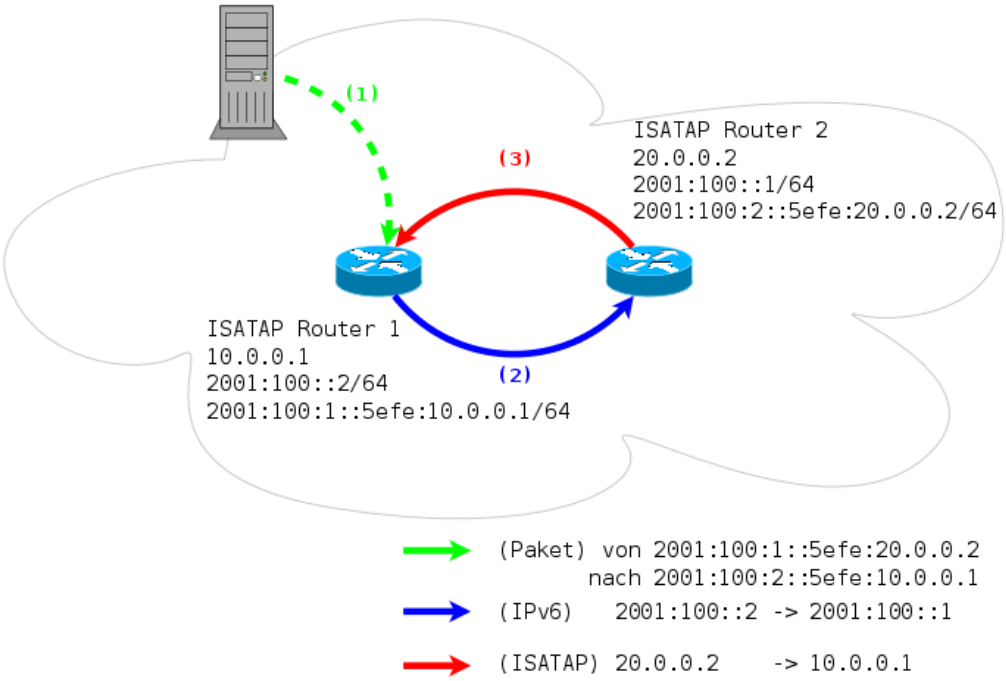
- Das IPv4 Gateway oder der ISATAP Router sollte proto-41 Datenverkehr von und nach außen streng filtern (3)
- Das IPv6 Gateway sollte netzinternen Datenverkehr von außen ablehnen (4)

4.2.6. Kreisrouting zwischen ISATAP Links

Die folgende Attacke wird neben ähnlichen Angriffen in einem Paper Routing Loop Attacks using IPv6 Tunnels *[15] beschrieben.*

Überschneiden sich zwei ISATAP Links, d.h. befinden sich in einem IPv4 Netz mehrere ISATAP Router für unterschiedliche IPv6 Subnetze, kann es zu unerwünschtem Kreisrouting und möglichen *Denial of Service*-Angriffen auf Router kommen (Abbildung 4.3).

Abbildung 4.3.: Kreisrouting bei ISATAP



1. Ursache ist ein eingeschleustes IPv6 Paket mit

```
src: 2001:100:1::5efe:20.0.0.2/64
dst: 2001:100:2::5efe:10.0.0.1/64
```

   Dabei wird die IPv4 Adresse von Router 2, der für das Prefix 2 zuständig ist, in eine ISATAP Adresse des Prefix 1 eingebettet und umgekehrt. Damit ist Router 1 Teil von Prefix 2 und Router 2 Teil von Prefix 1.

2. Nimmt Router 1 das Paket an, wird es über natives IPv6 an Router 2 geroutet, da dieser für das Subnetz `2001:100:2::/64` zuständig ist.

3. Router 2 nimmt das Paket an. Da die Zieladresse des Pakets Teil des eigenen ISATAP Netzes `2001:100:2::/64` ist, wird das Paket über ISATAP von `20.0.0.2` (Router 2) an die eingebettete IPv4 Adresse `10.0.0.1` (Router 1) weiter gesendet.

   Router 1 wird das ISATAP Paket annehmen, da die Quelladresse aus dem eigenen ISATAP Netz `2001:100:1::/64` stammt und die IPv4 Adresse der Quelle übereinstimmt. Router 1 wird das Paket erneut an Router 2 senden und es kommt zu Kreisrouting bis die TTL des IPv6 Pakets abläuft.

Für dieses Szenario darf *proto-41* zwischen Router 1 und Router 2 nicht gefiltert sein.

Filterung von proto-41 Eine mögliche Gegenmaßnahme ist es, auf den Routern *proto-41* nur von nicht-Routern anzunehmen. Da die beiden Router in diesem Beispiel stets über IPv6 miteinander kommunizieren werden und nie über ISATAP, können Firewallregeln ISATAP Verkehr zwischen den Routern unterbinden.

- *proto-41* Pakete von `10.0.0.1` (Router 1) an `20.0.0.2` (Router 2) verwerfen

- *proto-41* Pakete von `20.0.0.2` (Router 2) an `10.0.0.1` (Router 1) verwerfen

Diese Vorgehensweise ist aber nur bei kleinen Netzen möglich, bei denen alle möglichen ISATAP Router bekannt und unter Kontrolle sind, sodass sie voneinander abgeschottet werden können. Bei globalen ISATAP Links ist das nicht mehr möglich.

Filterung anhand des u-Bits Von Fred L. Templin stammt der Vorschlag, bei gesetztem *u*-Bit das IPv6 Prefix bei der Adressprüfung zu ignorieren. Ist das *u*-Bit gesetzt, ist die eingebettete IPv4 Adresse global eindeutig und kann eindeutig einem Prefix zugeordnet werden. Erhält Router ein ISATAP Paket, bei dem die eingebettete IPv4 Adresse mit der lokal konfigurierten IPv4 Adresse übereinstimmt und besitzt er eine globale IPv4 Adresse, ist das Paket für ihn bestimmt, unabhängig des verwendeten IPv6 Prefixes. Stimmt das Prefix dann nicht überein, kann er das Paket als ungültig verwerfen.

Im genannten Beispiel erhält Router 1 (`10.0.0.1`) ein ISATAP Paket von Router 2 mit der Zieladresse `2001:100:2::5efe:10.0.0.1`. Obwohl das IPv6 Prefix nicht auf Router 1 konfiguriert ist, besitzt der Router die eingebettete IPv4 Adresse. Wäre diese global eindeutig, muss ein Prefix Fehler vorliegen und das Paket sollte verworfen werden, um ein Kreisrouting zu vermeiden.

Dieses Verfahren funktioniert nur, wenn die IPv4 Adressen global eindeutig sind, also das *u*-Bit gesetzt ist. Da bei privaten IPv4 Adressen die niederwertigen 64 Bit der ISATAP Adresse *nicht* eindeutig sind, können Pakete auch dann gültig sein, wenn das Prefix nicht übereinstimmt.

4.2.7. Spoofing bei ICMPv4 Fehlern

Auch die Behandlung von ICMPv4 Fehlern birgt erhöhtes Risiko für Spoofing Attacken.

Bei einer nativen Verbindung entstehen ICMP Fehler üblicherweise auf dem Router zum Zielnetz. Dieser führt auf Layer-2 mehrfach *Address Resolution Protocol* (ARP) aus. Erst bei wiederholten Layer-2 Fehlern generiert der Router eine ICMP *Destination Unreachable* Nachricht, für die er autorativ ist. ARP Fehler lassen sich nur schwer von Spoofern provozieren.

Bei ISATAP hingegen ist Layer-2 die darunterliegende IPv4 Verbindung, von der der Host ICMPv4 Fehlermeldungen von anonymen und nicht vertrauendwürdigen Routern auf dem Link empfangen kann. Da der Host dazwischen liegende Router nicht kennt, kann ein Spoofer ICMPv4 Nachrichten fingieren, die als Layer-2 Fehler interpretiert werden würden. Es ist daher nicht empfehlenswert, jeden empfangenen ICMPv4 Fehler als Layer-2 Fehler zu verarbeiten und ICMPv6 Fehler zu generieren.

Vielmehr sollten, wie auch bei ARP, nur wiederholte ICMPv4 Fehler als “weiche” Anzeichen gewertet werden, dass Link auf Layer-2 gestört sein könnte. Empfangene ICMPv4 Fehler sollten genau geprüft werden, ob sie zu gesendeten Paketen passen und nicht von einem *Man-In-The-Middle* eingeschleust wurden.

5. Unterstützung gängiger Betriebssysteme

5.1. Windows

5.1.1. Windows XP

Windows XP enthält einen IPv6 Stack, der allerdings nicht vorinstalliert ist, sondern von Hand nachinstalliert werden muss. Der Stack ist nicht ausgereift und nicht für den produktiven Einsatz gedacht. Da der Stack für Endanwender im Normalfall nicht aktiviert ist, sind die Unzulänglichkeiten allerdings nicht von größerer Bedeutung.

Sobald der IPv6 Stack installiert ist, wird *Stateless Autoconf* für die konfigurierten Netzwerkschnittstellen durchgeführt und automatisch Tunnel für Teredo und ISATAP erstellt. Bei Verfügbarkeit einer globalen IPv4 Adresse, wird zusätzlich ein *6to4* Tunnel konfiguriert.

Ist eine native IPv6 Verbindung vorhanden, wird diese den Übergangstechniken vorgezogen. So verweigert Windows z.B. die Konfiguration einer ISATAP Verbindung, wenn native Konnektivität vorhanden ist; ein Teredo-Tunnel wird dennoch installiert.

Konfiguration von IPv6 Unter Windows XP lässt sich IPv6 nur über die Kommandozeile konfigurieren.

Listing 5.1: 6to4 und Teredo unter Windows XP deaktivieren

```
> netsh interface ipv6 6to4 set state state=disabled
OK.

> netsh interface ipv6 set teredo disable
OK.
```

Listing 5.2: ISATAP unter Windows XP konfigurieren

```
> netsh interface ipv6 isatap set state enabled
OK.

> netsh interface ipv6 isatap show router
Routername                  : default
Relay verwenden             : default
Auflosungsintervall         : default

> netsh interface ipv6 isatap set router isatap.fh-weingarten.de \
      interval=15
OK.
```

Es kann lediglich ein DNS Name als ISATAP Router konfiguriert werden, diesem können aber mehrere IPv4 Adressen zugeordnet sein. Das Standardintervall unter Windows XP, in dem *Router Solicitations* gesendet werden sollen, beträgt 15 Minuten.

ISATAP IPv6 Adresse Windows XP konfiguriert die ISATAP Tunnel Schnittstelle mit einer entsprechenden linklokalen IPv6 Adresse. Hier fällt auf, dass Windows XP das *u*-Bit auch bei globaler IPv4 Adresse stets auf 0 setzt und nicht auf 1, wie in Kapitel 4.1.1 beschrieben.

Abbildung 5.1.: Windows XP linklokale ISATAP Adresse

```
Tunneladapter Automatic Tunneling Pseudo-Interface:

        Verbindungsspezifisches DNS-Suffix:
        IP-Adresse. . . . . . . . . . . . : fe80::5efe:141.69.100.100%2
        Standardgateway . . . . . . . . . :

C:\Dokumente und Einstellungen\Sascha>
```

Die Implementierung in Windows XP folgt dabei einem frühen Entwurf der ISATAP Spezifikation, der bis Version 15[1] die Verwendung des *u*-Bits nicht vorsah. Erst ab Version 16[2] des Entwurfs wird das *u*-Bit beschrieben. Ab Windows Vista wird die Adresse RFC-konform gesetzt. Auch ohne ISATAP Router kann man bereits über linklokale Adressen kommunizieren, wenn dabei die Schnittstelle mit angegeben wird, z.B.:

```
> ping fe80::5efe:192.168.6.5%2
```

Firewall Aktuelle Windows Versionen haben standardmäßig für alle Netzwerkschnittstellen eine stateful Firewall aktiviert, die Verbindungen von außen nach innen filtern. Für Diagnosezwecke ist diese allerdings störend.

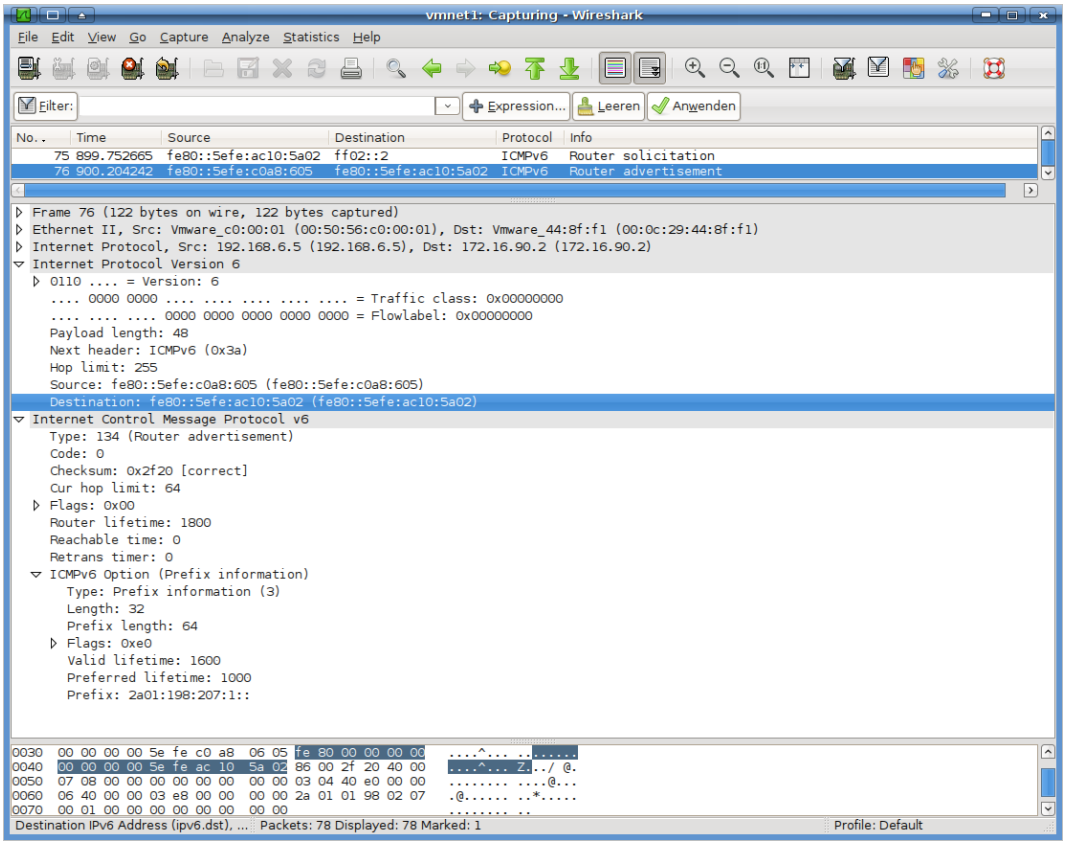
Die Firewallfähigkeiten in Bezug auf IPv6 und insbesondere getunnelten Datenverkehr wurde in dieser Arbeit nicht weiter untersucht.

Stateless Autoconf Windows XP führt auf der ISATAP Schnittstelle *Stateless Autoconf* aus, sobald der ISATAP Router erreichbar ist. Dazu werden Unicast IPv4 Pakete an die Adressen des ISATAP Routers geschickt. Die Payload enthält ein ICMPv6 Paket, das an die Multicast Adresse `ff02::2` gerichtet ist. Obwohl diese Adresse alle im Netz verfügbaren Router anspricht, wird das Paket lediglich von dem per IPv4-Unicast angesprochenen ISATAP Router empfangen. Die Antwort des Routers ist an die linklokale Unicast Adresse des Clients gerichtet (Abbildung 5.2).

Das empfangene *Router Advertisement* enthält Informationen über das Prefix des Subnetzes und die Gültigkeitsdauer der Routen. Diese Informationen müssen erneut angefordert werden, bevor die Gültigkeit der Adressen erlischt. Ein Test zeigt, dass Windows XP die Informationen des *Router Advertisements* überhaupt nicht auswertet, sondern nur nach Ablauf des konfigurierten Auflösungsintervals erneut *Router Solicitations* sendet. Ist die Lebensdauer des Prefixes geringer als dieses Interval, verlieren die Adressen für den Rest der Zeit ihre Gültigkeit und der Rechner verliert die IPv6 Konnektivität.

[1] isatap-draft-15: `http://tools.ietf.org/html/draft-ietf-ngtrans-isatap-15#section-4.1`
[2] isatap-draft-16: `http://tools.ietf.org/html/draft-ietf-ngtrans-isatap-16#section-4.1`

Abbildung 5.2.: Windows XP ISATAP Stateless Autoconf



Empfang von ICMPv4 Zwar zeigt der `ping`-Befehl von Windows XP ICMPv4 Meldungen an, die von Routern verschickt werden, aber der Fehler scheint nicht an den IPv6-Stack weitergereicht zu werden. Im Fehlerfall können Anwendungen nicht reagieren und geraten in Timeouts. Eigene ARP Fehler werden ebenfalls nicht behandelt.

Senden von ICMPv4 Empfängt Windows XP bei aktiviertem IPv6 Stack ein verirrtes UDP Paket, schickt es ein *ICMPv4 Destination unreachable (Port unreachable)* zurück. Als Payload sendet Windows XP exakt 148 Byte: IPv4 Header (20 Byte), UDP Header (8 Byte) und 120 Byte der Payload des UDP Pakets.

Ist kein IPv6 Stack installiert und wird ein *proto-41* Paket empfangen, antwortet der Host mit einem ICMPv4 Destination unreachable (Protocol unreachable). Ist ein IPv6 Stack installiert aber keine ISATAP Verbindung konfiguriert, wird keine ICMPv4 Fehlernachricht gesendet.

Beispiel Trace von ICMPv4 Der folgende Mitschnitt zeigt die Antwort von Windows XP ohne IPv6-Stack auf ein empfangenes ISATAP-Paket. Das original Paket bestand aus:

1. IPv4 Header (20 Byte)
   von `192.168.1.110` an `192.168.1.100`, Protokoll `0x29`=41

2. IPv6 Header (40 Byte)
   von `fe80::5efe:c0a8:16a` an `fe80::5efe:c0a8:164`
   Next Header `0x31`=58=ICMPv6

3. ICMPv6 (8 Byte) + Daten
   Typ 128=Echo Request, Code 0

Listing 5.3: ICMPv4 Paket mit IPv6 als Payload

```
<- IP-Header (20)
45 00 00 98 02 38 00 00  80 01 b4 0e c0 a8 01 64
         -> <- ICMPv4 Header (8)  -> <- IP-Header
c0 a8 01 6a 03 02 62 f7  00 00 00 00 45 00 00 7c
                                                ->
00 00 40 00 40 29 b6 3a  c0 a8 01 6a c0 a8 01 64
<- Payload IPv4 (=IPv6 Header)
60 00 00 00 00 40 3a 40  fe 80 00 00 00 00 00 00

00 00 5e fe c0 a8 01 6a  fe 80 00 00 00 00 00 00
                      ->  <- Payload (=ICMPv6)
00 00 5e fe c0 a8 01 64  80 00 63 42 cf 6d 00 01
                                                --
fa 12 88 4a 92 55 0e 00  08 09 0a 0b 0c 0d 0e 0f..
```

Das hier abgedruckte Antwortpaket von Windows XP besteht aus:

1. IPv4 Header (20 Byte)
   von `192.168.1.100` an `192.168.1.110`, Protokoll `0x01`=ICMP

2. ICMPv4 (8 Byte)
   Typ 3 (destination unreachable), Code 2 (protocol unreachable)

3. Originalpaket ab IP Header ($\geq$ 84 Byte)
   - IPv4 Header (20 Byte)
     von `192.168.1.110` an `192.168.1.100`
   - IPv6 Header (40 Byte)
     von `fe80::5efe:c0a8:16a` an `fe80::5efe:c0a8:164`
   - ICMPv6 (8 Byte) + Daten
     Typ 128=Echo Request, Code 0

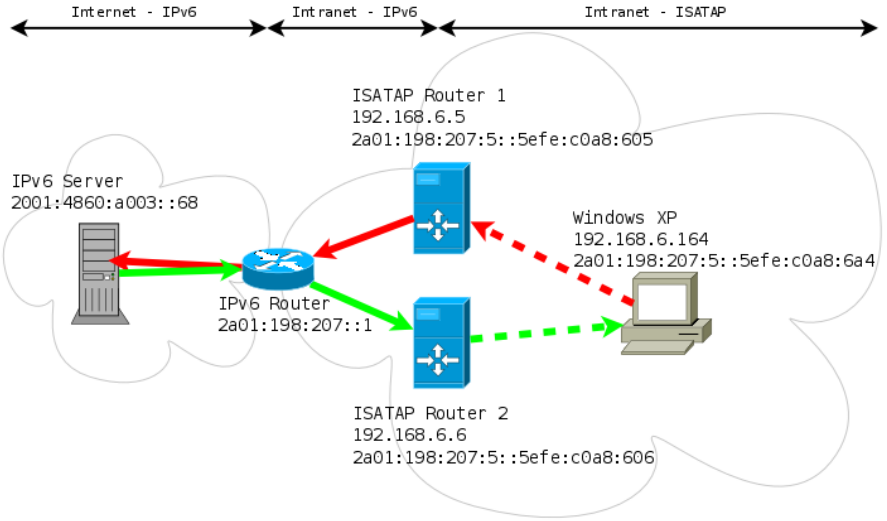
Dabei ist erfreulich, dass Windows XP als Payload des ICMPv4-Pakets weit mehr als die üblichen 8 Byte Payload zurücksendet, was ausreicht, um die Verbindung zu identifizieren, auf die sich das Paket bezieht. Ein ISATAP-Stack kann auf diese Antworten angemessen reagieren.

Prüfung der Source Adresse Bei linklokalen ISATAP Adressen prüft Windows XP, ob die eingebettete IPv4 Adresse mit der Quelladresse des Pakets übereinstimmt. Bei globalen ISATAP Adressen prüft Windows XP allerdings nicht mehr und nimmt Pakete aller Art von Hosts an. Dabei wird selbst subnetzfremder Datenverkehr von nicht-Routern angenommen, nicht nur von konfigurierten Routern.

Testszenario Abbildung 5.3 zeigt die aufgebaute Netztopologie, in dem zwei ISATAP Router für dasselbe ISATAP Netz installiert wurden. Das Windows XP System hat Router 1 mit der IP `192.168.6.5` als ISATAP Router eingetragen, führt *Stateless Autoconf* durch, erhält das verwendete ISATAP Prefix und konfiguriert seine korrekte ISATAP Adresse. Danach benutzt er Router 1 als Router für ausgehende IPv6 Pakete.

Auf dem IPv6 Gateway ist die Route für das ISATAP Netz aber auf Router 2 mit der IP `192.168.6.6` gesetzt, sodass einkommende Pakete an diesen Router gesendet werden, der sie dann in das ISATAP Netz einspeist. Damit erhält der Windows XP Host einkommende Pakete von Router 2, der aber dort nicht als ISATAP Router eingetragen ist. Da der Sender des subnetzfremden Datenverkehrs nicht als Router konfiguriert ist, sollte Windows XP den Datenverkehr eigentlich verwerfen.

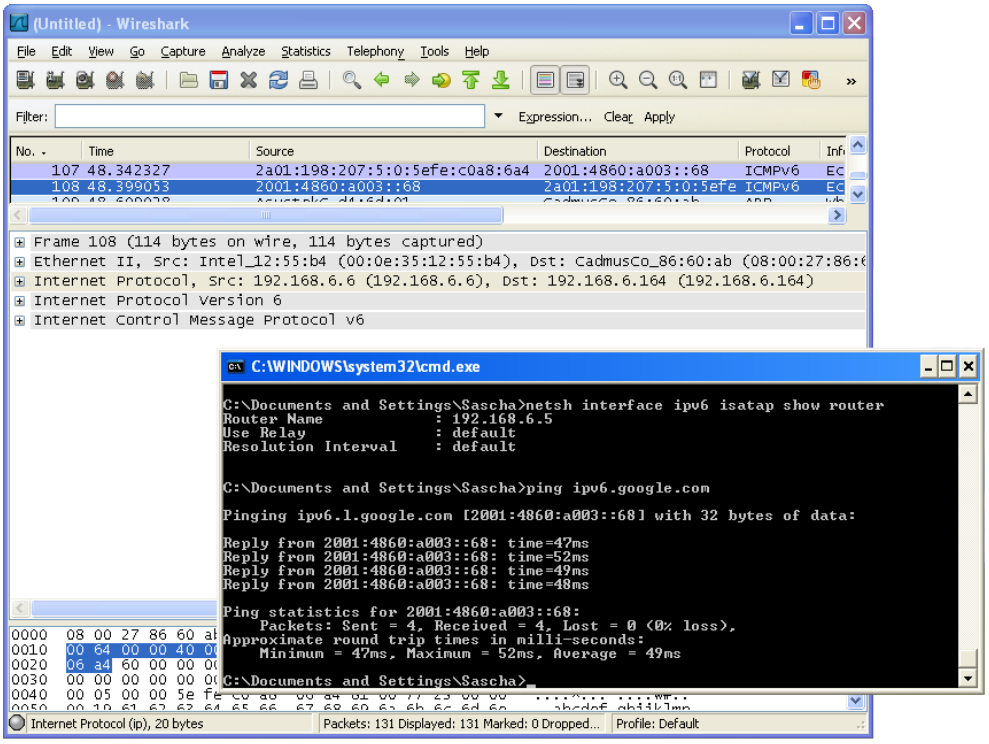
Abbildung 5.3.: Windows XP Szenario asynchrone ISATAP Routen



Mitschnitt Wie in Abbildung 5.4 erkennbar, nimmt Windows XP subnetzfremde Pakete von Router 2 (`192.168.6.6`) an, obwohl dieser nicht als ISATAP Router konfiguriert ist.

Für dieses Szenario wurde allerdings die Windows-interne Firewall deaktiviert. Bei aktivierter Firewall werden die Antwortpakete verworfen, was den Schluss nahelegt, dass der ISATAP Verkehr bereits im IPv4 Stack (stateful) gefiltert wird, und nicht im IPv6 Stack.

Abbildung 5.4.: Windows XP Mitschnitt von asynchronen ISATAP Routen



5.1.2. Windows Vista

Bei Windows Vista ist standardmäßig ein IPv6 Stack installiert, sodass automatisch IPv6 Verbindungen konfiguriert werden. Auch unter Windows Vista wird ISATAP nur aktiviert, wenn keine native IPv6 Konnektivität gegeben ist. Die Implementierung unterscheidet sich von Windows XP in folgenden Punkten.

Adressformat Windows Vista konfiguriert korrekte ISATAP Adressen; insbesondere wird endlich das u-Bit der IPv6 Adresse korrekt gesetzt, sobald eine globale IPv4 Adresse verwendet wird.

Subnetzfremder Datenverkehr Windows Vista nimmt subnetzfremden Datenverkehr von nicht-Routern auch an, wenn die Firewall aktiviert ist. Das Szenario aus Abbildung 5.3 funktioniert damit immer, was Spoofingangriffe weiter begünstigt.

Stateless Autoconf Windows Vista sendet *Router Solicitations* nicht mehr an die Multicast Adresse `ff02::2`, sondern an die IPv6 linklokale Unicast Adresse, die gemäß `rfc5214, Abschnitt 6.1` aus der IPv4 Adresse abgeleitet wird. Ist der ISATAP Router nicht mit einer ordnungsgemäßen Adresse konfiguriert, schlägt der *Stateless Autoconf*-Mechanismus fehl, was beispielsweise bei vielen Cisco Geräten noch der Fall ist (Siehe

Kapitel 5.2).

Wie Windows XP wertet auch Windows Vista empfangene *Router Advertisements* nicht detailliert genug aus. Ist die Lebenszeit der Routen und Adressen im *Router Advertisement* geringer als das konfigurierte Intervall, verliert Windows für die Zwischenzeit IPv6 Konnektivität, bis erneut *Router Solicitations* gesendet werden.

ICMPv4 Pakete Eim empfangenes ICMPv4 Paket wird direkt an den IPv6 Stack weitergereicht, der daraufhin reagieren kann.

In Listing 5.4 kann für Host `141.69.100.100` kein gültiges ICMP Paket empfangen werden; es kommt zu Paketverlust und Timeouts. Für Host `10.0.0.10` wird allerdings eine ICMPv4 Nachricht erhalten, die der IPv6 Stack sofort in ein simples *Zielanschluss nicht erreichbar* übersetzt.

Listing 5.4: Empfang von ICMPv4 Nachrichten in Windows Vista

```
C:\Windows\system32>ping fe80::5efe:141.69.100.100%9

Ping wird ausgefuehrt fuer fe80::5efe:141.69.100.100%9
    von fe80::5efe:192.168.6.177%9 mit 32 Bytes Daten:

Zeitueberschreitung der Anforderung.
Zeitueberschreitung der Anforderung.
Zeitueberschreitung der Anforderung.
Zeitueberschreitung der Anforderung.

Ping-Statistik fuer fe80::5efe:141.69.100.100%9:
    Pakete: Gesendet = 4, Empfangen = 0, Verloren = 4 (100% Verlust),

C:\Windows\system32>ping fe80::5efe:10.0.0.10%9

Ping wird ausgefuehrt fuer fe80::5efe:10.0.0.10%9
    von fe80::5efe:192.168.6.177%9 mit 32 Bytes Daten:

Zielanschluss nicht erreichbar.
Zielanschluss nicht erreichbar.
Zielanschluss nicht erreichbar.
Zielanschluss nicht erreichbar.

Ping-Statistik fuer fe80::5efe:10.0.0.10%9:
    Pakete: Gesendet = 4, Empfangen = 0, Verloren = 4 (100% Verlust),
```

Fazit Damit steht in Windows Vista eine relativ zuverlässige und ausgereifte ISATAP Implementierung zur Verfügung, obwohl sie sich in manchen Punkten nicht vollständig an `rfc5214` hält.

5.1.3. Windows 7

Der ISATAP Stack von Windows 7 gleicht dem in Windows Vista und wurde nicht weiter untersucht.

5.2. Cisco IOS

Untersucht wurde die Tauglichkeit eines Cisco 2600 (2621XM) Geräts mit IOS Version 12.3(22) als ISATAP Router. Die Tauglichkeit als ISATAP Client wurde nicht untersucht.

5.2.1. Konfiguration als ISATAP Router

Die Konfiguration einer Tunnel-Schnittstelle für ISATAP ist einfach:

Listing 5.5: Konfiguration eines ISATAP Routers unter Cisco-IOS

```
!
interface Tunnel2
 description ISATAP zum 66erNetz
 no ip address
 no ip redirects
 ipv6 address 2001:7C0:1900:11::/64 eui-64
 no ipv6 nd suppress-ra
 tunnel source FastEthernet0/0
 tunnel mode ipv6ip isatap
!
```

Listing 5.6: Konfigurierte ISATAP Schnittstelle unter Cisco-IOS

```
R2621_IPv6#show ipv6 interface

[...]
Tunnel2 is up, line protocol is up
  IPv6 is enabled, link-local address is FE80::5EFE:8D45:106
  Description: ISATAP zum 66erNetz
  Global unicast address(es):
    2001:7C0:1900:11:0:5EFE:8D45:106, subnet is 2001:7C0:1900:11::/64
  Joined group address(es):
    FF02::1
    FF02::2
    FF02::1:FF45:106
  MTU is 1480 bytes
  ICMP error messages limited to one every 100 milliseconds
  ICMP redirects are enabled
  ND DAD is not supported
  ND reachable time is 30000 milliseconds
  ND advertised reachable time is 0 milliseconds
  ND advertised retransmit interval is 0 milliseconds
  ND router advertisements are sent every 200 seconds
  ND router advertisements live for 1800 seconds
  Hosts use stateless autoconfig for addresses.
```

Über den Tunnel Modus *isatap* wird aus der IPv4 Adresse der Tunnel Schnittstelle *Ethernet0/0* und dem konfiguriertem Subnetz eine ISATAP IPv6 Adresse gebildet und konfiguriert.

Ungültige ISATAP Adresse Es fällt auf, dass das u-Bit auch bei Verwendung einer globalen IPv4 Adresse nicht gesetzt ist, die ISATAP IPv6 Adresse `FE80::5EFE:8D45:106` also nicht RFC-konform ist; vermutlich folgt auch Cisco-IOS noch der Spezifikation

früherer Entwürfe, in denen das *u*-Bit noch nicht vorgesehen war. Der Router mit der IPv4 Adresse `141.69.1.6` ist nur unter der IPv6 Adresse `fe80::5efe:8d45:106` erreichbar:

```
# ping6 -I is0 fe80::5efe:8d45:106
PING fe80::5efe:8d45:106(fe80::5efe:8d45:106) from fe80::200:5efe:8d45
    :7977 is0: 56 data bytes
64 bytes from fe80::5efe:8d45:106: icmp_seq=1 ttl=64 time=1.80 ms
```

Wie in Kapitel 4.2.2 beschrieben, zeigen alle ISATAP Router, die gefunden wurden, dieses Verhalten. Windows XP umschifft diesen Fehler indem die *Router Solicitation* an die IPv6 Multicastadresse `ff02::2` gerichtet wird, eingebettet in IPv4 Unicast.

Windows Vista und Windows 7 können mit diesem Router keine automatische Verbindung herstellen, da Router Solicitations nur an die Adresse `fe80::200:5efe:8d45:106` gerichtet werden, die aber nicht auf dem Router konfiguriert ist.

Stateless Autoconf Der Router beantwortet Router Solicitations an seine linklokale IPv6 Adresse ordnungsgemäß und setzt dabei die Router Lifetime standardmäßig auf 1800 Sekunden. Durch die fehlerhafte IPv6 Adresse bei einer globalen IPv4 Adresse, schlägt der Mechanismus allerdings fehl, wenn der Client *Router Solicitations* nur an die RFC-konforme ISATAP Adresse richtet.

Die in Kapitel 8.2 vorgestellte Lösung sendet *Router Solicitations* im Falle einer globalen IPv4 Adresse an beide möglichen ISATAP IPv6 Adressen, um den Fehler zu umgehen.

5.2.2. Prüfung von ISATAP Adressen

Empfängt der Cisco Router ein ISATAP Paket aus demselben Subnetz, wird die eingebettete IPv4 Adresse *nicht* überprüft.

Testszenario Mit wenigen Befehlen lassen sich unter Linux spezielle *proto-41* Pakete senden:

```
1 # ip tunnel add name is0 mode sit local 141.69.55.43 remote 141.69.1.6
  # ip link set is0 up
3 # ip -6 route add default via fe80::5efe:141.69.1.6 dev is0
  # ip -6 addr add 2001:7c0:1900:11:0:5efe:141.69.55.44 dev is0
5
  # ping6 ipv6.google.com
7 ...
```

Hier weicht die eingebettete IPv4 Adresse der konfigurierten ISATAP Adresse von der tatsächlichen IPv4 Adresse ab. Zwar werden Pakete vom Router nach außen geroutet, eine bidirektionale Kommunikation ist damit aber nicht möglich.

Beliebige IPv6 Adressen Der Router prüft ebenfalls nicht, ob die Quelladresse des *proto-41* Pakets überhaupt aus dem ISATAP Netz stammt. In obigem Szenario (Zeile 4) kann sich ein ISATAP Host demnach beliebige IPv6 Adressen konfigurieren. Solange die IPv6 Packete als *proto-41* an den Router gesendet werden, routet er sie weiter.

Routing von innen nach innen Der eingesetzte Cisco Router leitet empfangene *proto-41* Pakete, die von einem internen ISATAP Client kommen, auch an andere interne ISATAP Clients weiter (Szenario 2 aus Abbildung 4.2).

Das ist insofern problematisch, da die Quelladresse des Routers bei allen Clients besonderes Vertrauen genießt und subnetzfremder Verkehr vom Router immer ohne weitere Prüfung angenommen wird. Da der Router aber auch subnetzfremden Verkehr von internen Clients annimmt und weiterreicht, können so auch von innen beliebige IPv6 Pakete an andere ISATAP Teilnehmer gesendet werden, die bei direkter Kommunikation zwischen den Clients abgelehnt würden. *Access Control Lists* auf dem Cisco Router können das verhindern.

5.2.3. ICMPv4

In der eingesetzten Konfiguration sendet der Router empfangene ICMPv4 Pakete nicht über IPv6 zurück. Pakete an nicht erreichbare ISATAP IPv6 Adressen gehen ohne Information verloren.

Das ist allerdings nur dann von Bedeutung, wenn Pakete von außen an einen ISATAP Host gesendet werden und der Zielhost nicht erreichbar ist. Hosts außerhalb des Netzwerks werden so nicht über nicht-verfügbarkeit eines ISATAP Hosts informiert und geraten in Timeouts.

Kommunikation intern ist nicht betroffen, da ISATAP Clients sich üblicherweise direkt erreichen und nicht über den Router.

5.2.4. Fazit

Die eingesetzte Firmware 12.3(22) ist strenggenommen nur bedingt für ISATAP geeignet. In der Standardkonfiguration ist der Betrieb mit Windows Vista und Windows 7 nicht möglich und das ISATAP Routing ist nur unzureichend abgesichert. Durch manuelle Konfiguration des Routers und von *ACL*s lassen sich diese Probleme jedoch umgehen.

Listing 5.7: Konfiguration einer korrekten ISATAP Adresse auf einem Cisco Router

```
R2621_IPv6#conf t
R2621_IPv6(config)#interface Tunnel2
R2621_IPv6(config-if)#ipv6 address FE80::200:5EFE:8D45:106 link-local
```

Hierbei wird die ISATAP Adresse allerdings nicht mehr automatisch von der IPv4 Adresse abgeleitet.

Neue Firmware Eine Aktualisierung der Firmware auf Version 12.3(26) brachte keine nennenswerte Verbesserung in Bezug auf ISATAP, sodass die Unterstützung noch immer dürftig ist.

5.3. FreeBSD

Die offizielle FreeBSD Distribution enthält zum Zeitpunkt der Arbeit keine ISATAP-Unterstützung. Das KAME Projekt[3], dessen Ziel es war, eine freie Implementation von IPv6 und IPSec für BSD-Systeme zu schaffen, soll zwar eine funktionierende ISATAP Unterstützung enthalten[4], diese wurde aber bisher noch nicht in den offiziellen FreeBSD Kernel übernommen. Die Entwicklung des KAME Projekts wurde im März 2006 offiziell abgeschlossen. Da eine offizielle Unterstützung von ISATAP in FreeBSD fehlt, wurde auch die Implementierung des KAME Projektes im Rahmen dieser Arbeit nicht weiter getestet.

Die Implementierung des Userspace Programms isatapd aus dem Miredo Projekt wurde vom Autor dieser Arbeit nicht unter FreeBSD getestet. Eine ausführliche Analyse des Programms unter Linux folgt in Kapitel 5.5.1.

Konfiguration GIF Tunnel Unter FreeBSD lassen sich mit einem *GIF*-Tunnel IPv6 Pakete in IPv4 transporieren. Damit ist Kommunikation über einen ISATAP Router als Gateway möglich.

Listing 5.8: Konfiguration eines GIF-Tunnels für ISATAP unter FreeBSD

```
# ifconfig rl0 10.0.0.20 netmask 255.255.255.0

# ifconfig gif0 create
# ifconfig gif0 tunnel 10.0.0.20 10.0.0.1
# ifconfig gif0 inet6 fe80::5efe:a00:14 prefixlen 64 alias
# ifconfig gif0 inet6 2001:db8:1234:5:0:5efe:a00:14 prefixlen 64 alias

# route add -inet6 2001:db8:1234:5::/64 -interface gif0
# route add -inet6 default -interface gif0
```

Da über den Tunnel kein *Stateless Autoconf* stattfindet, muss das globale Prefix und die Standard-Route von Hand konfiguriert werden.

Da *proto-41*-Pakete nicht von ISATAP Nachbarn angenommen wird, sondern nur vom Tunnel-Endpunkt, kann über diesen Tunnel nicht mit anderen ISATAP Hosts kommuniziert werden. Der Tunnel ist also *nicht* RFC 5214 konform.

5.4. MacOSX

Eine offizielle Unterstützung von ISATAP unter MacOSX konnte zum Zeitpunkt dieser Arbeit nicht festgestellt werden. Wegen Ähnlichkeit des Darwin-Kernels zu FreeBSD ist aber sowohl der Betrieb von isatapd aus dem Miredo Projekt als auch die manuelle Konfiguration eines GIF Tunnels möglich, um Zugang zu einem ISATAP Netz zu erhalten. Beide Möglichkeiten wurden nicht vom Autor getestet, da zum Zeitpunkt der Arbeit kein geeignetes System zur Verfügung stand.

[3] KAME-Projekt: `http://www.kame.net`
[4] `http://www.kame.net/newsletter/20041201/`

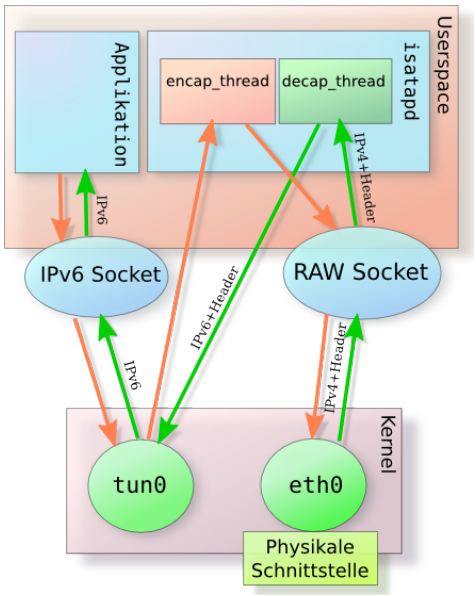
5.5. Linux 2.6.30

Diese Beschreibung bezieht sich auf Linux Version 2.6.30, die zu Beginn der Arbeit vorlag und beschreibt die zum Teil unzufriedenstellende ISATAP Unterstützung als Blackbox. Eine detaillierte Analyse der Implementierung und die Beschreibung der Verbesserungen in 2.6.31 findet sich in Kapitel 7.

5.5.1. isatapd aus dem Miredo Projekt

Das Miredo Projekt[5] stellt eigentlich eine Implementierung von Teredo für Linux und BSD bereit, enthielt aber bis Version 1.1.5 zusätzlich einen experimentellen ISATAP Daemon. Der ISATAP Daemon wurde von den Entwicklern selbst als *buggy* bezeichnet und mit Version 1.1.6 entfernt[6], da bereits native ISATAP Unterstützung in den Linux Kernel gelangt war. Der Daemon unterstützt den Betrieb als Client und als Router. Für den Betrieb als ISATAP Client wird lediglich die IPv4 Adresse des Routers übergeben, der Rest geschieht automatisch. Der Quelltext von isatapd ist nur noch im Quellpaket von miredo-1.1.5 zu finden!

Abbildung 5.5.: Funktionsweise von isatapd aus dem Miredo Projekt



Funktionsweise isatapd erstellt unter Linux und BSD einen *tun*-Tunnel und konfiguriert ihn mit einer korrekten linklokalen ISATAP IPv6 Adresse. Über einen IPv4 RAW-Socket, der an eine IPv4 Adresse gebunden wird, werden IPv4 Pakete empfangen und gesendet, die als Payload IPv6 Pakete enthalten (*proto-41*).

Empfangsthread Der Thread `decap_thread` empfängt diese IPv4 Pakete von der physikalischen Schnittstelle, packt das eingebettete IPv6 Paket aus und sendet es in die *tun*-Schnittstelle weiter. Anwendungen empfangen IPv6 Pakete transparent aus der virtuellen *tun*-Schnittstelle.

Der empfangene ISATAP Datenverkehr wird nur unzureichend überprüft; so werden auch Datenpakete angenommen, bei denen die IPv4 Adresse nicht mit der eingebetteten Adresse übereinstimmt. Zudem wird nicht zwischen normalen Hosts und Routern unterschieden.

[5] Miredo: `http://www.remlab.net/miredo`
[6] Remove buggy ISATAP support:
`http://git.remlab.net/cgi-bin/gitweb.cgi?p=miredo.git;a=commit;h=08be066`

Sendethread Der Thread `encap_thread` empfängt IPv6-Daten von der virtuellen *tun*-Schnittstelle aus dem Userspace, packt diese in IPv4 ein und sendet sie über den IPv4-RAW-Socket als *proto-41* weiter.

Ist die Zieladresse eine ISATAP Adresse desselben Subnetzes, wird das Paket direkt über IPv4 an den Empfänger geschickt. In allen anderen Fällen, einschließlich IPv6-Multicast, wird das Paket an den ISATAP Router gerichtet.

Unzulänglichkeiten

- Das Programm versendet keine *Router Solicitations*. Da auch IPv6-Multicast an den Router weitergeleitet wird, kann normales Stateless Autoconf erfolgreich sein, da Linux automatische *Router Solicitations* allerdings nur einmalig sendet, wenn die Schnittstelle aktiviert wird, verlieren die automatisch konfigurierten Adressen und Routen ihre Gültigkeit und werden nicht aufgefrischt. Abhilfe schafft das Tool rdisc6 aus dem ndisc6-Paket[7]. Beispiel:

  Listing 5.9: Manuelles Senden von Router Solicitations mit 'rdisc6'

  ```
  # sudo ./rdisc6 -1 fe80::5efe:c0a8:0102 is0
  ```

- ICMPv4-Fehler werden nicht verarbeitet und gehen verloren.

- Es gibt Probleme, wenn eine weitere Netzwerkschnittstelle verwendet werden soll, die an dieselbe IPv4 Adresse gebunden ist. So kann isatapd nicht zeitgleich mit einem *sit*-Tunnel verwendet werden, der ebenfalls *proto-41*-Verkehr verarbeiten möchte.

- Es kann nur ein einzelner Router gleichzeitig verwendet werden.

- isatapd nimmt Datenverkehr von allen Hosts ohne weitere Überprüfung an; PRLs sind nur unvollständig implementiert.

- Da das Programm Datenverkehr im Userspace verarbeitet, ist es anfällig gegenüber Attacken. Um dem vorzubeugen gibt das Programm seine root-Rechte ab, sobald der Tunnel konfiguriert ist.

- Die IPv6 Routingtabelle des Kernels wird ignoriert, da nur die Zieladresse des IPv6-Pakets beachtet wird. So lassen sich z.B. keine Netzwerke über ISATAP verbinden:

  ```
  ip -6 route add 2001:db8:0:22::/64 via 2001:db8:0:11:0:5efe:10.0.0.5
  ```

  Empfängt der Host ein Paket für `2001:db8:0:22::1`, würde er dieses nicht an den eingetragenen Gateway `2001:db8:1234:1000:0:5efe:10.0.0.5` senden, sondern an den ISATAP Default Router, da das Programm entscheidet, dass die Zieladresse nicht auf dem ISATAP Link erreichbar ist.

[7] Router Discovery aus dem Userspace: `http://www.remlab.net/ndisc6`

5.5.2. Konfiguration SIT Tunnel

Ein regulärer *sit*-Tunnel kann so konfiguriert werden, das er in einem ISATAP Subnetz verwendet werden kann. Der *sit*-Tunnel verhält sich wie ein gif-Tunnel aus FreeBSD und ist nicht RFC-5214-konform!

Listing 5.10: Konfiguration eines SIT-Tunnels für ISATAP unter Linux

```
1  # modprobe sit

3  # ip tunnel add name is0 mode sit local 10.0.0.2 remote 10.0.0.1 dev eth0
   # ip -6 addr add fe80::5efe:a00:2/64 dev is0
5  # ip link set is0 up
   # ip -6 addr add 2001:db8::5efe:a00:2/64 dev is0
7  # ip -6 route add default via 2001:db8::5efe:a00:1 dev is0
```

Dieser statische *6in4*-Tunnel bewirkt, dass alle ausgehenden IPv6 Pakete an die IPv4 Adresse `10.0.0.1` gesendet werden, in diesem Beispiel der ISATAP Router. Da die IPv6 Adressen gültige ISATAP Adressen sind, leitet der Router die Pakete wie gewünscht weiter. Auch Antwortpakete, die vom Router an den Client gesendet werden, werden entgegengenommen. Die Konfiguration setzt sowohl Kenntnis über die IPv4 Adresse des Clients, des Routers, als auch das verwendete IPv6 ISATAP Prefix voraus.

Wegen der bisher unzureichenden Unterstützung von ISATAP unter Linux und der schlechten Dokumentation zu dessen Verwendung, ist diese Konfiguration noch in vielen Anleitungen im Internet beschrieben. So ist diese Vorgehensweise auch beim LRZ-München noch beschrieben[8].

Stateless Autoconf Zwar ist Multicast über einen *sit*-Tunnel nicht direkt verfügbar, aber da alle Pakete an die Endadresse geschickt werden, kommen auch Pakete mit der Multicast-Zieladresse `ff02::02` beim Endpunkt an.

Linux sendet *Router Solicitations* üblicherweise genau nachdem die Schnittstelle aktiviert wird. Es ist danach die Aufgabe eines Routers, periodisch *Router Advertisements* zu senden, um die Adressen und Routen aufzufrischen. Dem ISATAP Router ist der Host allerdings nicht bekannt.

Im Fall von *sit* funktioniert *Stateless Autoconf*, wenn das Interface die korrekte linklokale ISATAP Adresse hat, bevor es in den up-Zustand wechselt (Zeile 4, Listing 5.10). Da Linux aber keine periodischen *Router Solicitations* sendet und der Router keine automatischen *Router Advertisements* an Hosts senden kann, können das Prefix und die Routen nicht aufgefrischt werden. Die automatisch konfigurierten IPv6 Adressen und Routen werden nach Ablauf der Lebensdauer wieder gelöscht und die Verbindung ist nicht mehr verfügbar. Auch hier kann das Programm rdisc6 verwendet, um regelmäßig *Router Solicitations* an den Router zu senden (Listing 5.9).

[8] ISATAP beim Leibniz-Rechenzentrum:
`http://www.lrz-muenchen.de/services/netz/ipv6/isatap1/index.html`

Kommunikation mit anderen ISATAP Hosts Die Verbindung mit echten ISATAP Hosts im selben Subnetz scheitert, da der so konfigurierte Tunnel ausschließlich Pakete vom ISATAP Router entgegennimmt und *proto-41* Datenverkehr von anderen ISATAP Nachbarn verwirft. Hosts, die auf dieselbe Weise konfiguriert sind, können sich ggf. über den ISATAP Router erreichen, wenn dieser netzinternen Datenverkehr von innen nach innen weiterleitet.

5.5.3. Konfiguration ISATAP Tunnel

Linux enthält seit Version 2.6.25 eine ISATAP Unterstützung im Kernel, die in Version 2.6.26 erheblich verbessert wurde. Unter Linux kümmert sich das *sit*-Modul (*Simple Internet Transition*) zugleich um *6in4*, *6to4* und *ISATAP*.

Die Konfiguration von ISATAP unter Linux ist nur wenig dokumentiert und widersprüchlich, da bereits vor der offiziellen Aufnahme in den Kernel verschiedene Patches gab, um ISATAP Unterstützung von Hand nachzurüsten.

Zwar ist die Unterstützung im Kernel ausreichend implementiert, aber zum Zeitpunkt der Arbeit war mit iproute2[9] lediglich ein einziges Userspace Programm zu finden, mit dem eine ISATAP Schnittstelle erstellt und konfiguriert werden kann. Die Unterstützung von PRL in iproute2 war allerdings unvollständig.

Linklokale Adresse

Listing 5.11: Konfiguration einer ISATAP Schnittstelle unter Linux mit iproute2

```
# modprobe sit

# ip tunnel add name is0 mode isatap local 10.0.0.2 dev eth0
# ip link set is0 up

# ip -6 addr show dev is0
7: is0: <NOARP,UP,LOWER_UP> mtu 1480
    inet6 fe80::5efe:a00:2/64 scope link
       valid_lft forever preferred_lft forever

# iptables -A INPUT  ! -s 10.0.0.0/24 -p 41 -j REJECT
# iptables -A OUTPUT ! -d 10.0.0.0/24 -p 41 -j REJECT
```

Eine Schnittstelle `eth0` und die konfigurierte IPv4-Adresse `10.0.0.2` vorausgesetzt, erstellen diese Kommandos eine ISATAP Schnittstelle `is0`. Der Kernel konfiguriert automatisch eine korrekte linklokale IPv6 Adresse `fe80::5efe:a00:2/64`. Wird eine globale IPv4 Adresse verwendet, wird das *u*-Bit entsprechend `rfc5214` korrekt auf 1 gesetzt. Der konfigurierte Tunnel hat als lokalen Endpunkt die IPv4 Adresse `10.0.0.2` und nimmt Pakete von allen Hosts an.

Durch Firewallregeln wird sichergestellt, dass ISATAP Datenverkehr nur innerhalb `10.0.0.0/24` möglich ist und sich der ISATAP Link nicht über den gesamten Globus

[9] iproute2: `http://www.linuxfoundation.org/en/Net:Iproute2`

erstreckt. Für mehr Sicherheit sollte die Filterung von *proto-41* auf dem IPv4 Router stattfinden und nicht auf den einzelnen Hosts.

Über die linklokale IPv6 Adresse kann bereits mit anderen Hosts auf demselben ISATAP Link kommuniziert werden. Bei der manuellen Konfiguration muss die lokale IPv4 Adresse bekannt sein, was die Vorgehensweise bei Einsatz von DHCP oder ppp erschwert.

Adressauswahl Der Anwender soll eine ISATAP Verbindung aufbauen können, ohne die derzeitige IPv4 Adresse wissen zu müssen.

```
# ip tunnel add name is0 mode isatap dev eth0
```

Möchte man einen Tunnel erstellen, der keinen lokalen Endpunkt hat sondern lediglich einer Schnittstelle zugeordnet ist, stößt man auf ein Problem. Das Kommando schlägt zwar nicht fehl, führt aber auch nicht zu einem ISATAP Tunnel: es kann pro lokalem und entferntem Endpunkt nur ein Tunnel existieren. Das *sit*-Modul erstellt aber immer einen Tunnel `sit0` mit `any any` als Endpunkte. Da damit bereits ein Tunnel für dieses Endpunktpaar existiert, wird kein weiterer Tunnel erstellt. Fälschlicherweise gibt das obige Kommando *Erfolg* zurück, wenn bereits ein Tunnel mit den gleichen Endpunkten existiert, obwohl es sich eigentlich um einen Fehler handelt. Dieses Problem wurde im Rahmen dieser Arbeit behoben (Anhang A.1 und A.2), Linux konfiguriert nach Eingabe des obigen Kommandos die Schnittstelle aber nun mit der ungültigen linklokalen ISATAP Adresse `fe80::5efe:0:0/64`, da als lokaler IPv4 Endpunkt `0.0.0.0` (=`any`) übergeben wurde. Auch dieser kleine Fehler wurde korrigiert (Anhang A.4), indem die Konfiguration einer solchen linklokalen Adresse abgelehnt wird.

Es können bei Betrieb weitere linklokalen Adressen konfiguriert oder entfernt werden, sodass ISATAP Tunnel für mehrere IPv4 Adressen konfiguriert werden können oder sich die IPv4 Adresse ändern kann ohne den ISATAP Tunnel neu konfigurieren zu müssen. In der Praxis gibt es dafür aber nur wenige Anwendungsfälle, da die Netzwerkschnittstellen meist komplett entfernt und neu konfiguriert werden.

IPv4 Adresse auf einer ISATAP Schnittstelle Bei Einsatz eines Routingprotokolls kann es hilfreich sein, die IPv4 Adresse auf der ISATAP Schnittstelle selbst zu konfigurieren, um nicht von darunterliegenden physikalischen Schnittstellen abhängig zu sein. Die physikalische Schnittstelle muss dabei keine IPv4 Adresse konfiguriert zu haben, wohl aber die Route.

```
# ip addr flush dev eth0

# ip tunnel add name is0 mode isatap local 10.0.0.1
# ip link set is0 up
# ip addr add 10.0.0.1/24 dev is0

# ip route del 10.0.0.0/24 dev is0
# ip route add 10.0.0.0/24 dev eth0
```

Wichtig ist dabei, dass das Subnetz `10.0.0.0/24` an die physikalische Schnittstelle `eth0` gebunden ist und nicht an `is0`.

5.5.4. Linux als ISATAP Router

Linux kann bereits als ISATAP Router arbeiten, da hier der Host weder Stateless Autoconf durchführen muss, noch die PRL verwendet. Auf einer ISATAP Schnittstelle kann ein *Router Advertisement Daemon* laufen, beispielsweise radvd[10]. Schnittstellen können über normales IPv6 Routing verbunden werden.

Listing 5.12: Konfiguration von Linux als ISATAP Router

```
# ip tunnel add name is0 mode isatap local 10.0.0.1
# ip link set is0 up
# ip -6 addr add 2001:db8:1234:5:0:5efe:a00:1/64 dev is0

# echo -n 1 > /proc/sys/net/ipv6/conf/all/forwarding

# iptables -A INPUT  ! -s 10.0.0.0/24 -p 41 -j REJECT
# iptables -A OUTPUT ! -d 10.0.0.0/24 -p 41 -j REJECT

# radvd -C radvd.conf
```

Ohne die iptables-Firewallregeln könnte sich jeder Host im Internet als Teil dieses IPv6 Subnetzes ausgeben (Abbildung 4.1). Ähnliche Sicherheitsvorkehrungen sollten für Clients getroffen werden. Der Einsatz von ISATAP und *6to4* auf demselben Host wird durch diese Firewallregeln allerdings sehr erschwert.

Listing 5.13: radvd.conf für Linux als ISATAP Router

```
interface is0
{
        AdvSendAdvert on;
        UnicastOnly on;

        prefix 2001:db8:1234:5::/64
        {
                AdvOnLink on;
                AdvRouterAddr on;
        };
};
```

Es ist zu beachten, dass *Router Solicitations* nur mit *Router Advertisements* über Unicast beantwortet werden dürfen, nicht über Multicast.

5.5.5. Potential Router List

Der Kernel verwaltet bereits eine PRL und lehnt netzfremden Datenverkehr von nicht-Routern ab, allerdings konnte diese PRL zum Zeitpunkt der Arbeit nicht über iproute2 konfiguriert werden. Zwar lässt sich einer ISATAP Schnittstelle eine globale IPv6 Adresse mit einer Standard-Route zuweisen, allerdings werden empfangene Pakete des Gateways abgelehnt, da dieser nicht in der PRL aufgeführt ist.

Die nötige API ist im Kernel enthalten, ist im Userspace aber unzureichend implementiert. Die PRL enthält lediglich IPv4 Adressen, deshalb muss DNS-Auflösung vorher im Userspace geschehen.

[10] Router Advertisement Daemon: `http://www.litech.org/radvd`

Stateless Autoconf Es findet kein automatisches *Stateless Autoconfiguration* mit Einträgen der PRL statt. Konfiguriert man globale IPv6 Adressen von Hand, erhält man eine funktionierende ISATAP Verbindung, dabei ist allerdings Wissen über das IPv6 Prefix und den IPv6 Router nötig. Auch hier kann das Programm rdisc6 verwendet werden, um *Router Solicitations* an einen möglichen Router zu senden.

Prüfung der ISATAP Adresse Linux nimmt ISATAP Pakete an, wenn:

- Die IPv6 Adresse des Senders eine korrekte ISATAP Adresse ist und das Prefix mit dem Prefix der Zieladresse übereinstimmt. Die eingebettete IPv4 Adresse der IPv6 Adresse muss mit der tatsächlichen IPv4 Adresse des Absenders übereinstimmt, dann kommt das Paket von einem benachbarten ISATAP Host des selben IPv6 Subnetzes.

- Die IPv4 Adresse des Senders in der PRL enthalten ist. Dann handelt es sich bei dem Host um einen Router, der auch subnetzfremden Datenverkehr einbringen darf.

Subnetzfremder Datenverkehr von nicht-Routern wird verworfen.

Senden von Paketen Der Next-Hop eines ausgehenden Pakets über eine ISATAP Schnittstelle muss eine Unicast ISATAP Adresse sein, damit die eingebettete IPv4 Adresse als Ziel des IPv4 Pakets verwendet werden kann. Alle anderen Pakete werden verworfen. Dies geschieht völlig transparent, ein Anwender hat nicht die Möglichkeit, die Zieladresse des IPv4 Pakets zu beeinflussen. Das verhindert allerdings, dass *Router Solicitations* an `ff02::02` als Unicast an die IPv4 Adresse des Routers gesendet werden können und erzwingt, dass *Router Solicitations* ausschließlich an linklokale ISATAP Adressen gerichtet sein müssen, wie `fe80::5efe:10.0.0.1`. Das funktioniert nur zuverlässig, wenn die ISATAP Adresse des Routers korrekt gesetzt ist.

Die IPv6 Routingtabelle wird beachtet und es ist möglich, weitere Netze über ISATAP zu verbinden, vorausgesetzt die Router sind in der PRL aufgeführt.

5.5.6. isatapd aus ndisc6-0.9.9

Nach Beginn dieser Arbeit und nach Veröffentlichung der eigenen Lösung wurde am 04. August 2009 ein weiteres Userspace-Programm mit dem Namen isatapd veröffentlicht, enthalten im Paket ndisc6-0.9.9[11] von Rémi Denis-Courmont. Als Autoren des Programms werden Kwong-Sang Yin und Fred L. Templin genannt. Fred Templin ist maßgeblich an der Entwicklung von `rfc5214`[7], der Unterstützung von ISATAP im Kernel sowie der Implementierung von isatapd/ndisc6 beteiligt. Untersucht wurde Version 0.9.9 des Programms, die zum Zeitpunkt dieser Arbeit vorliegt.

[11] ndisc6-0.9.9 enthält isatapd: `http://www.remlab.net/ndisc6/`

Beschreibung des Aufrufs

```
Usage: ./isatapd [options]... [addrs]...
ISATAP Router and Prefix Discovery

[options]
  -h, --help      display this help and exit
  -d, --dev       device name (default: 'is0')
  -p, --prl       Potential Router List (PRL) name
  -q, --quiet     only print the advertised prefixes (mainly for scripts)
  -r, --retry     maximum number of attempts (default: 3)
  -V, --version   display program version and exit
  -v, --verbose   print debug messages on screen
  -w, --wait      how long to wait for a response [ms] (default: 1000)
 example:
        ./isatapd -d is1 -p isatap.example.com
        ./isatapd -v -p isatap.example.com
        ./isatapd -r 5 -p isatap.example.com
        ./isatapd -d is1 192.168.1.1 192.168.1.2 192.168.1.3
```

Features

Eine ISATAP Schnittstelle muss vor Aufruf von isatapd mit z.B. iproute2 angelegt worden sein und an das Programm übergeben werden.

- Das Programm erhält bei Aufruf eine Liste von IP Adressen oder DNS Namen, die in einer internen PRL geführt werden.
- Die Namen werden in IPv4 Adressen aufgelöst und in einer verkettete Liste gespeichert.
- Für jede IPv4 Adresse in der internen PRL wird die zugehörige linklokale IPv6 Adresse errechnet und für jede Adresse ein Timer gestartet.
- Bei Ablauf eines Timers wird für die entsprechende IPv6 Adresse eine *Router Solicitation* Nachricht über einen IPv6 Raw-Socket versendet. Dabei werden *Router Solicitations* mehrfach neu gesendet, bis ein entsprechendes *Router Advertisement* empfangen wurde, um Paketverlust vorzubeugen.
- Das Programm empfängt *Router Advertisements* und wertet die mitgeschickte *Router life-time*[12] aus. Dabei wird das Intervall, in dem *Router Solicitations* für diese Adresse gesendet werden sollen, auf höchstens 80% der *life-time* reduziert. So wird sichergestellt, dass IPv6 Adressen so lange wie möglich gültig sind.

Unzulänglichkeiten

- Der Tunnel muss von Hand erstellt werden. So sind derzeit mindestens zwei verschiedene Werkzeuge nötig, um eine funktionierende ISATAP Verbindung herzustellen.

[12] Die *Router life-time* eines RA-Pakets ist die Zeit, in der konfigurierte Adressen und Routen von diesem Router gültig sind. Nach Ablauf der Zeit entfernt ein Host die automatisch konfigurierten Routen und Adressen, wenn die Zeit nicht durch weitere RA-Pakete aufgefrischt wird.

- Die interne PRL wird noch *nicht* an den Kernel übertragen. Ohne Kenntnis der PRL wird der Kernel aber ankommende *Router Advertisements* verwerfen, da er die Absender für normale Hosts hält (siehe Kapitel 7.1.5). Auch wird kein subnetzfremder Datenverkehr angenommen, da keine Router konfiguriert sind, von denen Datenverkehr angenommen werden darf.

In Zukunft

Die derzeitige Implementierung ist vollständig unbrauchbar, es ist aber davon auszugehen, dass sie weiter verbessert wird. Fred L. Templin teilte auf Anfrage mit, die primäre Aufgabe von isatapd sei das Senden und Empfangen von von *Router Solicitations*/*Router Advertisements*. Dafür darf das Programm in einem unprivilegiertem Modus laufen.

Zum Erstellen des Tunnels und füllen der PRL soll ein weiteres Programm verwendet werden, welches über root-Rechte verfügen kann und im Anschluss isatapd starten kann. Die Programme sollen modular aufgebaut sein, sodass sie über simple Shell-Skripte ineinandergesteckt werden können.

Vorteile Der modulare Ansatz hat den Vorteil, kleine Programme für Teillösungen zu bieten. Spezielle Anforderungen lassen sich gezielter von Anwendern umsetzen.

Nachteile Durch die Modularität geht viel Automatismus verloren. Zudem wird Komplexität erhöht, da hier mehrere Instanzen eine Kopie der PRL führen müssen:

1. Der Linux Kernel
2. isatapd zum Senden von *Router Solicitations* an die Router der PRL
3. Das vereinende Skript oder Programm, das den Tunnel erstellt und die Kernel PRL füllt.

Während der Arbeit wurde versucht, aus beiden Ansätzen das Beste zu verwenden und beide Projekte zu vereinen, um eine solide Lösung zu schaffen.

5.5.7. ICMPv4 Fehler

Senden von Fehlern Linux sendet ICMPv4 Fehler an den Absender, falls das Ziel nicht erreichbar ist. Erhält der IPv4 Stack ein *proto-41*-Paket bei geladenem SIT-Modul, wird ein `ICMP_PORT_UNREACH` gesendet, bei ungeladenem Modul ein `ICMP_PROTO_UNREACH`.

Der Kernel sendet in ICMPv4 bis zu 576 Bytes der Payload zurück, was es dem Absender ermöglicht, die zugehörige IPv6 Verbindung zu identifizieren.

Verarbeiten von Fehlern Da wie in Kapitel 4.1.5 beschrieben, nur wenige Implementierungen genug ICMPv4-Payload senden, um ICMPv4 Fehler einer IPV6 Verbindung zuzuordnen, findet in Linux bei ISATAP keine Behandlung von ICMPv4 statt. Fehler auf Layer-2 werden *nicht* in die höhere IPv6 Schicht gehoben, es tritt Paketverlust auf und Anwendungen geraten bei Netzwerk-Problemen in Timeouts.

Teil III.

Implementierung

6. Pflichtenheft

Nach Abschluss dieser Arbeit soll eine funktionierende, vollständige und einfach einzurichtende ISATAP Unterstützung unter Linux vorliegen. Die existierende Implementierung wird dabei in den folgenden Punkten erweitert.

6.1. Stateless Autoconf im Kernel

Obwohl *Router Solicitations* auch aus dem Userspace gesendet werden können, erscheint eine direkte Implementierung im Linux Kernel effizienter und sinnvoller. Ein Daemon im Userspace wäre für den *Stateless Autoconf* Mechanismus nicht mehr notwendig.

Der Datenstruktur im Kernel für die Potential Router List wird ein weiterer Parameter für das Intervall hinzugefügt, in dem *Router Solicitations* gesendet werden sollen. Der Parameter soll aus dem Userspace konfigurierbar sein, Standardwert soll 900 Sekunden betragen.

Ein Ergebnis liegt in Form zweier Patches vor (Anhang A.5 und A.6), die Vorgehensweise wurde aber später zugunsten einer reinen Implementierung im Userspace wieder aufgegeben, da sie sich als komplex und nicht flexibel genug herausgestellt hat.

6.2. ICMPv4 Fehler im Kernel behandeln

Sofern möglich, soll der Kernel ICMPv4 Fehler als Layer-2 verarbeiten und in die IPv6 Schicht führen, damit auf Verbindungsfehler reagiert werden kann. Bereits ein simples `ping6 -I is0 fe80::5efe:10.0.0.3` führt üblicherweise zu Timeouts wenn der Host nicht erreichbar ist.

Ein funktionierender Patch liegt in Anhang A.9 vor, wurde aber wegen Sicherheitsbedenken nicht eingereicht.

6.3. Unterstützung für PRL in iproute2

Zwar enthält der Kernel eine gute Unterstützung für *Potential Router Lists*, sie kann aber aus dem Userspace nicht verwaltet werden. Die Werkzeugsammlung iproute2 kann ISATAP Tunnel konfigurieren, soll aber um die Fähigkeit erweitert werden, *Potential Router Lists* manuell bearbeiten zu können:

- IPv4-Adressen sollen der PRL hinzugefügt werden können, wahlweise als Default-Router oder als normaler Router.

- Alle konfigurierten Router einer ISATAP Schnittstelle sollen aufgelistet werden können.
- Router einer ISATAP Schnittstelle sollen selektiv gelöscht werden können.

Dazu wird die vorhandene API des Linux Kernel 2.6.30 verwendet.

6.4. Userspace Daemon für ISATAP

Da die manuelle Konfiguration eines ISATAP Tunnels umständlich ist, soll im Rahmen dieser Arbeit weiter ein Programm entwickelt werden, das Benutzern die Konfiguration des Tunnels vollständig abnimmt, ähnlich `isatapd` aus dem Miredo Projekt.

Der Userspace Daemon soll folgende Features bieten:

- Automatische Konfiguration eines ISATAP Tunnels anhand einer gegebenen Basis-Netzwerkschnittstelle.
  - Optional automatische Erkennung der Schnittstelle bei gegebenen ISATAP Router.
  - Optional Konfiguration von IPv4 Basisadressen für den Tunnel.
- Periodische DNS Auflösung der Router und Verwaltung der PRL im Kernel.
- Periodisches Senden von *Router Solicitations* an potentielle Router.
- Empfangen und Verarbeiten von *Router Advertisements* von Routern.
- Änderungen der Netztopologie sollen erkannt werden und bei Bedarf der Tunnel neu konfiguriert oder entfernt werden, um Konnektivität zu erhalten.

Die meisten Aufgaben können einmalig bei Start des Programms durchgeführt werden. Werden *Router Solicitations* vom Kernel gesendet, wäre der Betrieb des Programms als Daemon optional. Wenn das Userspace Programm die Aufgabe übernimmt, muss es permanent im Hintergrund laufen um die ISATAP Verbindung aufrecht zu erhalten. Dabei soll sich das Programm möglichst gut in bestehende Distributionen einbinden lassen.

7. Linux Kernel

7.1. Analyse

Dieses Kapitel beschäftigt sich genauer mit der Implementierung von ISATAP im Linux Kernel. Dabei wird die interne API und das Design erläutert, auf dem im nächsten Schritt die eigenen Projekte aufbauen können. Grundkenntnisse in C und Systemprogrammierung unter Linux sind Voraussetzung.

Versionsüberblick

Linux-2.6.25 enthält die erste Implementierung von Fred L. Templin[16], der auch als Autor von `rfc5214` genannt wird.

Linux-2.6.26 erhielt erste Unterstützung für Potential Router Lists und eine API für den Userspace[17].

Linux-2.6.30 dient als Grundlage der folgenden Analyse.

Linux-2.6.31 enthält einige vom Autor entwickelte Patches.

Für diese Arbeit wird die derzeit aktuelle Version **2.6.30** des Linuxkernels verwendet, die sich in Bezug auf ISATAP-Unterstütznug nicht nennenswert von 2.6.26 unterscheidet. Da selbst Debian Stable mittlerweile einen Linux Kernel 2.6.26 einsetzt, wurde die Implementierung in 2.6.25 nicht weiter beachtet.

Verweise auf den Quelltext Alle Verweise auf den Linux Kernel Quelltext beziehen sich auf Version **2.6.30** und sind in der Form "`Bezeichner` [*Dateiname*@*Zeilennummer*]" gekennzeichnet, wobei sich der Dateiname relativ auf das Hauptverzeichnis des Linux Kernel Quelltextes bezieht. Der Quellcode des Linux Kernels ist u.A. unter den folgenden Quellen abrufbar:

Download Komplettes Quelltextarchiv
`http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.30.tar.bz2`

Git Online Repository
`http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=tree;hb=07a2039`

LXR Suchen nach Bezeichnern in der Linux Cross Reference
`http://lxr.linux.no/linux+v2.6.30`

7.1.1. Schnittstelle zwischen Userspace und Tunnel

Eine Suche nach dem Schlüsselwort `ISATAP` im Linux Kernel Quelltext ergibt, dass das `sit`-Modul wegen der prinzipiellen Ähnlichkeit der Protokolle für alle *proto-41*-Tunnel zuständig ist. Dabei wird die Art des Tunnels intern durch Flags oder anhand der verwendeten IPv6 Adressen unterschieden.

Datenstrukturen für sit-Tunnel

Um einen *proto-41*-Tunnel zu erstellen, bereitet der Userspace eine Datenstruktur vom Typ **struct** `ip_tunnel_parm` [*include/linux/if_tunnel.h*@28] für den Kernel vor, welche die Konfiguration des zu erstellenden Tunnels enthält. Die darin verwendetete Datenstruktur **struct** `iphdr` [*include/linux/ip.h*@85] bezeichnet genau einen IPv4 Header (20 Byte).

Listing 7.1: Definition struct ip_tunnel_parm (Userspace)

```
/* include/linux/if_tunnel.h */
[...]
struct ip_tunnel_parm
{
        char                    name[IFNAMSIZ];
        int                     link;
        __be16                  i_flags;
        __be16                  o_flags;
        __be32                  i_key;
        __be32                  o_key;
        struct iphdr            iph;
};

/* SIT-mode i_flags */
#define SIT_ISATAP      0x0001
[...]
```

Listing 7.2: Definition struct iphdr (IPv4 Header)

```
/* include/linux/ip.h */
[...]
struct iphdr {
        __u8    ihl:4,
                version:4;

        __u8    tos;
        __be16  tot_len;
        __be16  id;
        __be16  frag_off;
        __u8    ttl;
        __u8    protocol;
        __sum16 check;
        __be32  saddr;
        __be32  daddr;
        /* The options start here. */
};
[...]
```

Das derzeit einzige Flag eines *sit*-Tunnels ist `SIT_ISATAP`, das für einen ISATAP Tunnel gesetzt ist. Nicht alle Strukturelemente werden verwendet, die Übrigen werden mit 0 initialisiert. Im Fall eines ISATAP Tunnels sind folgende Elemente von Bedeutung:

`name` Name der Tunnelschnittstelle, z.b. "`is0`".

`link` Index der Schnittstelle mit der der Tunnel verknüpft werden soll oder 0, wenn Datenverkehr von allen Schnittstellen empfangen werden darf.

`i_flags` Flags für die Tunnelschnittstelle. Bisher nur `SIT_ISATAP`.

`iph.ihl` Größe des Headers als Vielfaches von 32 Bit. Muss 5 sein, da $5 \cdot 4 Byte = 20 Byte$.

`iph.version` Muss 4 sein (=IPv4)

`iph.protocol` Muss für *sit*-Tunnel auf `IPPROTO_IPV6` gesetzt werden.

`iph.ttl` TTL der IPv4 Pakete, wenn sie den Tunnel verlassen. Der spezielle Wert 0 bewirkt, dass ausgehende IPv4-Pakete die TTL der IPv6-Payload erben (`inherit`). Üblicherweise wird ein statischer Wert von 64 Hops gewählt.

`iph.frag_off` Flag, ob ausgehende IPv4 Pakete fragmentiert werden dürfen oder nicht. Gleichzeitige Fragmentierung von verschiedenen Stacks (IPv6, IPv4) sollte aus Performancegründen vermieden werden.

Da IPv6 bereits fragmentiert, wird üblicherweise `IP_DF` verwendet, damit IPv4 Pakete nicht zusätzlich fragmentiert werden.

`iph.saddr` Lokale IPv4 Adresse, an die der Tunnel gebunden wird. Wenn gesetzt, wird Datenverkehr nur empfangen, wenn das IPv4 Paket an diese Adresse gerichtet ist. Wird `INADDR_ANY` (= 0) verwendet, wird Tunnelverkehr von allen lokalen IPv4 Schnittstellen empfangen.

`iph.daddr` Zielendpunkt des Tunnels. Wenn gesetzt, wird jeder ausgehende Datenverkehr an den Endpunkt weitergeleitet und einkommender Datenverkehr nur vom Endpunkt angenommen. Wenn `INADDR_ANY`, wird Datenverkehr von beliebigen Endpunkten angenommen.

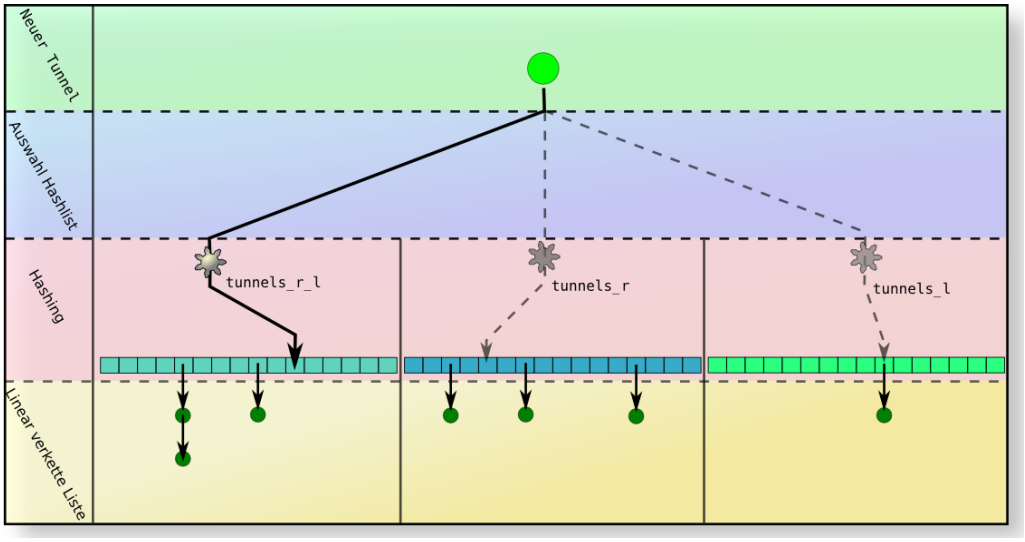
Bei 6to4 und ISATAP wird prinzipbedingt kein Zielendpunkt angegeben, da Datenverkehr auch von direkten Nachbarn angenommen werden muss und die Zieladresse ausgehender IPv4 Pakete von der IPv6 Adresse abgeleitet werden kann.

Es kann pro `saddr-daddr` Paar nur ein Tunnel existieren. Das führt dazu dass bei einem Tunnel ohne `daddr`, wie bei ISATAP, die Angabe von `saddr` zwingend notwendig ist, da das sit-Modul standardmäßig bereits einen Tunnel `sit0` mit den Endpunkten `any-any` erstellt, der sich nicht löschen lässt.

Intern werden Tunnel in verketteten Hashlisten gespeichert, zu deren Adressierung `saddr` und `daddr` herangezogen werden. So ist eine große Anzahl von Tunneln möglich, ohne dass die Performance durch ständige Iterierung einer einzigen linear verketteten Liste stark beeinträchtigt wird. Das ist besonders für Tunnelanbieter mit mehreren tausend Tunneln wie SixXS von Bedeutung. Tunnel werden in drei Hashlisten eingeteilt:

- `saddr!=0` und `daddr!=0` (`tunnels_r_l`)
- `saddr!=0` und `daddr==0` (`tunnels_l`)
- `saddr==0` und `daddr!=0` (`tunnels_r`)

Abbildung 7.1.: Hashtabellen und verkettete Listen um Tunnel zu speichern



Wird zu einem `saddr-daddr`-Paar eines empfangenen IPv4 Pakets der zugehörige Tunnel gesucht, werden der Reihe nach alle drei Hashlisten durchsucht.

Schnittstelle zum Kernel

Kommunikation zwischen Userspace und Kernel geschieht über sogenannte `ioctls`[1] verwendet, die zu einem Socket aufgerufen werden können. Der Socket baucht dabei nicht verbunden sein und dient nur als Einsprungspunkt in den Kernel für den `ioctl`.

Der `ioctl`-Handler des Sockets erwartet als Parameter immer eine Datenstruktur vom Typ **struct** `ifreq` [*include/linux/if.h*@161]. Benötigt werden hier lediglich der Zeiger auf benutzerdefinierte Daten sowie der Name der Schnittstelle auf die sich der Aufruf bezieht. Das ist hier die immer existierende Schnittstelle `sit0`, die das *sit*-Modul beim Laden anlegt. So wird `ioctl`-Aufruf vom Socket-Stack an das *sit*-Modul weitergereicht.

Der Socket kann nach dem Aufruf direkt wieder geschlossen werden. Einsprungspunkt im sit-Modul bei einem `ioctl` ist die Funktion `ipip6_tunnel_ioctl(...)` [*net/ipv6/sit.c*@771].

Listing 7.3: Beispielcode: Anlegen eines SIT-Tunnels aus dem Userspace

```
{
        struct ip_tunnel_parm tunnel;
        struct ifreq ifr;
        int fd;

        memset(&tunnel, 0, sizeof(tunnel));
        tunnel.iph.version = 4;
```

[1] ioctl: Ein `ioctl` ist ein Systemaufruf, bei dem zu einem übergebenen Dateideskriptor eine zugehörige Funktion im Kernel aufgerufen wird. So wird direkt mit einem Gerätetreiber kommuniziert um geräteabhängige Daten vom Kernel zu empfangen oder zu übergeben.
Signatur: **int** `ioctl(`**int** `d,` **int** `request, ...)`

```
8          tunnel.iph.ihl = 5;
           tunnel.iph.protocol = IPPROTO_IPV6;
10         tunnel.iph.saddr = htonl(0x0a000002);
           tunnel.iph.frag_off = htons(IP_DF);
12         tunnel.iph.ttl = 64;

14         p.i_flags |= SIT_ISATAP;
           strncpy(p.name, "mytunnel", IFNAMSIZ);
17
           /* request soll an 'sit0' gelangen */
19         strncpy(ifr.ifr_name, "sit0", IFNAMSIZ);
           ifr.ifr_ifru.ifru_data = (void*)&tunnel;
21
           fd = socket(AF_INET, SOCK_DGRAM, 0);
23         ioctl(fd, SIOCADDTUNNEL, &ifr);
           close(fd);
25 }
```

Definierte IOCTLs für Tunnel Zur Verwaltung der Tunnel behandelt das *sit*-Modul die folgenden IOCTLs, die in *include/linux/if_tunnel.h*@10 definiert sind:

`SIOCGETTUNNEL` Informationen zu einem Tunnel abfragen.

`SIOCADDTUNNEL` Einen neuen Tunnel erstellen.

`SIOCDELTUNNEL` Tunnel löschen.

`SIOCCHGTUNNEL` Einen existierenden Tunnel verändern.

7.1.2. Schnittstelle zwischen Userspace und PRL

Datenstrukturen für PRL

Aus dem Userspace werden auf Einträge in der Potential Router List über den Datentyp **struct** `ip_tunnel_prl` [*include/linux/if_tunnel.h*@42] zugegriffen:

Listing 7.4: Definition struct ip_tunnel_prl (Userspace)

```
   /* include/linux/if_tunnel.h */
2  struct ip_tunnel_prl {
           __be32                  addr;
4          __u16                   flags;
           __u16                   __reserved;
6          __u32                   datalen;
           __u32                   __reserved2;
8          /* data follows */
   };
10
   /* PRL flags */
12 #define PRL_DEFAULT             0x0001
```

Um die Definition der Struktur stabil zu halten, enthält sie reservierte Elemente, damit sie später erweitert werden kann, ohne dass bereits compilierte Programme binär inkompatibel werden.

`addr` Die IPv4 Adresse des *potentiellen Routers* in *Network Byte Order*. Der Wert 0 kann verwendet werden, um alle PRL Einträge zu bezeichnen, soweit sinnvoll.

`flags` Derzeit 0 oder `PRL_DEFAULT`, wenn der Router als potientieller Default Router behandelt werden soll.

`__reserved` Padding von 16 Bit zur späteren Verwendung.

`datalen` Größe des Speicherfeldes in Bytes. Der Kernel kann mehrere PRL Einträge mit einem Aufruf zurückgeben, wenn ein genügend großer Speicherbereich übergeben wurde (siehe Beispiellisting 7.5).

`__reserved2` Padding von 32 Bit zur späteren Verwendung.

Schnittstelle zum Kernel

Die Kommunikation mit dem Kernel geschieht auch hier über einen `ioctl` auf einem Socket. Es wird diesmal direkt die zuvor erstellte Tunnelschnittstelle `"mytunnel"` angesprochen.

Listing 7.5: Beispielcode: Hinzufügen eines PRL Eintrages aus dem Userspace

```
{
        struct ip_tunnel_prl prl;
        struct ifreq ifr;
        int fd;

        memset(&prl, 0, sizeof(prl));
        prl.addr = htonl(0xa0000001);
        prl.flags |= PRL_DEFAULT;

        strncpy(ifr.ifr_name, "mytunnel", IFNAMSIZ);
        ifr.ifr_ifru.ifru_data = (void*)&prl;
        fd = socket(AF_INET, SOCK_DGRAM, 0);
        ioctl(fd, SIOCADDPRL, &ifr);
        close(fd);
}
```

Definierte IOCTLs für Potential Router List Die folgenden IOCTLs sind definiert, um Potential Router Lists aus dem Userspace zu verwalten:

`SIOCGETPRL` Einträge der Potential Router List abfragen. Der Wert von `datalen` des ersten Elements gibt die Anzahl der Bytes an, die kopiert werden dürfen, so können mehrere Elemente mit einem Aufruf kopiert werden. Hier fand sich zudem ein *off-by-one*-Fehler, der mit Patch A.10 behoben werden konnte.

Über den Wert von `prl.addr` können gezielt Einträge abgefragt werden, ein Wert von 0 kopiert alle.

`SIOCADDPRL` Einen Eintrag in die PRL hinzufügen.

`SIOCDELPRL` Eintrag von `addr` gezielt aus der PRL löschen oder alle, wenn `addr==0`.

`SIOCCHGPRL` Einen existierenden Eintrag der PRL verändern.

Diese folgenden Kapitel beschreiben detailliert, was beim Betrieb eines Tunnels im Kernel passiert.

7.1.3. Konfiguration des Tunnels im Kernel

Anlegen des Tunnels

Während des Aufrufs von `ioctl(SIOCADDTUNNEL)` wird im Kernel zuerst über die Funktion `ipip6_tunnel_locate(...)` [*net/ipv6/sit.c*@157] der zugehörige Tunnel gesucht. Nur wenn der Tunnel mit den angegebenen Endpunkten noch nicht existiert, wird er erstellt. Unschön ist dabei, dass die Funktion auch Erfolg zurückgibt, wenn ein Tunnel mit denselben Endpunkten bereits existiert, sodass im Userspace nicht angemessen reagiert werden kann; das Problem wird mitPatch A.1 behoben.

Intern werden die Parameter des Tunnels zusammen mit der zugehörigen PRL in der kernelinternen Datenstruktur **`struct ip_tunnel`** [*include/net/ipip.h*@10] gespeichert. Die Referenz auf diese Struktur wird an der zugehörigen Stelle der entsprechenden Hashliste angehängt (Abbildung 7.1), um den Tunnel schnell finden zu können. Die Tunnel Schnittstelle erhält intern das Flag `IFF_ISATAP`, wenn es sich um einen ISATAP Tunnel handelt.

Listing 7.6: Definition struct ip_tunnel und struct ip_tunnel_prl_entry (Kernelspace)

```
/* include/net/ipip.h */
struct ip_tunnel
{
        struct ip_tunnel        *next;
        struct net_device       *dev;

        int                     recursion;
        int                     err_count;
        unsigned long           err_time;

        /* These four fields used only by GRE */
        __u32                   i_seqno;
        __u32                   o_seqno;
        int                     hlen;
        int                     mlink;

        struct ip_tunnel_parm   parms;

        struct ip_tunnel_prl_entry      *prl;
        unsigned int                    prl_count;
};

struct ip_tunnel_prl_entry
{
        struct ip_tunnel_prl_entry      *next;
        __be32                          addr;
        u16                             flags;
};
```

Durch verkettete Listen wird eine unbegrenzte Anzahl von Tunneln mit unbegrenzter Anzahl Einträgen der PRL möglich.

Konfiguration der Schnittstelle Der Kernel nimmt in der Funktion `ipv6_add_dev(...)` [*net/ipv6/addrconf.c*@335] weitere Konfigurationen vor, wenn die Schnittstelle in den Zustand `NETDEV_UP` wechselt.

Da ein ISATAP Tunnel keinen remote Endpunkt besitzt, ist der Einsatz von Multicast nicht möglich und automatische *Router Solicitations* werden deaktiviert, wenn es sich bei der Schnittstelle um einen ISATAP-Tunnel handelt (`ndev->cnf.rtr_solicits = 0`, Zeile 392). Bei einem SIT-Tunnel mit remote mit Endpunkt wird echtes Multicast zwar nicht

unterstützt, kann aber dennoch verwendet werden, da die Multicastnachricht eingebettet in Unicast an den remote Endpunkt gesendet wird. Dadurch funktionieren *Router Solicitations* eingeschränkt auch, wenn ein normaler SIT-Tunnel als Verbindung zu einem ISATAP-Netz verwendet wird (siehe Kapitel 5.5.2). Zudem wird für alle SIT-Tunnel der Einsatz von Privacy Extensions unterbunden.

Automatische Konfiguration von IPv6 Adressen

Wenn eine sit-Schnittstelle aktiviert wird, wird die Funktion `addrconf_sit_config(...)` [*net/ipv6/addrconf.c@2340*] aufgerufen, um automatisch linklokale IPv6 Adressen der Schnittstelle zu konfiguriererieren. Hier findet sich die erste Unterscheidung zwischen SIT und ISATAP.

ISATAP Die linklokale Adresse enthält in den höherwertigen 64 Bit stets das Prefix `fe80:0000:0000:0000`. Die Interface-ID (die niederwertigen 64 Bit) der Adresse berechnet sich bei ISATAP über die folgende Funktion:

Listing 7.7: Interface-ID einer ISATAP Schnittstelle berechnen

```
/* net/ipv6/addrconf.c@1521 */
[...]
int __ipv6_isatap_ifid(u8 *eui, __be32 addr)
{
        eui[0] = (ipv4_is_zeronet(addr) || ipv4_is_private_10(addr) ||
                  ipv4_is_loopback(addr) || ipv4_is_linklocal_169(addr) ||
                  ipv4_is_private_172(addr) || ipv4_is_test_192(addr) ||
                  ipv4_is_anycast_6to4(addr) || ipv4_is_private_192(addr) ||
                  ipv4_is_test_198(addr) || ipv4_is_multicast(addr) ||
                  ipv4_is_lbcast(addr)) ? 0x00 : 0x02;
        eui[1] = 0;
        eui[2] = 0x5E;
        eui[3] = 0xFE;
        memcpy(eui + 4, &addr, 4);
        return 0;
}
```

Hierbei zeigt der Parameter `u8 *eui` auf das neunte Byte einer IPv6-Adresse, ab dem die Interface-ID beginnt. Anhand des Quellcodeausschnittes ist zu erkennen, dass das *u*-Bit (hier Bit 2 von `eui[0]`) korrekt auf 1 gesetzt wird, wenn die übergebene IPv4 Adresse global eindeutig ist, andernfalls auf 0. Damit hält sich die Implementierung korrekt an die Empfehlung in `rfc5214`. Die übergebene IPv4-Adresse wird in die niederwertigsten 32 Bit der IPv6-Adresse kopiert.

Die Funktion terminiert auch dann mit Erfolg, wenn über `__be32 addr` der Wert 0 übergeben wurde, was der Fall ist, wenn der ISATAP Tunnel nicht mit einem lokalen Endpunkt konfiguriert wurde (`addr == INADDR_ANY`). Obwohl diese Situation im Normalfall nicht eintritt, wurde der Fehler in der Arbeit mit Patch A.4 korrigiert.

Aus `addr = htonl(0x8d45372d)` (=`141.69.55.45`)[2] wird durch diese Funktion:

```
                            eui[0]        eui+4
                              |            |
( FE 80 00 00 00 00 00 00 ) 02 00 5E FE 8D 45 37 2D
```

[2]Die Adresse muss in Network Byte Order (=Big Endian) im Speicher liegen.
Auf einem Little Endian System ist `8D 45 37 2D` = `0x2d37458d` = `htonl(0x8d45372d)`.

Neben der linklokalen IPv6 Adresse wird eine Route zu `fe80::/64` konfiguriert.

Andere SIT Tunnel Linklokale Adressen anderer *sit*-Tunnel werden über die Funktion `sit_add_v4_addrs(...)` [*net/ipv6/addrconf.c*@2221] konfiguriert:

- Bei lokalem und remote Endpunkt (statiche *proto-41*-Tunnel):
  Adresse `fe80::aabb:ccdd/64`
  Route zu `fe80::/64` (Linklokal)
  Route zu `ff00::/8` (Multicast)

- Bei lokalem ohne remote Endpunkt (z.B. *6to4*):
  Adresse `::aabb:ccdd/96`
  Route zu `::/96` (IPv4-kompatible IPv6 Adressen)

Ist kein lokaler Endpunkt definiert, wird für *jede* IPv4 Adresse des Hosts eine entsprechende linklokale IPv6 Adresse konfiguriert.

7.1.4. Duplicate Address Detection (DAD)

Der *Duplicate Address Detection*-Algorithmus (DAD) stellt sicher, dass eine konfigurierte IPv6 Adresse auf einem Link eindeutig ist, damit keine Adresskonflikte zwischen zwei Hosts auftreten. Dazu werden eine Reihe von *Neighbour Solicitations* an die neu zu konfigurierende IPv6 Adresse geschickt. Erhält der Sender keine Antwort, ist die Adresse frei und kann verwendet werden. IPv6-Adressen werden zunächst mit dem Flag `IFA_F_TENTATIVE` konfiguriert um zu signalisieren, dass der DAD-Algorithmus noch nicht abgeschlossen ist und die Adresse nicht verwendet werden sollte. IPv6 Adressen mit diesem Flag werden bei der Adressauswahl nicht berücksichtigt, sodass in der Regel keine Daten über diese Adresse empfangen oder gesendet werden können.

Start von Duplicate Address Detection Wird eine Schnittstelle aktiviert, startet der Kernel für alle bereits konfigurierten IPv6-Adressen, bei denen das Flag `IFA_F_TENTATIVE` gesetzt ist, den DAD-Algorithmus. Wird einer Schnittstelle explizit eine neue IPv6-Adresse hinzugefügt (z.B. bei Erfolg von *Stateless Autoconf*), wird für diese beim Hinzufügen ebenfalls der DAD-Algorithmus gestartet.

Der DAD-Algorithmus im Kernel beginnt in der Funktion `addrconf_dad_kick(...)` [*net/ipv6/addrconf.c*@2721], die für eine IPv6 Adresse zunächst nur einen Timer startet. Die Anzahl der *Neighbour Solicitations*, die versendet werden sollen bevor die Adresse als frei gewertet wird, kann pro Schnittstelle in */proc/sys/net/ipv6/conf/eth0/dad_transmits* eingestellt werden und beträgt standardmäßig 1. In Netzsegmenten mit hohem Paketverlust sollte dieser Wert erhöht werden. Das Interval zwischen zwei *NS*-Nachrichten wird über */proc/sys/net/ipv6/neigh/eth0/retrans_time* eingestellt und beträgt standardmäßig 300ms. Das bedeutet, dass eine IPv6 Adresse erst nach 300ms nach Konfiguration verfügbar ist.

Bei Adresskonflikt Wird ein *Neighbour Advertisement* (RA) empfangen, wird dieses in der Funktion `ndisc_recv_ns(...)` [*net/ipv6/ndisc.c*@717] verarbeitet. Wird für die Zieladresse des Pakets noch der DAD-Algorithmus ausgeführt, wird dieser durch die Funktion `addrconf_dad_failure(...)` [*net/ipv6/addrconf.c*@1402] abgebrochen. Der DAD-Algorithmus ist damit auf dem Link fehlgeschlagen.

Ist die betroffene Adresse eine linklokale Adresse im EUI-64-Format, wird für die Schnittstelle IPv6 komplett deaktiviert, da demnach bereits die MAC Adresse der Netzwerkkarte im Segment nicht eindeutig ist. Wurde die IPv6-Adresse automatisch hinzugefügt, wird diese gelöscht. Manuell erstellte Adressen bleiben mit gesetzem Flag `IFA_F_TENTATIVE` erhalten.

Bei Erfolg Wird zu keiner gesendeten *Neighbour Solicitation* ein zugehöriges *Neighbour Advertisement* empfangen, wird das `IFA_F_TENTATIVE`-Flag der Adresse entfernt und die Adresse damit als gültig markiert.

Bei Erfolg des DAD-Algorithmus werden im Anschluss von der konfigurierten IPv6 Adresse *Router Solicitations* gesendet, falls alle folgenden Bedingungen zutreffen:

- Die IPv6 Adresse, für die DAD erfolgreich durchgeführt wurde, muss linklokal sein (`fe80::/64`).
- IPv6 Forwarding für diese Schnittstelle muss deaktiviert sein, d.h. es darf sich nicht um einen Router handeln.
- Die Anzahl zu sendender *Router Solicitations* `idev->cnf.rtr_solicits > 0`. Bei einem ISATAP Tunnel wird der Wert im Kernel bei Konfiguration der Schnittstelle auf 0 gesetzt um automatische *Router Solicitations* zu verhindern.

Optimistic DAD `rfc4429`[18] beschreibt eine *optimistische* Erweiterung zu DAD, bei der konfigurierte IPv6 Adressen bereits für Datenverkehr benutzt werden können, auch wenn der DAD-Algorithmus noch läuft. Das Verfahren beschleunigt DAD und wird in Situationen empfohlen, bei denen der Erfolg von DAD wesentlich wahrscheinlicher ist, als der Miserfolg. Beispiele, bei denen die Eindeutigkeit der Adresse sehr wahrscheinlich ist:

- Automatisch konfigurierte linklokale IPv6 Adressen, bei denen eine eindeutige MAC Adresse verwendet wird.
- IPv6 Adressen die als Antwort eines *Router Advertisements* auf Basis der linklokalen Adresse generiert werden, da diese bereits auf Eindeutigkeit überprüft wurde.
- Temporären Adressen, die über die *Privacy Extensions* konfiguriert wurden, da die Wahrscheinlichkeit, dass zwei Rechner dieselbe zufällige Adresse konfigurieren, extrem gering ist.
- Adressen, die über (stateful) DHCPv6 bezogen werden, da der Server bereits die Eindeutigkeit der Adresse sicherstellt.

Eine *optimistische* IPv6 Adresse darf nur in Situationen verwendet werden, in denen der *Neighbour Cache* eines Nachbarn nicht verändert wird, um bestehende Kommunikation im Fall einer doppelten Adresse nicht zu beeinträchtigen.

Ausnahmen von DAD In bestimmten Fällen ist der DAD-Algorithmus nicht möglich oder nicht nötig und wird übersprungen:

- Bei Adressen, für die zuvor bereits DAD durchgeführt wurde.
- Bei loopback-Schnittstellen.
- Adressen, bei denen das Flag `IFA_F_NODAD` manuell gesetzt wurde:

  ```
  # ip addr add 3ffe::1234/64 dev eth0 nodad
  ```

- Schnittstellen, bei denen das Flag `IFF_NOARP` gesetzt ist, welches anzeigt, dass auf dem Netzsegment ARP/NS nicht verfügbar sind. Das ist u.A. bei Tunneln der Fall, die prinzipiell kein Multicast unterstützen. Für alle *sit*-Tunnel wird das Flag in der Funktion `ipip6_tunnel_setup(...)` [*net/ipv6/sit.c*@938] explizit gesetzt, sodass für *sit*-Tunnel der DAD-Algorithmus grundsätzlich übersprungen wird.

Ist die bei ISATAP verwendete IPv4 Adresse im Netzsegment eindeutig, so ist auch die abgeleitete IPv6 Adresse eindeutig und DAD braucht nicht durchgeführt werden. Automatische *Router Solicitations* werden nicht gesendet, da die Verbindungsschickt kein Multicast unterstützt.

7.1.5. Stateless Autoconf

Wurde nach DAD die IPv6 Adresse auf dem Link als eindeutig eingestuft, wird automatisch der *Stateless Autoconf*-Algorithmus gestartet, der vom Prinzip dem DAD-Algorithmus ähnelt. Dabei werden über einen Timer solange *Router Solicitations* an die Multicastadresse `ff02::2` (*ip6-allrouters*) gesendet, bis ein zugehöriges *Router Advertisement* empfangen wurde. Die Anzahl der zu sendenden *RS*-Nachrichten kann über */proc/sys/net/ipv6/conf/eth0/router_solicitations* konfiguriert werden; Standardwert ist 3. Der Linux Kernel ruft bei Ablauf des Timers die Funktion `addrconf_rs_timer(...)` [*net/ipv6/addrconf.c*@2682] aus. Der Timer beendet sich, wenn:

- bereits die maximale Anzahl an *Router Solicitations* gesendet wurden, also kein Router auf dem Link vorhanden ist,
- oder das Flag `IF_RA_RCVD` der Schnittstelle gesetzt wurde, als Zeichen dass ein *Router Advertisement* empfangen wurde.

Der *Router Discovery*-Algorithmus wird nur einmalig bei Aktivierung der Schnittstelle durchgeführt. Es ist die Aufgabe des Routers, mit periodischen *Router Advertisements* die Lebensdauer der konfigurierten IPv6 Adressen aufzufrischen.

Senden von Router Solicitations

"*Neighbor Discovery for IPv6*", detailliert beschrieben in `rfc4861, Abschnitt 4.1`[19], wird mit normalen ICMPv6 Paketen durchgeführt, bei denen spezielle Optionen gesetzt sind.

Über die Funktion `ndisc_send_rs(dev, saddr, daddr)` [*net/ipv6/ndisc.c*@627] wird eine IPv6 *Router Solicitation*-Nachricht an die mit `saddr` bezeichnete IPv6 Adresse erstellt und gesendet. Dabei wird die Funktion `ndisc_build_skb(...)` [*net/ipv6/ndisc.c*@440] verwendet, um ein entsprechendes ICMPv6-Paket zu erstellen.

Die Routine füllt die Werte des IPv6- und ICMPv6-Headers. Die darunterliegende Layer-2-Schicht ist völlig transparent. Das Paket wird an dieselbe Zieladresse gesendet, die auch in das ICMPv6-Paket geschrieben wurde. Im Falle von ISATAP bedeutet das, dass *Router Solicitation* nicht an die Multicast Adresse `ff02::2` gesendet werden können, da die ISATAP Layer-2 Schicht die Unicast Zieladresse des Pakets nicht mehr aus der Zieladresse ableiten kann. Derzeit ist es bei ISATAP nur möglich, *Router Solicitations* direkt an die IPv6 linklokale Unicast ISATAP Adresse zu versenden.

Empfang von Router Solicitations

Der Kernel empfängt *Router Solicitations* zunächst in der Funktion `ndisc_recv_rs(...)` [*net/ipv6/ndisc.c*@1005], in der lediglich der Eintrag im *Neighbor Discovery Cache* aktualisiert wird.

Üblicherweise werden *Router Solicitations* im Userspace über Programme wie radvd[3] abgefangen. Diese verarbeiten die Nachricht und versenden entsprechende *Router Advertisements*. Da hierfür keine weitere Behandlung im Kernel notwendig ist, wird auf den Vorgang an dieser Stelle nicht weiter eingegangen.

Empfang von Router Advertisements

Die Funktion `ndisc_router_discovery(...)` [*net/ipv6/ndisc.c*@1108] prüft und verarbeitet empfangene *Router Advertisements*. Ist das Paket gültig, wird für die Schnittstelle das Flag `IF_RA_RCVD` gesetzt, um den *RS*-Timer zu beenden. Die Funktion verarbeitet Optionen und konfiguriert Routen, die über das *Router Advertisement* mitgesendet werden, sofern der Host nicht als Router arbeitet und automatische Konfiguration von IPv6 aktiviert ist.

Das empfangene Prefix wird anschließend an die Funktion `addrconf_prefix_rcv(...)` [*net/ipv6/addrconf.c*@1746] übergeben, die für die Schnittstelle automatische eine weitere IPv6 Adresse für das Prefix und eine Standard Route auf den Router konfiguriert. Für die neue Adresse wird der DAD-Algorithmus ausgeführt. Da es sich bei der neuen Adresse nicht um eine linklokale Adresse handeln kann, werden hierfür keine weiteren *Router Advertisements* gesendet.

[3] Router Advertisement Daemon: `http://www.litech.org/radvd`

Prüfung der Quelladresse `rfc5214, Abschnitt 8.3.3` empfiehlt, bei Empfang von *Router Advertisements* auf einer ISATAP Schnittstelle zusätzlich die Quelladresse zu überprüfen um den Empfang nur von möglichen Routern zuzulassen. Da ISATAP Router im voraus bekannt sind, muss die Quelladresse eine linklokale ISATAP Adresse sein, bei der die IPv4 Adresse eines potentiellen Routers eingebettet ist. *Router Advertisements* von anderen Quellen sollen zur Sicherheit verworfen werden.

Um diese Entscheidung treffen zu können, werden zusätzliche Informationen vom Linklayer benötigt, ob es sich bei dem Sender der Nachricht um einen Host oder einen Router handelt.

Informationen vom Linklayer Der Linklayer bei ISATAP (=IPv4) besitzt bei ISATAP Informationen über die Art eines IPv6 Hosts. Ist die zugehörige IPv4 Adresse in der PRL des Tunnels aufgeführt, handelt es sich um einen Router, andernfalls um einen normalen Host. Diese zusätzliche Information wird in der Variablen `skb->ndisc_nodetype` hinterlegt und von der Funktion `isatap_chksrc(...)` [*net/ipv6/sit.c*@361] (Listing 7.8) auf einen der folgenden Werte gesetzt, um den Sender eines IPv6 Pakets zu klassifizieren (definiert in *include/net/ndisc.h*@14):

```
/*
 * Router type: cross-layer information from link-layer to
 * IPv6 layer reported by certain link types (e.g., RFC4214).
 */
#define NDISC_NODETYPE_UNSPEC       0     /* unspecified (default) */
#define NDISC_NODETYPE_HOST         1     /* host or unauthorized router */
#define NDISC_NODETYPE_NODEFAULT    2     /* non-default router */
#define NDISC_NODETYPE_DEFAULT      3     /* default router */
```

Die Funktion `ndisc_router_discovery(...)` [*net/ipv6/ndisc.c*@1134] wertet diese Information aus, um *Router Advertisements* von nicht-Routern zu verwerfen:

```
[...]
/* ndisc_router_discovery(), net/ipv6/ndisc.c@1134 */
#ifdef CONFIG_IPV6_NDISC_NODETYPE
        if (skb->ndisc_nodetype == NDISC_NODETYPE_HOST) {
                ND_PRINTK2(KERN_WARNING
                           "ICMPv6 RA: from host or unauthorized router\n");
                return;
        }
#endif

[...]
/* ndisc_router_discovery(), net/ipv6/ndisc.c@1165 */
#ifdef CONFIG_IPV6_NDISC_NODETYPE
        /* skip link-specific parameters from interior routers */
        if (skb->ndisc_nodetype == NDISC_NODETYPE_NODEFAULT)
                goto skip_linkparms;
#endif
```

7.1.6. Empfang von Datenverkehr

Installation der Hooks Aus Anwendungssicht ist ein *sit*-Tunnel eine normale Netzwerkschnittstelle und wird als solche transparent verwendet. Das tunnel4-Modul registriert beim Laden in der Funktion `tunnel4_init` [*net/ipv4/tunnel4.c*@151] sowohl das Protokoll `IPPROTO_IPIP` (=4) als auch das Protokoll `IPPROTO_IPV6` (=41). Das Modul dient als Art Container für Tunneltreiber und vermittelt empfangenen Datenverkehr an Tunnelmodule, die sich registriert haben. Nicht behandelte Datenpakete werden an den nächsten

Tunneltreiber weitergereicht, sodass mehrere unterschiedliche Tunnelarten mit demselben Protokoll-Typ möglich sind.

Das sit-Modul registriert sich in der Funktion `sit_init` [*net/ipv6/sit.c*@1073] als Tunneltreiber für IPv6 Datenverkehr (*proto-41*). Bei Empfang eines entsprechenden Datenpakets wird die Methode `ipip6_rcv` [*net/ipv6/sit.c*@473] aufgerufen, in der schließlich Datenverkehr vom sit-Modul verarbeitet werden kann.

Entpacken des Pakets Die Methode erhält einen Zeiger auf den IPv4-Header und sucht den ersten Tunnel, der zum saddr-daddr Paar passt. Wird kein aktiver Tunnel gefunden, sendet der Kernel ein `ICMP_DEST_UNREACH`/`ICMP_PORT_UNREACH` über IPv4 an den Absender zurück.

Im Anschluß entfernt die Methode den IPv4 Header und speist den resultierenden Puffer, der jetzt den IPv6 Header und die Folgedaten erhält, transparent in den IPv6-Stack ein. Der Stack behandelt das Paket als neu und reicht es an die darüberliegenden Schichten bis zum Userspace weiter.

Überprüfung des Absenders bei ISATAP Hat der Tunnel das `IFF_ISATAP`-Flag gesetzt, wird die Adresse des Absenders zusätzlich überprüft, bevor das entpackte Paket wieder in den Netzwerkstack gelangt.

Listing 7.8: Überprüfung des Absenders eines ISATAP Datenrahmens

```
/* net/ipv6/sit.c@361 */
2  static int
   isatap_chksrc(struct sk_buff *skb, struct iphdr *iph, struct ip_tunnel *t)
4  {
           struct ip_tunnel_prl_entry *p;
6          int ok = 1;

8          read_lock(&ipip6_lock);
           p = __ipip6_tunnel_locate_prl(t, iph->saddr);
10         if (p) {
                   if (p->flags & PRL_DEFAULT)
12                         skb->ndisc_nodetype = NDISC_NODETYPE_DEFAULT;
                   else
14                         skb->ndisc_nodetype = NDISC_NODETYPE_NODEFAULT;
           } else {
16                 struct in6_addr *addr6 = &ipv6_hdr(skb)->saddr;
                   if (ipv6_addr_is_isatap(addr6) &&
18                     (addr6->s6_addr32[3] == iph->saddr) &&
                       ipv6_chk_prefix(addr6, t->dev))
20                         skb->ndisc_nodetype = NDISC_NODETYPE_HOST;
                   else
22                         ok = 0;
           }
24         read_unlock(&ipip6_lock);
           return ok;
26 }
```

Stammt das Paket von einem Router, findet keine weitere Überprüfung statt. Kommt das Paket dagegen von einem IPv4 Host, müssen für die Quelle alle folgenden Bedingungen zutreffen, damit das Paket angenommen wird:

Zeile 17 `ipv6_addr_is_isatap(addr6)` [*include/net/addrconf.h*@253] ist erfüllt, wenn die IPv6 Zieladresse eine ISATAP Adresse ist.

Listing 7.9: Überprüfung auf IPv6 ISATAP Adresse

```
static inline int ipv6_addr_is_isatap(const struct in6_addr *addr)
{
        return ((addr->s6_addr32[2] | htonl(0x02000000))
                == htonl(0x02005EFE));
}
```

Diese strenge Überprüfung lässt von den ersten 4 Byte der Interface-ID lediglich das u-Bit variabel und ist damit kompatibel zum Gegenstück `__ipv6_isatap_ifid()` (Siehe Listing 7.7).

Zeile 18 `addr6->s6_addr32[3] == iph->saddr`, d.h. die in der IPv6 Absenderadresse eingebettete IPv4 Adresse muss mit der IPv4 Absenderadresse übereinstimmen.

Zeile 19 `ipv6_chk_prefix(addr6, t->dev)` [*net/ipv6/addrconf.c*@1325] ist erfüllt, wenn die Tunnelschnittstelle mindestens eine IPv6 Adresse besitzt, dessen Prefix mit der zu vergleichenden IPv6 Adresse übereinstimmt, d.h. die ersten 64 Bit der Adressen übereinstimmen. Dann gehört die IPv6 Adresse einem direkten Nachbarn des Hosts.

Gehört die ISATAP Adresse keinem lokal konfiguriertem Subnetz an, müssen die Datenrahmen wie IPv6 Verkehr behandelt werden, der über einen Router eingeliefert werden muss.

Von Routern wird demnach jeder Datenverkehr angenommen, von benachbarten Hosts lediglich Datenverkehr, wenn diese sich auf demselben Link befinden.

Die Zieladresse des Datenpakets wird nicht untersucht. Der IPv6-Stack wird das IPv6 Paket neu verarbeiten und unabhängig des ISATAP-Tunnels untersuchen. Das ist nicht weiter kritisch, da das Paket nur angenommen werden kann, wenn der Host die Zieladresse auch konfiguriert hat, ansonsten wird das Paket verworfen.

Empfang von ICMPv4 Nachrichten

Ein ICMPv4 Paket enthält laut `rfc792, Seite 4` als Payload den IPv4-Header und mindestens 64 Bit des originalen Pakets. Im Fall von TCP oder UDP sind das die ersten 8 Byte des TCP/UDP Headers, bei `IPPROTO_IPV6` die ersten 8 Byte des IPv6-Headers.

Empfängt der Kernel eine ICMPv4 Fehlernachricht, wird anhand des Protokoll-Feldes im IP-Header das Paket an das entsprechende Modul weitergereicht, bei `IPPROTO_IPV6` ist das das Modul tunnel4. Dieses reicht ICMPv4 Meldungen an alle registrierten Tunnelmodule weiter, sodass das sit-Modul das Paket in der Funktion `ipip6_err()` [*net/ipv6/sit.c*@405] bearbeiten kann. Dort finden sich die Zeilen:

> /* All the routers (except for Linux) return only 8 bytes of packet payload.
> It means, that precise relaying of ICMP in the real Internet is absolutely
> infeasible. */

Die aktuelle Implementierung im Linux Kernel verarbeitet momentan lediglich zwei ICMP Fehler: Destination Unreachable und Time Exceeded/TTL Expired in Transit. Der

Autor des Codes ging davon aus, dass die meisten Router nur 8 Byte Payload des Quellpakets zurücksenden, was nicht ausreicht, um die Verbindung zu identifizieren, auf die sich das Paket bezieht.

Statische Tunnel Eine ICMP Nachricht, die für einen Tunnel empfangen wird, der einen remote Endpunkt hat, kann nur auf dem Pfad zwischen Host und Endpunkt gesendet worden sein und muss sich auf den ganzen Tunnel beziehen; Eine Nachricht kann demnach als Zeichen gedeutet werden, dass der gesamte Tunnel gestört ist.

Ausnahme ist ein Time Exceeded, wenn die TTL vom Tunnel auf *vererben* gestellt ist, wobei das IPv4 Paket die TTL vom eingepacktem IPv6 Paket erhält. Gerät die TTL des IPv4 Pakets auf 0, wäre bei nativem Betrieb ebenso die TTL vom IPv6 Paket auf 0 verringert worden, was nicht als Layer-2 Fehler des Tunnels gedeutet werden darf. In diesem Fall wird das ICMP Paket verworfen.

Auch bei statischen Tunneln ist das Verhalten möglich, höhere Schichten über Layer-2 Fehlern zu benachrichtigen (sofern die empfangene Payload des ICMP Pakets das ermöglicht), statt den ganzen Tunnel als fehlerhaft zu markieren.

Im Fehlerfall Im Falle eines korrekten ICMP Fehlers für den statischen Tunnel wird der ganze Tunnel für 30 Sekunden als *fehlerhaft* markiert. Pakete können weiterhin regulär gesendet und empfangen werden, aber zusätzlich werden während des Fehlerzeitraums ICMPv6 Fehler an die höhere Schicht gereicht, als Zeichen, dass der *Link* gestört ist. Anwendungen können so reagieren.

Dynamische Tunnel Besitzt der Tunnel keinen festen Endpunkt (6to4 und ISATAP), sind einzelne ICMPv4 Nachrichten kein Zeichen für einen Fehler des gesamten Tunnels. Den ganzen Tunnel auf fehlerhaft zu setzen, wäre fatal.

ICMP Fehler für Tunnel ohne Endpunkt werden grundsätzlich verworfen und nicht weiter behandelt.

I_WISH_WORLD_WERE_PERFECT

Geht man in der Zeit etwas zurück, findet man Code im Linux Kernel, der das gewünschte Verhalten bereits implementierte. Er wurde zwischen linux-2.6.25 und linux-2.6.26 ersatzlos aus dem Kernel entfernt[4], da er direkte Fehler enthielt und nur wenig getestet war, da er nur aktiviert wurde, wenn die Option `I_WISH_WORLD_WERE_PERFECT` gesetzt war. Im Standardfall war das zuvor bereits geschriebene Verhalten aktiv. Der entfernte Code sollte ICMPv4 Nachrichten wie folgt behandeln:

1. Wenn die ICMPv4 Payload weniger als einen IPv6 Header (=40 Byte) enthält, wird das Paket verworfen.

[4] net: The world is not perfect patch:
`http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=071f92d05`

2. Der ICMPv4 Code wird in einen entsprechenden ICMPv6 Code übersetzt oder, wenn es keine sinnvolle Übersetzung gibt, verworfen.

3. Über die Routingtabelle wird der ausgehende Tunnel gesucht und das Paket im Fehlerfall verworfen.

4. Das eingebettete ICMPv6 Paket wird über die gefundene Schnittstelle des Tunnels als ICMPv6 versendet und gelangt so in höhere Netzwerk-Schichten.

Diese alte Implementierung verwirft alle Pakete, die nicht genug Payload haben, während die neue Implementierung genau diese Pakete nicht beachtet und stattdessen nur Pakete behandelt, die zu statischen Tunneln gehören. Kapitel 7.2.2 beschäftigt sich mit dem Versuch, die beiden Ansätze zu vereinen, sodass auf ISATAP Fehler angemessen reagiert werden kann.

7.1.7. Senden von Datenverkehr

Ermitteln der IPv4 Zieladresse eines Pakets

Beim Versenden von Datenverkehr über einen SIT-Tunnels sind weitere Besonderheiten der einzelnen Tunnelarten zu beachten. Die Funktion `ipip6_tunnel_xmit` [*net/ipv6/sit.c@538*] wird aufgerufen, wenn IPv6 Daten zum Versenden über den Tunnel vorliegen. Das Modul muss das IPv6 Paket in ein entsprechendes IPv4 Paket verpacken, und über IPv4 an den Tunnel Endpunkt versenden, wobei die verschiedenen Adressierungsarten und Tunnelarten, wie *static*, *6to4* oder *ISATAP*, berücksichtigt werden müssen.

Destination Jedes IPv6 Paket enthält im IPv6-Header eine Zieladresse, an die das Paket schlussendlich gelangen soll (über Multicast oder Unicast).

Next-Hop Der *Next-Hop* eines IPv6 Pakets ist ein direkter Nachbar des Hosts, der das Paket einen Schritt näher ans Ziel routet, und wird direkt über die Routingtabelle ermittelt:

- Existiert eine direkte Route für das Subnetz der Zieladresse über eine Schnittstelle, ist die Zieladresse ein direkter Nachbar. Der *Next-Hop* des Pakets ist dann die Zieladresse selbst.
- Existiert keine direkte Route, ist der *Next-Hop* üblicherweise ein Router für ein Subnetz, oder ein Standard-Router. Der Next-Hop des Pakets ist der Router, der aus der Routingtabelle ermittelt wird.
- Existiert keine Route, muss das Paket mit `Route Unreachable` verworfen werden.

Der *Next-Hop* ist damit der erste direkte Nachbar auf der Route zum Ziel oder das Ziel selbst.

Das *sit*-Modul ermittelt aus dem *Next-Hop* eines IPv6 Pakets die Zieladresse des IPv4 Pakets. Erst wenn das IPv4 Paket in den IPv4 Stack gelangt, wird dessen *Next-Hop* durch die IPv4 Routingtabellen bestimmt.

Statische Tunnel

Ist für einen Tunnel ein IPv4 remote Endpunkt konfiguriert (statischer *6in4*-Tunnel), wird ausschließlich diese IPv4 Adresse als Zieladresse für ausgehende *proto-41*-Pakete verwendet. Die IPv6 Routingtabelle und der IPv6 *Next-Hop* werden nicht beachtet.

```
struct iphdr  *tiph = &tunnel->parms.iph;
/* [...] */
/* net/ipv6/sit.c@549 */
__be32 dst = tiph->daddr
```

ISATAP

Ist das `IFF_ISATAP`-Flag des Tunnels gesetzt, wird aus der IPv6-Adresse des *Next-Hop* die IPv4-Adresse abgeleitet, ein direkter Nachbar ist dafür zwingend notwendig. Kann aufgrund unvollständig gesetzter Routen kein entsprechender *Next-Hop* gefunden werden, wird das Paket mit einem *Transmission*-Fehler (*TX*) verworfen.

Listing 7.10: Ermittlung der IPv4-Adresse aus dem IPv6 Next-Hop bei ISATAP

```
        /* [...] */
        /* net/ipv6/sit.c@569 */
        if (neigh == NULL) {
                if (net_ratelimit())
                        printk(KERN_DEBUG "sit: nexthop == NULL\n");
                goto tx_error;
        }
        addr6 = (struct in6_addr*)&neigh->primary_key;
        addr_type = ipv6_addr_type(addr6);

        if ((addr_type & IPV6_ADDR_UNICAST) &&
                ipv6_addr_is_isatap(addr6))
                dst = addr6->s6_addr32[3];
        else
                goto tx_error;
        /* [...] */
```

Der *Next-Hop* einer ISATAP Schnittstelle wird immer ein weiterer ISATAP Host sein. Nur wenn der ermittelte IPv6 *Next-Hop* über Unicast erreichbar ist, und es sich dabei um eine gültige ISATAP-Adresse handelt (siehe Listing 7.9), kann gemäß der Adresskonvention die eingebettete IPv4 Adresse extrahiert und als Ziel für das *proto-41*-Paket verwendet werden. Alle anderen Pakete weisen auf ein Konfigurationsproblem hin und können verworfen werden.

Gemäß dieser Implementierung kann ein Multicast *Next-Hop* keiner IPv4 Adresse zugeordnet werden und ist nicht zustellbar.

Beim Senden wird zudem nicht zwischen potentiellen Routern und Hosts unterschieden, da hierfür allein die IPv6 Routingtabelle zuständig ist.

6to4

Handelt es sich nicht um einen statischen oder einen ISATAP Tunnel, wird versucht, die IPv4 Adresse aus einer 6to4 Zieladresse abzuleiten. Da zwei 6to4-Hosts sich gegenseitig direkt erreichen können sollen, kann die IPv4-Zieladresse direkt aus der 6to4-Adresse

abgelesen werden, ähnlich wie bei ISATAP. Der 6to4-Adresse `2002:aabb:ccdd::/48` wird somit die IPv4-Zieladresse `aa.bb.cc.dd` zugeordnet.

Listing 7.11: Ermittlung der IPv4-Adresse eines 6to4 Ziels

```
/* net/ipv6/sit.c@519 */
/* Returns the embedded IPv4 address if the IPv6 address
comes from 6to4 (RFC 3056) addr space */

static inline __be32 try_6to4(struct in6_addr *v6dst)
{
        __be32 dst = 0;

        if (v6dst->s6_addr16[0] == htons(0x2002)) {
                /* 6to4 v6 addr has 16 bits prefix, 32 v4addr, 16 SLA, ... */
                memcpy(&dst, &v6dst->s6_addr16[1], 4);
        }
        return dst;
}

/* [...] */
/* net/ipv6/sit.c@585 */
        if (!dst)
                dst = try_6to4(&iph6->daddr);
```

Während die Ermittlung der IPv4 Adresse bei einem Prefix `2002::/16` trivial ist, fällt auf, dass beim Aufruf der Funktion über `dst = try_6to4(&iph6->daddr)` **nicht** der *Next-Hop* des IPv6 Pakets übergeben wird, der sich aus der Routingtabelle ableitet, sondern die direkte Zieladresse des IPv6 Pakets. Zwar wird die Routingtabelle vom Kernel verwendet, um die Schnittstelle für ausgehende Pakete zu finden, aber spezielle Routen über dieselbe Schnittstelle werden ignoriert. Beispiel Routingtabelle:

Listing 7.12: Spezielle Routen auf einer 6to4 Schnittstelle

```
# ip -6 route show
2002::/16 dev 6to4  metric 256
2002:a00::/32 via ::192.168.6.6 dev 6to4  metric 1
default via 2002:c058:6301:: dev 6to4  metric 10
default via ::192.88.99.1 dev 6to4  metric 1200
```

`2002::/16` Diese Route hat keinen Effekt. Das Paket wird bereits über die Default-Route über die Schnittstelle `6to4` gesendet und da obiger Code den *Next-Hop* nicht berücksichtigt, besteht kein Unterschied zu der Default Route.

`2002:a00::/32` Ein Paket an die Adresse `2002:a00:1::42` dieses Bereichs wird über die Schnittstelle `6to4` gesendet, aber statt an den *Next-Hop* von `::192.168.6.6`, wird das Paket direkt an die eingebettete Adresse `10.0.0.1` gesendet, da die Zieladresse aus dem Prefix `2002::/16` stammt; die spezielle Route sollte dies eigentlich verhindern.

`default via 2002:c058:6301::` Diese Route *schlägt* sogar *fehl*! Der Router ist zwar ein direkter Nachbar auf der Schnittstelle, aber ein Paket an ein Ziel außerhalb von `2002::/16` wird nicht an den *Next-Hop* `2002:c058:6301::` geschickt, da die Zieladresse nicht in den 6to4-Bereich fällt.

Das Paket rutscht durch den Vergleich auf `2002::/16` und landet im darauf folgendem Code-Block (IPv4-kompatible Adresse, nächster Abschnitt), in dem alle ausgehenden Pakete für diese Route schließlich verworfen werden.

Um sicherzustellen, dass sich zwei 6to4-Hosts[5] immer direkt erreichen, reicht es im Normalfall aus, explizit eine Route nach `2002::/16` auf die Tunnel Schnittstelle zu setzen, durch die alle Ziele dieses Adressbereichs direkte Nachbarn werden.

Ist die Zieladresse keine 6to4-Adresse, wird üblicherweise die IPv4-kompatible Anycast Adresse `::192.88.99.1` als *Next-Hop* verwendet, die im nächsten Abschnitt behandelt wird.

Der in `rfc3068, Abschnitt 2.5`[20] empfohlene Default Router `2002:c058:6301::` kann unter Linux aufgrund des Fehlers *nicht* verwendet werden. Der Fehler wurde in Anhang A.7 korrigiert und eingereicht, allerdings wurde die Meinung des Autors nicht geteilt und der Patch von den Maintainern wegen Geringfügigkeit nicht weiter untersucht.

IPv4-kompatible IPv6 Adressen

Ist auch *6to4* nicht erfolgreich, wird versucht, den *Next-Hop* als IPv4-kompatible Adresse zu interpretieren, bei der die IPv4 Adresse in den niederwertigsten 32 Bit der IPv6 Adresse eingebettet ist und die höheren Bits alle auf 0 gesetzt sind.

```
/* [...] */
/* net/ipv6/sit.c@608 */
if ((addr_type & IPV6_ADDR_COMPATv4) == 0)
        goto tx_error_icmp;

dst = addr6->s6_addr32[3];
/* [...] */
```

Als Zieladresse des IPv4 Pakets wird somit die eingebettete Adresse extrahiert. Der Code wird ausgeführt, wenn alle vorherigen Bedingungen nicht zutreffen und verwirft schließlich das Paket, wenn das Ziel auch keine IPv4-kompatible Adresse ist.

Die standard Route einer 6to4 Schnittstelle ist üblicherweise die IPv4-kompatible IPv6 Adresse `::192.88.99.1`, die durch obigen Code in die IPv4 Adresse `192.88.99.1` übersetzt wird.

Path MTU und Fragmentierung

Bei einem *sit*-Tunnel können unabhängig voneinander zwei Arten von Fragmentierung auftreten: IPv4- und IPv6 Fragmentierung. Beide Transportschichten haben eine MTU[6], woraus sich abhängig der Größe der zu transportierenden Payload folgende Fälle ergeben können:

[5] Mit 6to4-Host wird hier ein Host bezeichnet, der eine IPv4 Adresse und eine Tunnelschnittstelle konfiguriert hat, sowie eine Route zu `::192.88.99.1`. Native IPv6-Hosts mit einer Adresse aus dem Subnetz `2002:aabb:ccdd::/48` fallen nicht darunter.

[6] MTU (Maximum Transmission Unit): Maximale Größe der Nutzdaten für ein Netzwerksegment, die unfragmentiert verschickt werden können. Die MTU einer Ethernet Schnittstelle beträgt üblicherweise 1500 Byte, für einen IP-in-IP-Tunnel 1480 Byte, da der äußere IP Header abgezogen werden muss

- Payload $\leq$ MTU$_{IPv6}$ $\Rightarrow$ Keine IPv6 Fragmentierung
- Payload $>$ MTU$_{IPv6}$ $\Rightarrow$ `ICMPV6_PKG_TOOBIG` zurück, IPv6 Fragmentierung
- MTU$_{IPv6}$ $\leq$ MTU$_{IPv4}$ $\Rightarrow$ Nie IPv4 Fragmentierung nötig
- MTU$_{IPv6}$ $>$ MTU$_{IPv4}$ $\Rightarrow$ Abhängig von Payload ist IPv4 Fragmentierung nötig

Aus Performacegründen sollte der Fall

$$\text{Payload} > \text{MTU}_{IPv6} > \text{MTU}_{IPv4}$$

vermieden werden, das hier Fragmentierung gleichzeitig bei IPv6 als auch bei IPv4 stattfindet. IPv6 Fragmentierung kann der IPv4 Fragmentierung vorgezogen werden, indem darauf geachtet wird, dass stets

$$\text{MTU}_{IPv6} \leq \text{MTU}_{IPv4}$$

IP-Fragmentierung Ein IPv4 Paket wird fragmentiert, wenn die MTU der Schnittstelle nicht ausreicht, das Paket in einem Stück zu übertragen. Dabei wird im IP Header das *MF*-Flag (=*More Fragments*) gesetzt, als Zeichen dass weitere Fragmente folgen. Das Fragment-Offset im Header wird auf ein Vielfaches von 8 Byte gesetzt. Bis auf das letzte Paket müssen alle Fragmente Vielfache von 8 Byte übertragen; jedes Paket muss den IP Header enthalten. Da das Offset eine Länge von 13 Bit hat, ist der größte Wert $(2^{13} - 1) \cdot 8$ Byte $= 65528$ Byte. Einschließlich der Payload des letzten Pakets lassen sich alle IP Pakete vollständig fragmentieren, die die maximale Größe von 65535 nicht überschreiten.

Bei IPv4 findet Fragmentierung immer auf dem Router statt, dessen Link eine zu kleine MTU aufweist. Wenn der Sender des Pakets IPv4 Fragmentierung verhindern möchte, muss im original IP Header das *DF*-Flag (=*Don't Fragment*) gesetzt werden. Ist dann Fragmentierung nötig aber das Bit gesetzt, schickt der Router eine ICMPv4-*fragmentation-needed* Nachricht zurück und verwirft das Paket. Der sendende Host sollte diese ICMPv4 Nachricht abfangen, die Path MTU[7] anpassen und das Paket erneut senden.

Path MTU Discovery bei IPv4 Das *DF*-Flag kann dazu verwendet werden, die kleinste MTU auf einem Pfad zu einem Ziel zu erkennen. Ein Paket, bei dem das Flag gesetzt ist, kommt nur dann am Ziel an, wenn die Payload die MTU jedes Hops unterschreitet. Sobald ein Router das Paket nicht verarbeiten kann, kommt eine ICMPv4-*fragmentation-needed* Nachricht zurück, die auswertet werden kann. Path MTU Discovery ist in `rfc1191`[21] beschrieben.

Die Path MTU (PMTU) ist definiert als die kleinste MTU aller Hops:

$$PMTU = \min_{\forall hop} MTU(hop)$$

[7] PMTU: Die kleinste MTU auf den Pfaden von Quelle zum Ziel. Ist die Payload kleiner aus die PMTU, findet keine Fragmentierung statt.

Das Linux-Tool tracepath setzt das Flag, um die MTU eines Pfads zu erkennen. Dazu sendet es große Pakete und passt die Payload an die aktuelle MTU an, bis alle Hops durchquert werden können.

Path MTU Discovery bei IPv6 Überschreitet die Payload eines IPv6 die MTU des ausgehenden Links, müssen Implementierungen immer ein `ICMPV6_PKT_TOOBIG` an den Absender zurücksenden, wodurch dessen IPv6-Stack die MTU des Pfads anpassen kann. Da IPv6 Fragmentierung ausschließlich beim Absender stattfindet und Router bei zu großen Paketen mit ICMPv6 antworten, geschieht PMTU bereits automatisch (`rfc1981`[22]).

Path MTU Discovery für Tunnel

Path MTU Discovery ist Teil des IPv6 Protokolls, Fragmentierung findet ausschließtlich auf dem Client statt, nie auf Routern. Erhält ein Router ein zu großes Paket, sendet er ein `ICMPV6_PKT_TOOBIG` an den Host zurück woraufhin dieser das Paket fragmentiert.

Fragmentierung der unter einem *sit*-Tunnel liegenden IPv4 Schicht lässt sich steuern, indem das Flag `IP_DF` der Tunnel Parameter gesetzt wird:

```
struct ip_tunnel_parm p;
p.iph.frag_off = htons(IP_DF);
```

"Don't-Fragment" gesetzt Ist der Tunnel Parameter `iph.frag_off == IP_DF` gesetzt, werden ausgehende IPv4 Pakete des Tunnels grundsätzlich nicht fragmentiert. Wird bei einem zu großem IPv6-in-IPv4 Paket die MTU eines IPv4 Hops überschritten, sendet dieser ein `ICMPv4_FRAGMENTATION_NEEDED` an den Host zurück. Diese Nachrichten werden vom IPv4 Stack verarbeitet, um die IPv4 Path MTU anzupassen.

Die aktuelle MTU des IPv6 Tunnels wird beim Senden direkt durch der MTU der IPv4 Verbindung bestimmt.

```
struct iphdr  *tiph = &tunnel->parms.iph;

/* [...] */
/* net/ipv6/sit.c@639 */
        if (tiph->frag_off)
                mtu = dst_mtu(&rt->u.dst) - sizeof(struct iphdr);
        else
                mtu = skb->dst ? dst_mtu(skb->dst) : dev->mtu;
```

So wird sichergestellt, dass $\text{MTU}_{IPv6} \leq \text{MTU}_{IPv4}$, damit IPv4 Pakete nie fragmentiert werden, mit einer Ausnahme: Bei IPv6 beträgt die minimal erlaubte MTU 1280 Byte. Pakete diese Größe müssen ohne IPv6 Fragmentierung ankommen können. Um das bei einem Tunnel garantieren zu können, muss es der darunterliegenden IPv4 Schicht erlaubt werden, bei einer Payload von 1280 Byte das Paket zu fragmentieren, wenn die IPv4 MTU diese Größe unterschreitet. In diesem Fall wird das `IP_DF`-Flag wieder entfernt (`ipip6_tunnel_xmit` [*net/ipv6/sit.c@706*]).

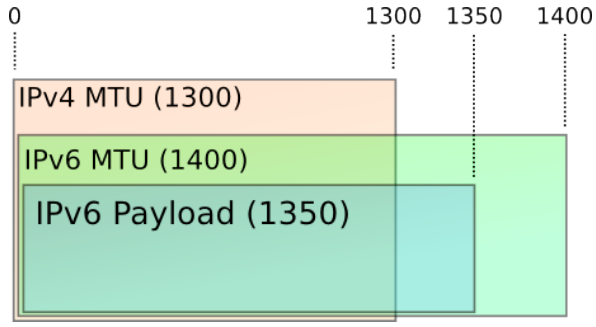
```
/* net/ipv6/sit.c@706 */
        if (mtu > IPV6_MIN_MTU)
                iph->frag_off   =       htons(IP_DF);
        else
                iph->frag_off   =       0;
```

Überschreitet die IPv6 Payload bei gesetzem `IP_DF`-Flag die vom IPv4-Stack abgeleitete MTU, erfhält der Absender ein `ICMPV6_PKT_TOOBIG` zurück und kann fragmentieren.

"Don't-Fragment" nicht gesetzt Ist der Tunnel Parameter `iph.frag_off == 0`, das *DF*-Flag also nicht gesetzt, sollten ausgehende IPv4 Pakete immer fragmentiert werden dürfen. Die MTU des IPv6 Tunnels wird nun nicht mehr durch die IPv4 MTU beschränkt, sondern hängt von der jeweiligen IPv6 PMTU oder der konfigurierten MTU der Tunnelschnittstelle ab, üblicherweise zwischen 1280 und 1480 Bytes, kann aber auch jenseits der üblichen IPv4 MTU Grenze von 1500 Byte liegen.

Allerdings setzt der Linux Kernel fälschlicherweise *immer* das *DF*-Flag, sobald die IPv6 MTU den Wert von 1280 überschreitet. Das verursacht in bestimmten Szenarios Paketverlust (Abbildung 7.2). Das zu sendende IPv6 Paket der Größe 1350 wird nicht

Abbildung 7.2.: Problematisches PMTU Szenario bei SIT-Tunnel



fragmentiert, da es innerhalb der IPv6 MTU von 1400 Byte liegt. Zusammen mit dem IPv4 Header beträgt das zu sendende IPv4 Paket nun 1370 Byte und überschreitet damit die IPv4 MTU von 1300 Byte und müsste fragmentiert werden. Da der Linux Kernel trotz deaktiviertem PMTU-Discovery das *DF*-Flag setzt, werden vom IPv4 Stack des Hosts oder Routers `ICMPv4_FRAGMENTATION_NEEDED` Pakete zurück gesendet, die der IPv4 Stack auch verwendet, um die IPv4 MTU anzupassen. Die IPv6 MTU wird bei deaktiviertem PMTU-Discovery allerdings nicht angepasst und auch nachfolgende Pakete werden die IPv4 MTU überschreiten und verworfen.

Da hier IPv4 Fragmentierung zwingend notwendig und auch erwünscht ist, handelt es sich um einen Bug im Kernel. Eine entsprechende Korrektur wurde eingereicht (Anhang A.8) und bereits für die Aufnahme in Linux-2.6.32 nominiert.

Fragmentierung und TTL Der Kernel enthält an einer Stelle eine direkte Abhängigkeit von `p.iph.frag_off == 0` zur TTL eines Tunnels. So wird beim Erstellen des Tunnels in `ipip6_tunnel_ioctl` [*net/ipv6/sit.c*@811] das `IP_DF`-Flag erzwungen, wenn die TTL $\neq 0$:

```
/* net/ipv6/sit.c@811 */
        if (p.iph.ttl)
                p.iph.frag_off |= htons(IP_DF);
```

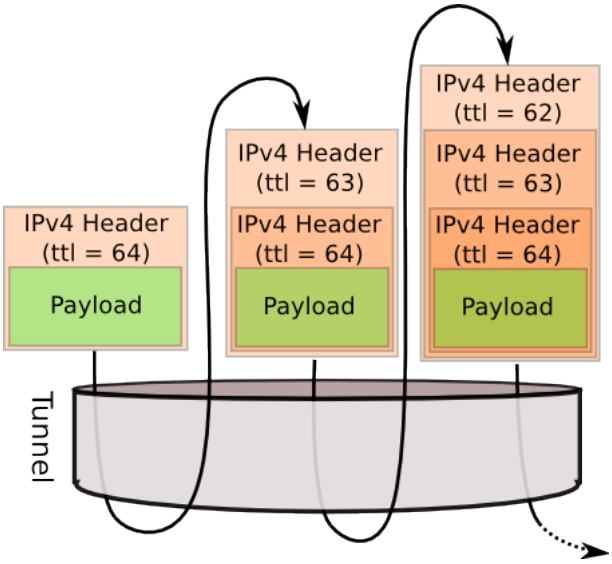
Dieselbe Überprüfung findet sich im Programm iproute2:

```
        if (p.iph.ttl && p.iph.frag_off == 0) {
                fprintf(stderr, "ttl != 0 and noptmudisc are incompatible\n");
                return -1;
        }
```

Der Grund für diese Einschränkung bei einem IPv6-in-IPv4 Tunnel ist nicht direkt ersichtlich. Eine Erklärung dazu ist in *net/ipv4/ip_gre.c*@53 zu finden:

> 1. The most important issue is detecting local dead loops. They would cause complete host lockup in transmit, which would be "resolved" by stack overflow or, if queueing is enabled, with infinite looping in net_bh.
>
> [...]
>
> Current solution: The solution was UNEXPECTEDLY SIMPLE. We force DF flag on tunnels with preconfigured hop limit, that is ALL. :-) Well, it does not remove the problem completely, but exponential growth of network traffic is changed to linear (branches, that exceed pmtu are pruned) and tunnel mtu fastly degrades to value < 68, where looping stops.
>
> [...]

Abbildung 7.3.: IPv4 Routing Loop bei Tunnel



Bei IPv4-In-IPv4 Tunnel besteht die Gefahr, dass bereits eingepackte Pakete bei einem Kreisrouting erneut in IPv4 eingepackt werden, ohne dass sie vorher ausgepackt werden. So wird einem Paket jedesmal, wenn es den Tunnel durchläuft, ein weiter IPv4-Header vorangehängt, sodass das Paket bei jedem Durchlauf um 20 Byte zunimmt. Solche Schleifen lassen sich nur schwer erkennen oder verhindern.

Keine IPv4 Fragmentierung Darf das IPv4 Paket nicht fragmentiert werden, wird das Paket verworfen, sobald die Größe des Pakets die MTU der Schnittstelle überschreitet.

Bei einer MTU von 1500 Byte ergeben sich so maximal 75 Durchläufe und ein gesamtes Datenvolumen von ca. 56 KB.

$$\sum_{n=1}^{75} n \cdot 20 \text{ Byte} = 57.000 \text{ Byte}$$

IPv4 Fragmentierung und vererbte TTL Der Kernel erlaubt IPv4 Fragmentierung bei Tunneln nur dann, wenn die TTL auf 0 (=inherit) gesetzt ist, bei dem die TTL des äußeren IPv4 Headers aus dem inneren Header übernommen wird. Pro Hop verringert sich die TTL des jeweiligen äußeren Headers und das Paket wird verworfen, wenn der Wert 0 erreicht. Bei einer TTL von 64 ergibt sich demnach ein maximales Datenvolumen von nur ca. 41 KB.

$$\sum_{n=1}^{64} n \cdot 20 \text{ Byte} = 41.600 \text{ Byte}$$

IPv4 Fragmentierung und feste TTL Ist IPv4 Fragmentierung dagegen erlaubt und erhält der äußere IP Header stets eine konstante TTL, wird das Paket erst verworfen, wenn es die maximale Größe von 65535 Byte überschreitet. Im *worst-case* ergibt das bei einem anfänglich leerem IPv4 Paket insgesamt 3275 Durchläufe und ein Datenvolumen von insgesamt ca. 102 MB[8]!

$$\sum_{n=1}^{3275} n \cdot 20 \text{ Byte} = 107.289.000 \text{ Byte}$$

Falls der Mechanismus im Kernel rekursiv implementiert ist, kann der Stack im Kernel (üblicherweise 4KB oder 8KB) überlaufen und so das System zum Absturz bringen ($3275 \cdot 4 \text{ Byte} = 12KB > 8KB$)

Die kritische Kombination Feste TTL/nopmtudisc wird vom Kernel verboten, indem bei einem Tunnel mit fester TTL das Flag `IP_DF` erzwungen wird.

Bedeutung für IPv6-in-IPv4 Tunnel Bei einem *sit*-Tunnel ist diese Art des Kreis Routings nicht möglich, da gesendete IPv6-In-IPv4 Pakete nicht wieder vom *sit*-Modul empfangen und erneut verpackt werden können, da sie vom IPv4-Stack empfangen werden. Dieser kann zwar das Paket auspacken und über *sit* erneut verpacken, das resultiert aber nicht in exponentiellen Datenverkehr und wurde bereits in Kapitel 4.2.6 beschrieben.

Aus Sicht des Autors könnte bei sit *die Kombination* Feste TTL *und* nopmtudisc *wieder erlaubt werden, da die oben geschilderte Situation nicht auftreten kann. Ein entsprechender Patch wurde noch nicht vorbereitet.*

[8] This is bad!

Senden des Pakets

Umgang mit Link Fehlern Wurde der Tunnel durch Empfang von ICMPv4 Nachrichten als fehlerhaft markiert und ist die Fehlerzeit von 30 Sekunden nicht abgelaufen, wird beim Senden neuer Pakete erneut die Funktion `dst_link_failure(skb)` aufgerufen, um ICMPv6 Nachrichten zu generieren.

```
658         /* [...] */
            /* net/ipv6/sit.c@660 */
660         if (tunnel->err_count > 0) {
                    if (time_before(jiffies,
662                                 tunnel->err_time + IPTUNNEL_ERR_TIMEO)) {
                            tunnel->err_count--;
664                         dst_link_failure(skb);
                    } else
666                         tunnel->err_count = 0;
            }
```

Ist die Zeit abgelaufen, wird der Tunnel wieder als funktionsfähig markiert. In jedem Fall werden zu sendende Pakete verpackt und als *proto-41* versendet.

IP Header erzeugen Nach der Vorbereitung wird dem zu sendenden Paket ein IPv4 Header vorangehängt und die Parameter entsprechend gesetzt. Ist für den Tunnel keine TTL gesetzt (0 = inherit), wird die TTL des IPv4 Pakets aus der TTL des IPv6 Pakets entnommen. Das resultierende IPv4 Paket wird in den IPv4 Netzwerkstack eingeführt und von diesem an den zuvor ermittelten IPv4 Zielhost versendet.

7.1.8. Zusammenfassung der ISATAP Unterstützung

Adresskonfiguration Die automatische Konfiguration der IPv6-Adressen ist ***gut***. Es werden korrekte linklokale IPv6 Adressen auf Basis von IPv4 Adressen gebildet.

Stateless Autoconf für ISATAP im Kernel ist ***nicht vorhanden***.

Die API ist zwar vorhanden, jedoch so transparent, dass die Besonderheiten der ISATAP-Schicht nur schwer berücksichtigt werden können.

Potential Router List Die Implementierung der PRL ist ***gut***. Eine unbegrenze Zahl von Routern kann konfiguriert weden, gerouteter Datenverkehr wird nur von Routern angenommen.

Der empfohlene Parameter *PrlRefreshInterval* fehlt, ist derzeit aber auch nicht nötig, da *Stateless Autoconf* zur Zeit nicht durchgeführt wird. Die Schnittstelle zum Userspace ist ausreichend.

Sicherheit Die Sicherheitsmaßnahmen im Kernel sind ***gut***. Potentielle Router erhalten eine Sonderrolle. Die eingebetteten IPv4-Adressen werden entsprechend `rfc5214, Abschnitt 7.3` überprüft und bei Unstimmigkeit verworfen.

Router Advertisements werden nur von konfigurierten Routern angenommen. Empfangen und Senden von Daten ist ***gut***.

ICMPv4 Fehler Die Behandlung von ICMPv4 Fehlern für ISATAP Tunnel ist ***nicht vorhanden***. Im Fehlerfall treten Paketverluste auf.

Userspace Die Unterstützung für ISATAP im Userspace ist ***schlecht***. Zum Zeitpunkt der Arbeit ist kein einziges Programm bekannt, das eine automatische Konfiguration der PRL durchführt.

Selbst manuelle Bearbeitung der PRL eines Tunnels mit iproute2 ist zum Zeitpunkt der Arbeit nicht möglich.

7.2. Lösung

Die vollständigen, in dieser Arbeit entwickelten Patches für den Linux Kernel sind in Anhang A aufgeführt.

7.2.1. Stateless Autoconf

Version 1

Datenstrukturen Die hier vorgestellte Implementierung rüstet *Stateless Autoconf* für ISATAP im Linux Kernel nach. Dafür wird zunächst der Struktur `struct ip_tunnel_prl` [*include/linux/if_tunnel.h*@28] ein Element `rs_delay` hinzugefügt. Ein Userspaceprogramm kann somit bestimmen, in welchem Abstand an einen Eintrag der PRL Router Solicitations gesendet werden sollen. Bei einem Wert von 0 wird intern der Standardwert von 900 Sekunden verwendet.

Der internen Datenstruktur eines PRL Eintrages wurden zusätzliche Elemente für Timer und Sperren hinzugefügt.

Timer starten Kernstück der Änderung ist der Timer, der bei Ablauf eine *Router Solicitation* Nachricht an den Router versendet. Jeder PRL Eintrag besitzt dabei einen eigenen Timer, der gestartet wird, sobald er der PRL hinzugefügt wird. Wird ein Eintrag wieder entfernt, wird der Timer gestoppt und gelöscht. Nach jedem Ablauf des Timers wird der Timer erneut auf die Zeit `rs_delay` gesetzt.

Senden der Router Solicitation Läuft der Timer für einen PRL Eintrag ab, wird über alle konfigurierten IPv6 Adressen der Tunnelschnittstelle iteriert und von allen linklokalen IPv6 Adressen der Schnittstelle eine *Router Solicitation* Nachricht gesendet. Die Zieladresse des ICMPv6-Pakets ist dabei die linklokale IPv6 Unicast Adresse des Routers, die über die Funktion `__ipv6_isatap_ifid(...)` [*net/ipv6/addrconf.c*@1521] (Listing 7.7) berechnet wird. Ist eine Tunnelschnittstelle mit mehreren linklokalen IPv6 Adressen konfiguriert, wird so von jeder möglichen Quelladresse eine *RS*-Nachricht an den Router gesendet.

Über die Funktion `ndisc_send_rs(dev, saddr, daddr)` [*net/ipv6/ndisc.c*@627] wird schließlich die ICMPv6/*Router Solicitation*-Nachricht erstellt und versendet. Da dieser Vorgang völlig transparent über die IPv6 Schicht abläuft und erst wesentlich später in IPv4 eingepackt wird, muss die Zieladresse hierfür eine IPv6 ISATAP Adresse sein. Das Versenden von *Router Solicitations* an das Multicast IPv6 Ziel `ff02::2`, eingebettet in ein Unicast IPv4 Paket, ist nicht ohne größere Änderungen möglich. Der hier implementierte Mechanismus schlägt fehl, wenn der Router nicht mit einer korrekten linklokalen ISATAP Adresse konfiguriert ist, wie es bei vielen Cisco Geräten oder auch Windows XP der Fall ist (siehe Kapitel 5.2).

Die zugehörigen *Router Advertisements* werden in dem bereits vorhandenem Code im Linux Kernel verarbeitet; es sind keine weiteren Modifikationen notwendig. Wird aus dem Userspace nun zu einem Tunnel ein potentieller Router konfiguriert, läuft dazu

automatisch der *Stateless Autoconf* Mechanismus ab und der Host erhält bei Erfolg globale IPv6 Adressen und Routen.

Veröffentlichung Der Patch findet sich in Anhang A.5 und wurde in Linux-2.6.31 aufgenommen, jedoch für Linux-2.6.32 zugunsten einer reinen Userspace Implementierung wieder entfernt.

Version 2

Die vorherige Implementierung stellt bereits eine funktionsfähige Implementierung von *Stateless Autoconf* im Kernel zur Verfügung, die allerdings einige Unzulänglichkeiten aufweist und sich nicht vollständig an Empfehlungen verschiedener RFCs hält:

- Es war keine Möglichkeit vorgesehen, *Stateless Autoconf* zu deaktivieren.
- *Router Solicitations* wurden stets unmittelbarversendet und nicht erst nach einer zufälligen Verzögerung, empfohlen in `rfc4861, Abschnitt 6.3.7`.
- Pro Interval wurde genau eine *Router Solicitation*-Nachricht gesendet, statt mehrere hintereinander, um Paketverlust zu begegnen.
- Ist Stateless Autoconf *nicht* erfolgreich, sollte der Mechanismus nach nur kurzer Zeit erneut beginnen.

Der vorhandene Code wurde in diesen Punkten verändert.

Stateless Autoconf deaktivieren Ist `rs_delay == 0` wird *Stateless Autoconf* für ISATAP nun deaktiviert, statt wie zuvor einen Standardwert anzunehmen.

Zufällige Verzögerung Eine zufällige Verzögerung bewirkt eine gleichmäßige Verteilung mehrerer *Router Solicitations* über einen Zeitraum, statt alle anfallenden *Router Solicitations* zur selben Zeit zu senden. Das beugt Paketverlust durch *Congestion* (Verstopfung) des Links vor. Dabei wird beim Anlegen des PRL Eintrags die Verzögerung des ersten Timer Intervals auf einen Zufallswert zwischen 0 und `cfn.rtr_solicit_delay` gesetzt.

Der Wert kann über */proc/sys/net/ipv6/conf/default/router_solicitation_delay* angepasst werden und beträgt üblicherweise 1 Sekunde. Nachfolgende *Solicitations* werden nicht erneut verzögert.

Erneutes Versenden von Router Solicitations Wird eine *Router Solicitation*-Nachricht gesendet, wird der Timer auf den Wert von `cfn.rtr_solicit_interval` gesetzt, einstellbar in */proc/sys/net/ipv6/conf/default/router_solicitation_interval*, üblicherweise 4 Sekunden. Nach Ablauf dieser Zeit wird erneut eine *RS*-Nachricht gesendet, außer es wurde zwischen zwei Ereignissen ein zugehöriges *Router Advertisement* auf der Schnittstelle empfangen. Bei Erfolg, findet des nächste Ereignis des Timers erst nach dem konfiguriertem Interval `rs_delay` statt.

Wurde auf eine gesendete *Router Solicitation* kein *Router Advertisement* empfangen, wird das Paket erneut gesendet und ein Zähler erhöht. Bis zu `cfn.rtr_solicits` Pakete werden hintereinander gesendet, sofern keine *Router Advertisements* empfangen werden, danach wird angenommen, dass auf dem Link kein Router verfügbar ist. Die Anzahl lässt sich über */proc/sys/net/ipv6/conf/default/router_solicitations* konfigurieren, üblich sind 3 Versuche. Damit kann *Stateless Autoconf* auch auf Verbindungen mit gelegentlichem Paketverlust Erfolg haben.

Router nicht verfügbar Wird nach dem letzten Versuch kein *Router Advertisement* empfangen, wird das nächste Timer Ereignis um den Wert `MIN_ROUTER_SOLICIT_INTERVAL` (`= HZ * 120` = 2 Minuten) verzögert, da anzunehmen ist, dass der Router momentan nicht verfügbar ist. Damit wird eine Überflutung des Links durch unnötige *Router Solicitations* verhindert und setzt den Host nicht unter unnötige Last.

Unzulänglichkeiten Die zweite Veröffentlichung ist zwar wesentlich RFC-konformer, hat aber noch immer entscheidende Schwächen:

- *Router Advertisements* werden nicht vom ISATAP-Code verarbeitet, sodass die Router Lebenszeit aus der *Router Solicitation* nicht für das Interval verwendet werden kann, wie in `rfc5214, Abschnitt 8.3.4` empfohlen.

- Der Code in Version 2 berücksichtigt nur, ob *irgendein Router Advertisement* zu *irgendeinem* Router empfangen wurde und beendet dann den Mechanismus für alle noch ausstehenden Router. Wie in Kapitel 4.1.4 beschrieben, muss der Mechanismus aber für alle Router unabhängig voneinander durchgeführt werden, was mit den derzeitigen Mitteln nicht möglich ist.

Veröffentlichung Die so angepasste Implementierung hält sich genauer an bestehende RFCs und besitzt vielfältigere Konfigurationsmöglichkeiten, ist aber nicht ausbaufähig genug.

Der zugehörige Patch ist in Anhang A.6 zu finden und wurde dem Autor von `rfc5214`, Fred L. Templin, zum Review vorgelegt. Aufgrund der Unzulänglichkeiten wurde von einer endgültigen Veröffentlichung abgesehen und zusätzlich der Code aus Version 1 zugunsten einer reinen Implementierung im Userspace aus dem Kernel entfernt.

7.2.2. ICMPv4 Fehler des Tunnels

Auf Basis der früheren `I_WISH_WORLD_WERE_PERFECT`-Implementierung (siehe Seite 82) wurde während dieser Arbeit eine funktionierende Implementierung erstellt, durch die der Linux Kernel bei *proto-41*-Tunneln angemessen auf ICMPv4 Fehler reagieren kann.

Die Implementierung im Linux Kernel wurde dabei um die folgenden Punkte erweitert:

Mapping von ICMPv4 zu ICMPv6

ICMPv4		ICMPv6	
Typ	Code	Typ	Code
ICMP_DEST_UNREACH	ICMP_SR_FAILED	*ignorieren*	
	ICMP_FRAG_NEEDED	*ignorieren*	
	sonstige	ICMP_DEST_UNREACH	ICMP_ADDR_UNREACH
ICMP_TIME_EXCEEDED	ICMP_EXC_TTL	ICMP_TIME_EXCEED	ICMP_EXC_HOPLIMIT
	sonstige	*ignorieren*	
sonstige	*sonstige*	*ignorieren*	

Nur eine Auswahl von ICMPv4 Fehlern wird auf entsprechende ICMPv6 Fehler gemappt. Da das ICMPv6 Paket als Reaktion eines Layer 2 Fehlers generiert wird, ist lediglich der Typ `ICMP_DEST_UNREACH` interessant, aber auch ein `ICMP_TIME_EXCEEDED` wird an den Userspace weitergereicht. ICMPv4 Nachrichten, die nicht unmittelbar einen Layer 2 Fehler bedeuten, werden verworfen.

Prüfung der Payload ICMPv4 Pakets

Enthält das empfangene ICMPv4 Paket in der Payload mindestens den IPv4 Header und den IPv6 Header ($= 20 + 40 = 60$ Byte), kann das Paket als ICMPv6 weiter verarbeitet werden, wozu lediglich der ICMPv4- und IPv4-Header entfernt werden müssen. Die resultierende Payload des *proto-41* Pakets wird als Payload eines neuen ICMPv6 Pakets mit der Funktion `icmpv6_send(...)` [*net/ipv6/icmp.c@308*] als IPv6 an die Tunnelschnittstelle gesendet. Der IPv6-Stack empfängt dieses lokale Paket und verarbeitet es weiter. Das Paket wird bei Bedarf an eine andere Schnittstelle geroutet oder an den Userspace übergeben, der über die Payload die Verbindung identifizieren kann.

Es kann vorkommen, dass bei einem entsprechendem ICMPv4 Paket ein ICMPv6 Paket mit einer Payload von 0 Byte gesendet wird, welches vom IPv6 Stack verworfen wird.

Fallback Reicht die empfangene Payload nicht für eine Identifizierung der IPv6 Verbindung aus, bleibt das derzeitige Verhalten erhalten. Im Falle eines statischen Tunnels wird der Tunnel für eine bestimmte Zeit als *fehlerhaft* markiert; während des Zeitraums werden für zu sendende Pakete ICMPv6 Fehler generiert, als Zeichen dass der Link gestört ist. Besitzt der Tunnel keinen Endpunkt, werden ICMPv4 Pakete verworfen, da einzelne ICMPv4 Pakete kein Anzeichen für einen fehlerhaften Link sind.

Veröffentlichung

Der beschriebene Patch ist in Anhang A.9 aufgeführt, stellt aber lediglich eine Art *Proof-Of-Concept* dar. Obwohl der Code in Tests sehr zufriedenstellend funktioniert, wurde er vom Autor nicht veröffentlicht, da die Vorgehensweise von Fred L. Templin nicht als richtig erachtet wird und zu anfällig für Spoofingangriffe ist (siehe Kapitel 4.2.7).

8. Implementierung Userspace

Für die Implementierung der Userspace Programme wird die in Kapitel 7.1.1 vorgestellte API verwendet.

8.1. iproute2

Mit das Programm ip aus dem Paket iproute2[1] kann von Hand ein ISATAP Tunnel erstellt werden:

```
# ip tunnel add name is0 mode isatap local 192.168.6.5 dev eth0
# ip link set is0 up
```

Es wird automatisch eine korrekte linklokale IPv6 Adresse für die Schnittstelle konfiguriert und es ist bereits Datenverkehr möglich. Zwar kann von Hand der Schnittstelle eine globale IPv6 Adresse und Routen konfiguriert werden, Datenverkehr mit der Außenwelt ist aber nicht möglich, da der Kernel subnetzfremden Datenverkehr von nicht-Routern ablehnt. Vor Beginn der Arbeit war es nicht möglich, mit ip die PRL des Kernels zu konfigurieren. Die API für PRLs, die mit Linux-2.6.25 im Kernel aufgenommen wurde, war noch nicht in *iproute2* implementiert.

Nach eingen Recherchen fand sich ein Patch von Fred L. Templin, der diese Funktionen und weitere implementiert, aber bisher noch nicht offiziell veröffentlicht wurde. Der Patch wurde speziell für ISATAP PRL aufbereitet und ist im Anhang A.11 zu finden.

Nach Anwendung des Patches sind die folgenden zusätzlichen Kommandos verfügbar:

PRL Eintrag hinzufügen

```
# ip tunnel prl dev is0 prl-default 192.168.6.5
# ip tunnel prl dev is0 prl-nodefault 192.168.6.6
```

PRL Eintrag löschen

```
# ip tunnel prl dev is0 prl-delete 192.168.6.5
```

Alle PRL Einträge löschen

```
# ip tunnel prl dev is0 prl-delete any
```

[1] iproute2: http://www.linuxfoundation.org/en/Net:Iproute2

Tunnel auflisten (incl. PRL) Ein Auflisten der Tunnel zeigt zusätzlich die konfigurierten Router an der PRL an:

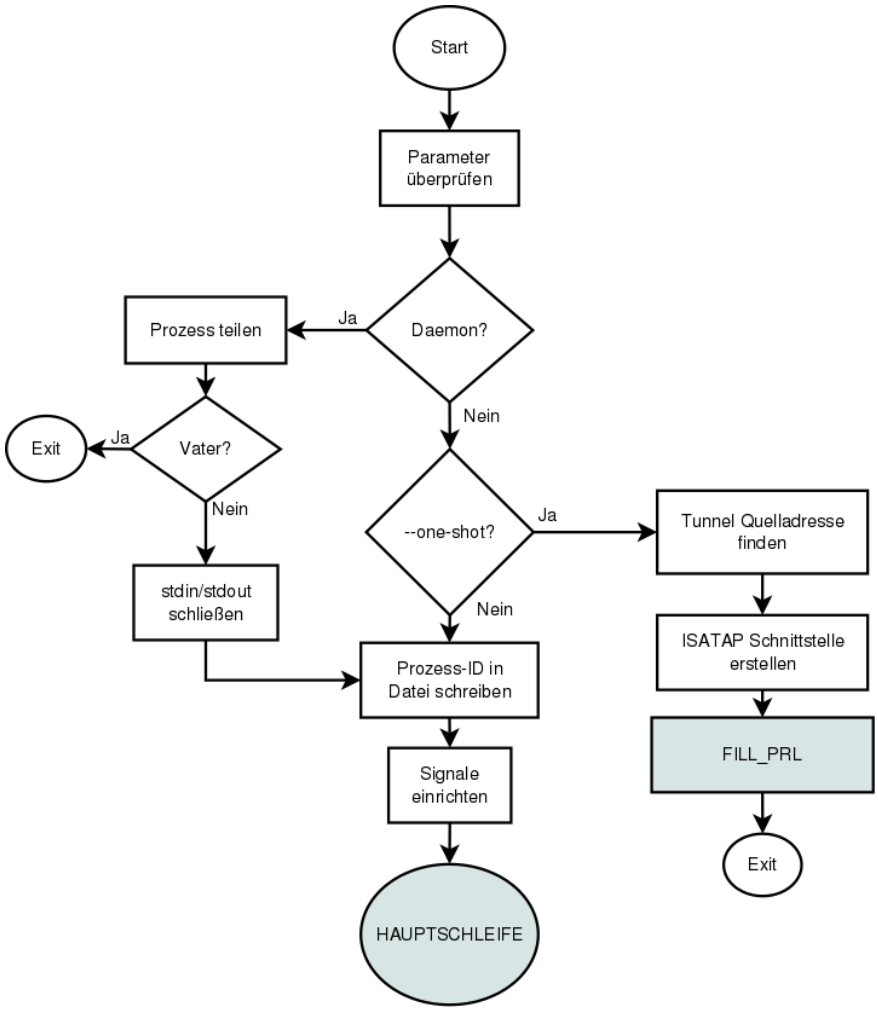
```
# ip tunnel list
sit0: ipv6/ip  remote any  local any  ttl 64  nopmtudisc
is0: ipv6/ip  remote any  local 192.168.6.4  pr 192.168.6.6  pdr
    192.168.6.5  ttl inherit
```

8.2. isatapd-0.9.5

Eine erste Version von isatapd konnte schnell entwickelt und veröffentlicht werden. Im Folgenden ist die veröffentlichte Version isatapd-0.9.5 beschrieben, die unter der Homepage des Projekts [24] herunter geladen werden kann:

`http://www.saschahlusiak.de/linux/isatap.htm`

Abbildung 8.1.: isatapd-0.9.5 - Initialisierung



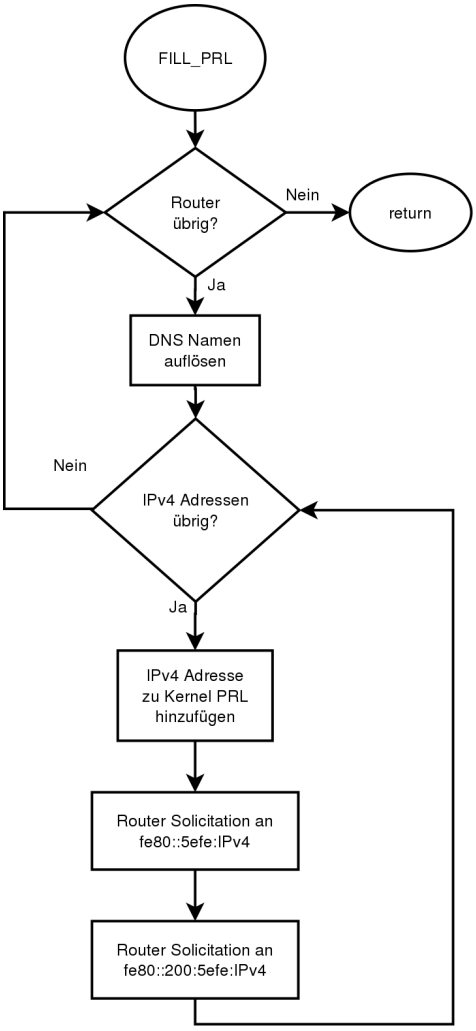
8.2.1. Initialisierung

Abbildung 8.1 zeigt den schematischen Aufbau der Initialisierung des Programms. Das Programm läuft standardmäßig im Vordergrund, kann aber auch als Daemon laufen, wenn bei Programmstart die entsprechenden Optionen gesetzt werden:

`--daemon` Das Programm wird als Daemon im Hintergrund laufen. In diesem Fall kehrt der Aufruf des Programms sofort zurück, sobald der Kindprozess gestartet wurde, der die eigentliche Aufgabe übernimmt. Das Programm schreibt Ausgaben ins Syslog.

`--one-shot` Das Programm kehrt sofort zurück, sobald die Schnittstelle initialisiert und konfiguriert ist. Diese Möglichkeit ist für den Fall vorgesehen, dass der Linux Kernel automatisch *Router Solicitations* an die eingerichteten Default Router versendet, was einen Daemon im Grunde unnötig macht. Das Programm dient dann lediglich der einmaligen Konfiguation.

Abbildung 8.2.: isatapd-0.9.5 - PRL füllen



8.2.2. PRL und Router Solicitations

Die Methode `FILL_PRL` in Abbildung 8.2 kümmert sich um die PRL des Kernels und sendet *Router Solicitations*, für den Fall, dass der Kernel das nicht übernimmt. Für jeden konfigurierten Router werden in dieser Funktion die IPv4 Adressen aufgelöst und jede IPv4 Adresse der PRL des Kernels hinzugefügt.

Da wegen einem Fehler in Cisco Routern die linklokale IPv6 Adresse des Routers nicht zuverlässig bestimmt werden kann, sendet `isatapd` im Anschluss zwei *Router Solicitations*, je eine an die entsprechende *local* und eine an die *global* IPv6 Adresse. Die empfangene Router Advertisement Nachricht wird vom Kernel behandelt, welcher die entsprechenden IPv6 Adressen und Routen konfiguriert.

Senden von Router Solicitations aus dem Userspace

Der folgende Code illustriert das Senden von einer *Router Solicitation* aus dem Userspace an einen IPv6 Host, wie es in isatapd-0.9.5 implementiert ist:

Listing 8.1: Router Solicitations aus dem Userspace senden

```
int send_rdisc(const char *dev, struct in6_addr *addr)
{
        struct sockaddr_in6 target;
        struct nd_router_solicit rs;
        int i;

        int fd = socket (PF_INET6, SOCK_RAW, IPPROTO_ICMPV6);

        if (fd < 0)
                return -1;

        i = 1;
        if (setsockopt (fd, SOL_SOCKET, SO_DONTROUTE, &i, sizeof(int)) < 0) {
                close(fd);
                return -1;
        }

        i = 255;
        if (setsockopt (fd, IPPROTO_IPV6, IPV6_UNICAST_HOPS, &i, sizeof (i)) < 0) {
                close(fd);
                return -1;
        }

        i = 255;
        if (setsockopt (fd, IPPROTO_IPV6, IPV6_MULTICAST_HOPS, &i, sizeof (i)) < 0) {
                close(fd);
                return -1;
        }

        i = 1;
        if (setsockopt (fd, IPPROTO_IPV6, IPV6_RECVHOPLIMIT, &i, sizeof (int)) < 0) {
                close(fd);
                return -1;
        }

        memset(&target, 0, sizeof(target));
        memset(&rs, 0, sizeof(rs));
        target.sin6_addr = *addr;
        target.sin6_family = AF_INET6;

        target.sin6_scope_id = if_nametoindex(dev);
        if (target.sin6_scope_id == 0) {
                close(fd);
                return -1;
        }

        rs.nd_rs_type = ND_ROUTER_SOLICIT;
        rs.nd_rs_code = 0;
        rs.nd_rs_cksum = 0;
        rs.nd_rs_reserved = 0;

        if (sendto(fd, &rs, sizeof(rs), 0,
                        (const struct sockaddr *)&target,
                        sizeof (target)) != sizeof(rs))
        {
                close(fd);
                return -1;
        }

        close(fd);
        return 0;
}
```

Zeile 6 Über die Funktion `socket` wird ein ICMPv6 RAW-Socket erstellt.

Zeile 12-30 Die gesetzten Optionen haben folgende Bedeuting:

`SO_DONTROUTE` Ausgehende Pakete werden nicht über einen Gateway gesendet sondern nur an direkte Nachbarn.

`IPV6_UNICAST_HOPS, IPV6_MULTICAST_HOPS` Setzt die TTL für ausgehende Pakete. `rfc4861, Abschnitt 3.1`[19] verlangt, dass *Router Solicitations* nur mit der maximalen TTL von 255 gesendet und empfangen werden. So werden Empfänger immun gegen Nachrichten, die von außerhalb gesendet werden, da die TTL von bereits gerouteten Paketen stets geringer als 255 sein muss.

`IPV6_RECVHOPLIMIT` Ist das Flag gesetzt, wird die TTL des empfangenen Pakets an den Userspace weitergereicht. Das Programm isatapd benötigt diese Information, um empfangene *Router Advertisements* auf Gültigkeit zu überprüfen.

Zeile 39 Da die Zieladresse linklokal ist, muss der Zieladresse mitgeteilt werden, auf welche Schnittstelle es sich bezieht, hier die Nummer der ISATAP Schnittstelle.

Das Format einer *ICMPv6/Router Solicitation*-Nachricht ist detailliert in `rfc4861, Abschnitt 4.1`[19] beschrieben.

8.2.3. Hauptschleife

Die Hauptschleife in Abbildung 8.3 dient dem periodischem Auffrischen der PRL und des Sendens neuer *Router Solicitations*. Die Schleife und das gesamte Programm wird beendet, sobald durch das Signal `SIGINT` oder `SIGTERM` die Variable `go_down` auf 1 gesetzt wird.

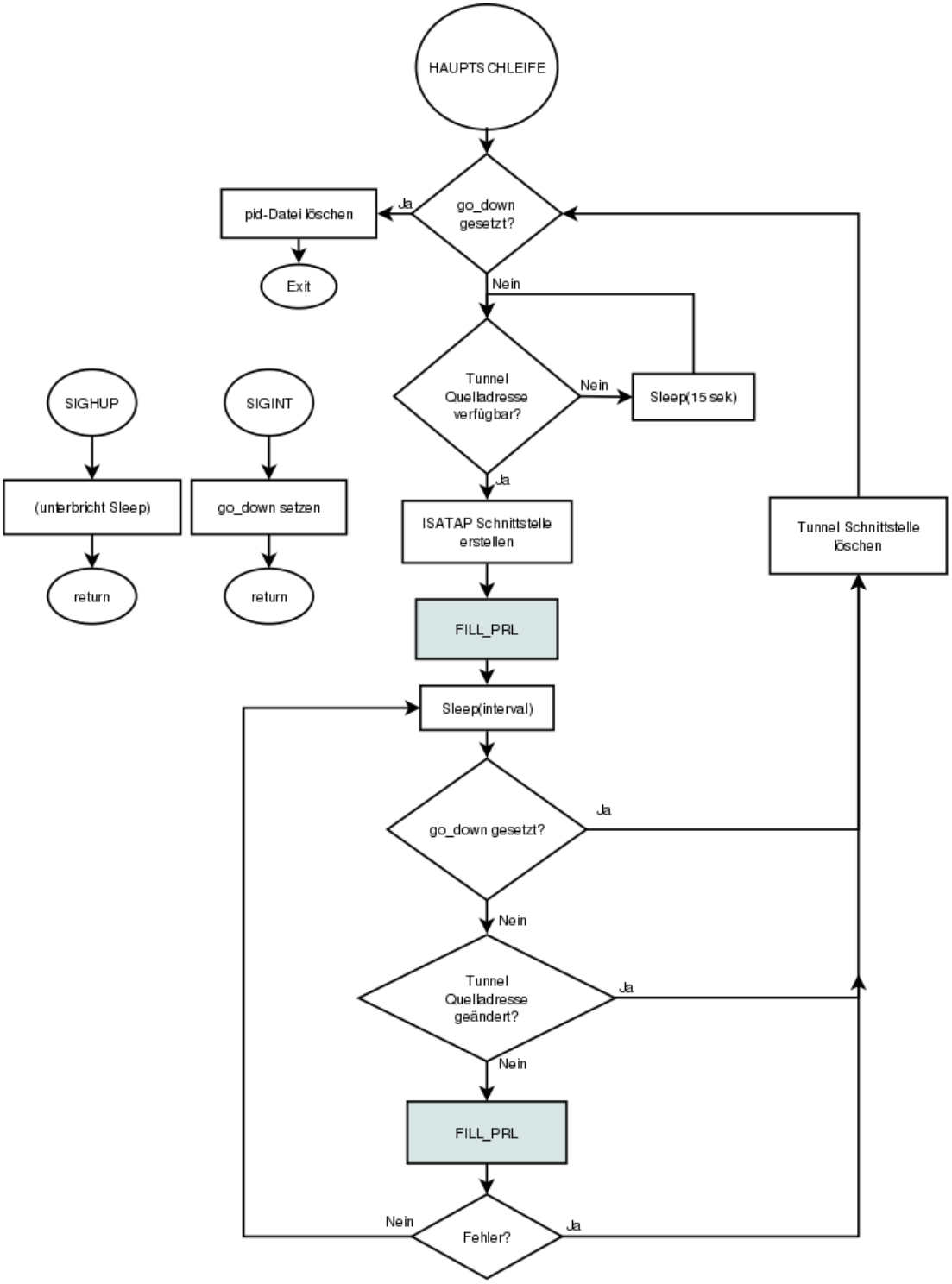
Tunnel Quelladresse ermitteln Da der Tunnel an eine lokale IPv4 Adresse gebunden werden muss, muss die ausgehende IPv4 Adresse bekannt sein, über die der potentielle Router erreicht werden kann. Wird der Tunnel über den Parameter `--link` explizit an eine Netzwerkschnittstelle gebunden, wird dessen konfigurierte IPv4 Adresse verwendet, andernfalls muss über die Routingtabelle die ausgehende Adresse ermittelt werden.

Das geschieht über einen simplen UDP Socket, der mit `connect` an eine Zieladresse gebunden wird. Sobald der UDP Socket verbunden ist[2], kann über die Systemfunktion `getsockname(...)` die lokale Adresse des Sockets erhalten werden, die der Kernel gemäß der Routingtabellen ausgewählt hat.

isatapd-0.9.5 wird diesen Vorgang wiederholen, bis eine lokale IPv4 Adresse gefunden werden kann, die sich für den Tunnel eignet. So wird versucht, Konnektivität aufrecht zu erhalten, slbst wenn die darunterliegende Netztopologie nicht stabil ist.

[2] Bei einem connect auf einem UDP Socket wird *kein* eigentliches Netzwerkpaket gesendet, da UDP im Allgemeinen verbindungslos ist. Der Aufruf dient nur dem lokalen IP Stack.

Abbildung 8.3.: isatapd-0.9.5 - Hauptschleife



Solicitation Schleife Das Programm durchläuft die folgenden Abschnitte:

1. Warten, bis eine Quelladresse für den Tunnel verfügbar ist
2. ISATAP Schnittstelle für lokale IPv4 Adresse erstellen und aktivieren
3. PRL füllen und einmalig *Router Solicitations* senden.
4. Stets nach `--interval` Sekunden wird der Status überprüft:
   - Hat sich die lokale IPv4 Adresse verändert, wird die ISATAP Schnittstelle gelöscht und wieder bei 1 begonnen.
   - Erneut PRL füllen und einmalig *Router Solicitations* senden. Tritt beim Senden ein Fehler auf, wird die ISATAP Schnittstelle zerstört und wieder bei 1 begonnen.

Für die Konfiguration der Schnittstelle und PRL sind für das Programm root-Rechte erforderlich. Da das Programm aber keinerlei Netzwerkverkehr verarbeitet, ist es gegenüber Sicherheitslücken in Netzwerkcode unanfällig.

8.2.4. Bedienung

Alle Optionen sind in `man isatapd` beschrieben und werden mit `isatapd --help` angezeigt. Die wichtigsten Optionen sind:

`--daemon` Programm im Hintergrund laufen lassen.

`--router` Fügt einen weiteren DNS Namen der internen Liste der Router hinzu. Die Namen werden periodisch über DNS in IPv4 Adressen aufgelöst und der PRL des Kernels hinzugefügt. Die Anzahl ist in Version 0.9.5 auf 10 Namen beschränkt, die Anzahl der damit verbundenen IPv4 Adressen ist unbeschränkt. Der unqualifizierte DNS Name `ISATAP` ist voreingestellt. Befindet sich der Host in einer Suchdomain, fügt der Linux Resolver diese automatisch an.

`--interval` Abstand in Sekunden, in dem das Programm *Router Solicitations* versenden soll. Standardwert ist 600 Sekunden, sodass der Router alle 10 Minuten aufgefordert wird, eine *Router Advertisement* zu verschicken.

`--link` Verknüpft den Tunnel mit der angegebenen Schnittstelle. Wird die Option weggelassen, versucht isatapd die ausgehende Schnittstelle für den Router automatisch zu finden.

`--no-user-rs` Verhindert, dass das Programm aus dem Userspace Router Solicitations versendet. Das ist für den Fall gedacht, dass der Kernel durch Patch A.5 selbständig *Router Solicitations* verschickt.

In der Regel reicht der Aufruf von `sudo isatapd --daemon` aus, um eine permanente ISATAP Verbindung zu erhalten.

8.2.5. Unzulänglichkeiten

Zwar ist es mit dem Programm einfach möglich, eine funktionierende ISATAP Verbindung zu erhalten, doch hält es sich an einigen Stellen nicht streng genug an `rfc5214` und zeigt Designschwächen:

- *Router Solicitations* an mehrere Router werden stets zeitgleich gesendet, sodass im Fall geringen Netzdurchsatzes und vielen konfigurierten Routern ein impulsartiger Engpass entstehen kann, durch den Pakete verworfen werden. In `rfc4861, Abschnitt 6.3.7` wird empfohlen, die erste *Router Solicitation* stets zufällig bis zu 1 Sekunde zu verzögern.

- Empfangene *Router Advertisements* werden nicht verarbeitet und geprüft.
  - So kann nicht festgestellt werden, ob ein *Router Solicitation* erfolgreich angekommen ist, um im Falle von Paketverlust das Paket erneut senden zu können. Linux sendet bei *Stateless Autoconf* üblicherweise bis zu 3 Solicitations im Abstand von je 4 Sekunden.
  - Geht ein Paket verloren verzögert sich das nächste mögliche *Router Advertisement* um erneut `--interval` Sekunden und die Gefahr droht, dass die Lebenszeit der Adressen und Routen ablaufen.
  - Ist die *Router Lifetime* geringer als das eingestellte Intervall, verliert der Host bis zum erneuten Senden der Router Solicitations die IPv6 Konnektivität.

    Der Benutzer muss die Lebensdauer der Router kennen, um das Intervall von Hand anpassen zu können. Zudem ist lediglich ein globales Intervall für alle konfigurierten Router möglich. Wünschenswert ist stattdessen eine robuste, automatische Konfiguration des Intervalls abhängig der vom Router gesendeten Informationen.

- PRL Einträge werden nicht wieder aus dem Kernel entfernt, wenn sich die IPv4 Adressen zu einem DNS Namen ändern.

Das Programm wurde unter der Annahme entwickelt, dass ausschließlich der Kernel wüf das Senden und Empfangen von *Router Solicitations/Advertisements* zuständig ist und das Userspace Programm lediglich für die Konfiguration der Schnittstelle.

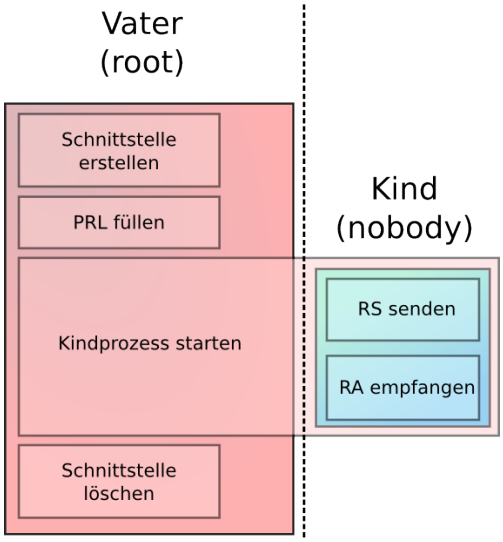
Beim Versuch, die obigen Punkte zu korrigieren zeigt sich, dass *Router Solicitations* und *Router Advertisements* zusammen gehören und beide im Userspace behandelt werden sollten, um den Kernel nicht unnötig zu erweitern. Zwar konfiguriert der Kernel noch immer automatisch zugehörige Adressen und Routen, aber ISATAP spezifischer Programmcode soll sich im Userspace befinden.

Die meisten Unzulänglichkeiten wurden in isatapd-0.9.6 korrigiert.

8.3. isatapd-0.9.6

Für isatapd-0.9.6 wurde insbesondere die Hauptschleife erweitert und Code zum Parsen von empfangenen *Router Advertisements* hinzugefügt. Da das Programm nun durch die Verarbeitung von Netzwerkverkehr anfälliger für Fehler ist, wird dieser Code im Kontext eines nicht-priligierten Benutzers ausgeführt, während der Code zum Verwalten des Tunnels weiterhin root-Rechte benötigt. Verglichen mit isatapd-0.9.5 ergeben sich für den Anwender folgende Änderungen:

Abbildung 8.4.: Trennung von Rechten in isatapd-0.9.6



- Zufällig verzögertes Senden von *Router Solicitations*, um Paketverlust durch Netzverstopfung zu verhindern.
- Wiederholtes Senden von *Router Solicitations* im Falle von Paketverlust.
- Verarbeiten von *Router Advertisements* und automatische Anpassung des Intervalls, in dem *Router Solicitations* gesendet werden sollen.
- Ändern sich während des Betriebs die DNS Einträge, werden veraltete Einträge aus der PRL des Kernels entfernt.

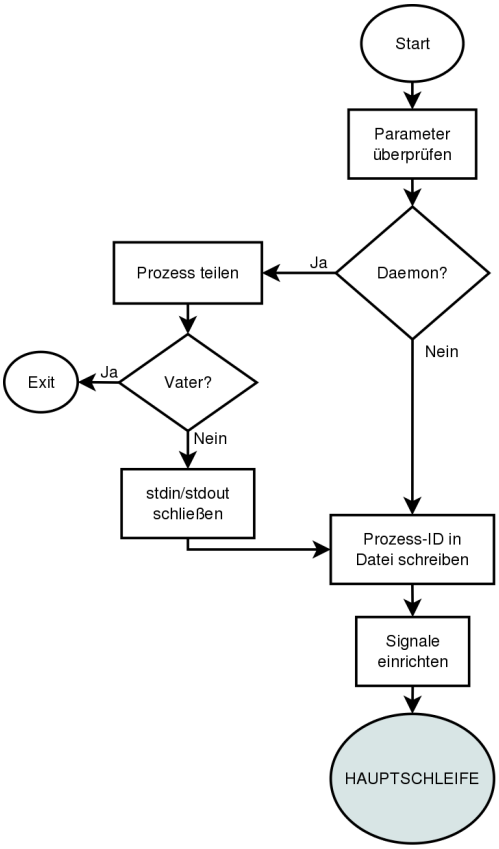
Durch den Code der *Router Advertisements* und genauerer Zeitvorgaben musste die Hauptschleife komplett umgeschrieben werden, um die unterschiedlichen Verzögerungen zu berücksichtigen.

8.3.1. Vater Prozess

Initialisierung Die übergebenen Parameter werden direkt zu Beginn des Hauptprozesses überprüft und alle übergebenen Router in eine verkettete Liste von Router Namen abgespeichert. Erst später werden die Namen über DNS zu IPv4 Adressen aufgelöst, wenn sie der PRL hinzugefügt werden.

Da der Kernel nun nicht mehr die Aufgabe übernehmen kann, *Router Solicitations* zu senden, muss das Userspace Programm permanent laufen. Die Option `--one-shot` wurde daher entfernt, die es zuvor ermöglichte, das Programm direkt nach Konfiguration der Schnittstelle wieder zu beenden.

Abbildung 8.5.: isatapd-0.9.6 - Initialisierung



Verwaltung der internen PRL Die Verarbeitung von *Router Advertisements* macht es nötig, eine interne Liste zu führen, die gesendete *Router Solicitations* und zugehörige Intervalle verfolgt. Beim Aufbau dieser Liste wird für jede IPv4 Adresse eines Router ein Eintrag erstellt[3]. Zur DNS Auflösung sind zwar keine root-Rechte nötig, jedoch aber um im nächsten Schritt die PRL des Kernels zu füllen.

Verwaltung der Kernel PRL Nach Aufbau der internen PRL werden die IPv4 Adressen an die PRL des Kernels übergeben. Wenn die PRL im Kernel alte Einträge enthält, die in der internen PRL nicht mehr enthalten sind, werden die Einträge aus dem Kernel entfernt. Das kann passieren, wenn sich die Adressen zu einem DNS Eintrag ändern, insbesondere wenn unqualifizierte DNS Namen verwendet werden.

[3]Tatsächlich werden pro IPv4 Adresse zwei Einträge erstellt, die es abzufragen gilt: Einen für die IPv6 Adresse mit gesetztem u-Bit und einen Eintrag ohne, damit der Mechanismus auch mit kaputten Cisco Routern erfolgreich sein kann.

Verwaltung der ISATAP Schnittstelle Zum Erstellen und Löschen eines ISATAP Tunnels sind weiterhin root-Rechte nötig.

Solicitation-Loop Prozess starten Ist die Schnittstelle und die PRL konfiguriert, wird der Prozess über `fork()` in einen Vater- und einen Kindprozess geteilt. Der Kindprozess wird im *Solicitation-Loop* den *Stateless Autoconf*-Mechanismus mit den konfigurierten Routern durchführen.

Der Vaterprozess hat die Möglichkeit, bei Bedarf Signale an den Kindprozess zu senden um sein Verhalten zu steuern, es ist derzeit aber nicht vorgesehen, dass der Kindprozess Daten mit dem Vater austauscht. Lediglich wenn der Kindprozess beendet wird, kann er einen Status an den Vaterprozess zurückgeben. Dieser Status kann einen der folgenden definierten Werte annehmen:

`EXIT_CHECK_PRL` Der Vaterprozess soll die PRL neu aufbauen. Das passiert entweder auf Wunsch des Anwenders (`SIGHUP`) oder bei Ablauf einer bestimmten Zeitspanne (Option `--check-dns`, 3600 Sek). Der Kindprozess wird im Anschluß erneut gestartet.

`EXIT_ERROR_LAYER2` Es ist ein Layer-2 Fehler (IPv4) aufgetreten, z.B. als Folge, wenn sich die IPv4 Topologie ändert und *Router Solicitations* nicht mehr gesendet werden können. Der Vaterprozess soll in diesem Fall die PRL neu aufbauen, die Schnittstelle löschen und neu konfigurieren. Der Kindprozess wird bei Erfolg im Anschluss erneut gestartet.

`EXIT_ERROR_FATAL` Ein Fehler ist aufgetreten, der nicht auftreten darf. Der Vaterprozess sollte sich ebenfalls beenden, statt das fehlerhafte Kommando zu wiederholen und möglicherweise in eine Endlosschleife zu geraten.

`number` Signalnummer, wenn der Kindprozess durch ein Signal beendet wurde.

Die Hauptschleife des Vaterprozesses ist vereinfacht in Abbildung 8.6 illustriert.

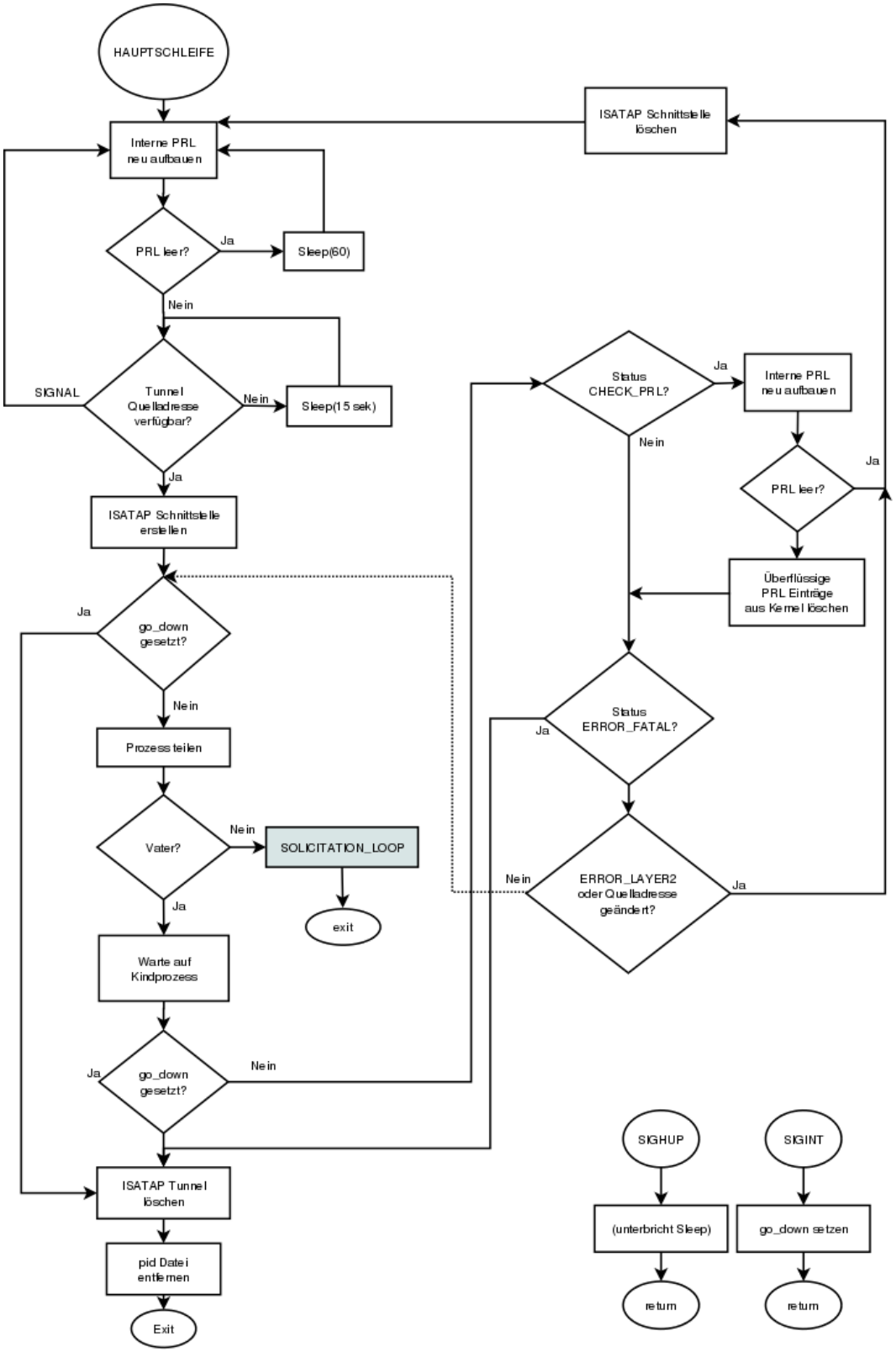
8.3.2. Kind Prozess

Sobald die Schnittstelle und die PRL konfiguriert ist, wird der Kindprozess gestartet. In dessen Initialisierung wird ein `RAW ICMPv6`-Socket erstellt, über den später Netzwerkverkehr gesendet und empfangen werden kann. Der Socket wird dabei mit root-Rechten erstellt, und behält diese Rechte auch nachdem der Kindprozess seine Rechte abgegeben und zu einem nicht-privilegierten Benutzer gewechselt hat.

Der *Solicitation Loop* ist vereinfacht in Abbildung 8.7 dargestellt.

Rechte abgeben Danach werden über den Aufruf von `setuid(65534)` die Rechte abgegeben, die der Prozess zuvor hatte; ab jetzt läuft der Prozess als Benutzer *nobody* (uid 65534); es ist dabei nicht möglich, diese Rechte später wieder zu erhöhen. Der Kindprozess erbt zwar alle Variablen des Elternprozesses, teilt aber nicht dessen Adressraum, sodass über Variablen keine Nachrichten vom Kind- zum Elternprozess ausgetauscht

Abbildung 8.6.: isatapd-0.9.6 - Hauptschleife



werden können. Der Elternprozess muss weiterhin mit root-Rechten existieren, um bei Beendigung des Kinds weitere administrative Befehle ausführen zu können.

Sollte die Bearbeitung von Datenverkehr Fehler enthalten und ein Angreifer das Programm kompromittieren können, kann er als Benutzer *nobody* nur begrenzt Schaden anrichten. Der Elternprozess ist im Hintergrund vor Angriffen geschützt.

Die Aufgaben des Kindprozesses sind:

- Periodisches Senden von Router Solicitations an die konfigurierten Router
- Empfangen und Auswerten von Router Advertisements, um das Intervall anpassen zu können, in dem Router Solicitations gesendet werden. Der Kindprozess ist *nicht* für die Konfiguration von IPv6 Adressen auf Basis der empfangenen *Advertisements* verantwortlich.
- Bei Layer-2 Fehlern soll der Fehler an den Elternprozess weitergereicht werden, damit dieser die Schnittstelle bei Bedarf neu konfigurieren kann.

Hauptschleife

Listing 8.2: Definition struct PRLENTRY - Werte für jeden internen PDR

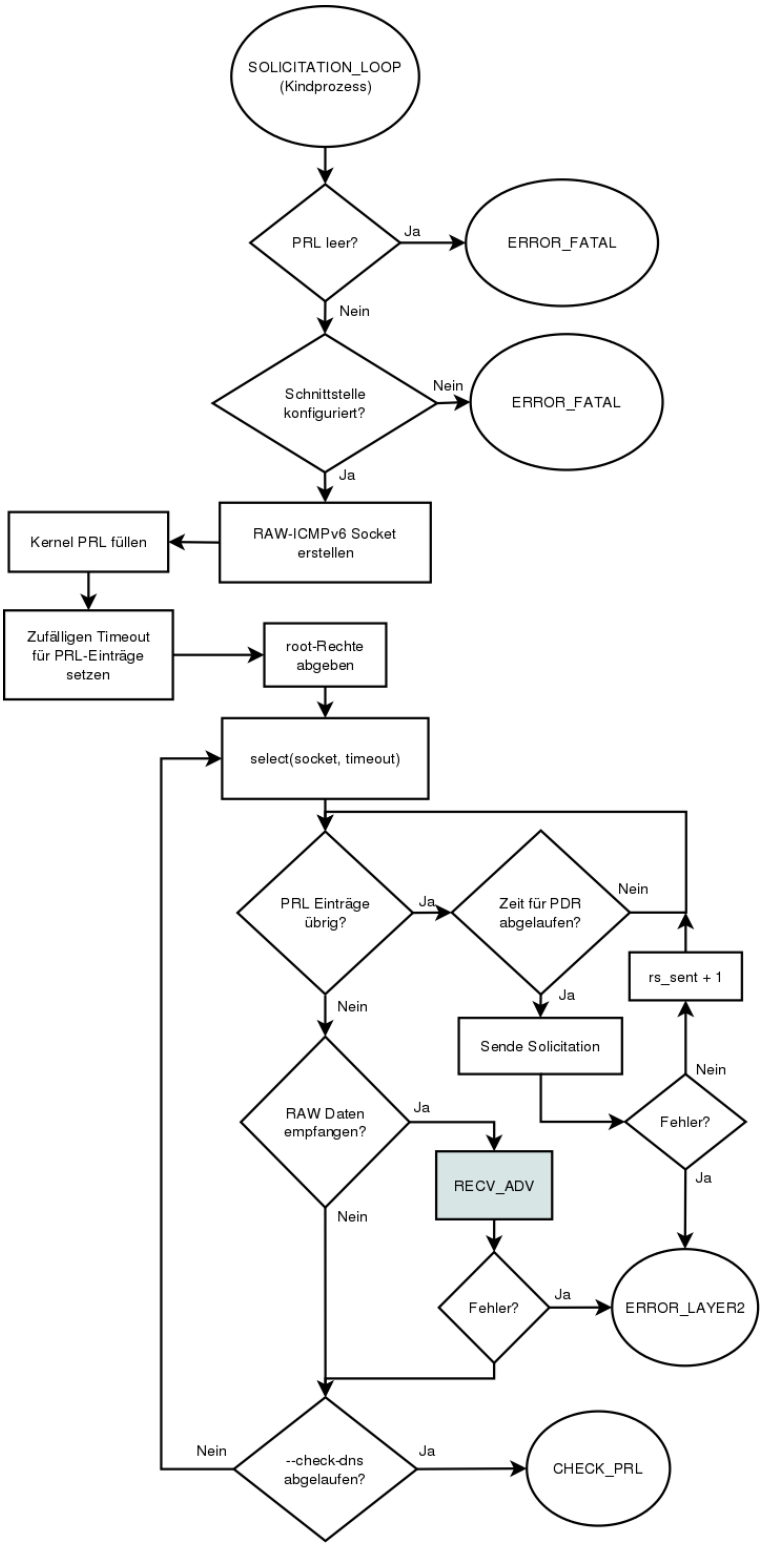
```
struct PRLENTRY {
        struct PRLENTRY* next;          /* next linked list element */

        struct PRLENTRY* sibling;       /* If there is another LL IPv6 address for same
                                           IPv4 address */
        uint32_t ip;                    /* IPv4 address */
        struct sockaddr_in6 addr6;      /* Calculated LL IPv6 address */
        int default_timeout;            /* Default lifetime */
        int next_timeout;               /* Time in ms for next RS */
        int rs_sent;                    /* Number of RS already sent */
        int stale;                      /* Find stale PRL entries, that are not
                                           in DNS anymore */
};
```

Für jeden internen PDR bezeichnet der Wert `struct PRLENTRY::next_timeout` die Anzahl der Millisekunden, nach denen erneut eine *Router Solicitation*-Nachricht gesendet werden soll. Die Hauptschleife schläft genau so lange, wie es die kürzeste Zeitspanne aller Einträge zulässt und benutzt dabei den `select(...)`-Befehl, der gleichzeitig prüfen kann, ob Daten an dem Socket empfangen werden können.

Die Hauptschleife des Kindprozesses wird verlassen, wenn das Intervall `--check-dns` abläuft oder ein Fehler auftritt. Der Hauptprozess erhält danach wieder die Kontrolle und baut die PRL neu auf um im Anschluss wieder den Kindprozess zu starten.

Abbildung 8.7.: isatapd-0.9.6 - Router-Solicitation Schleife

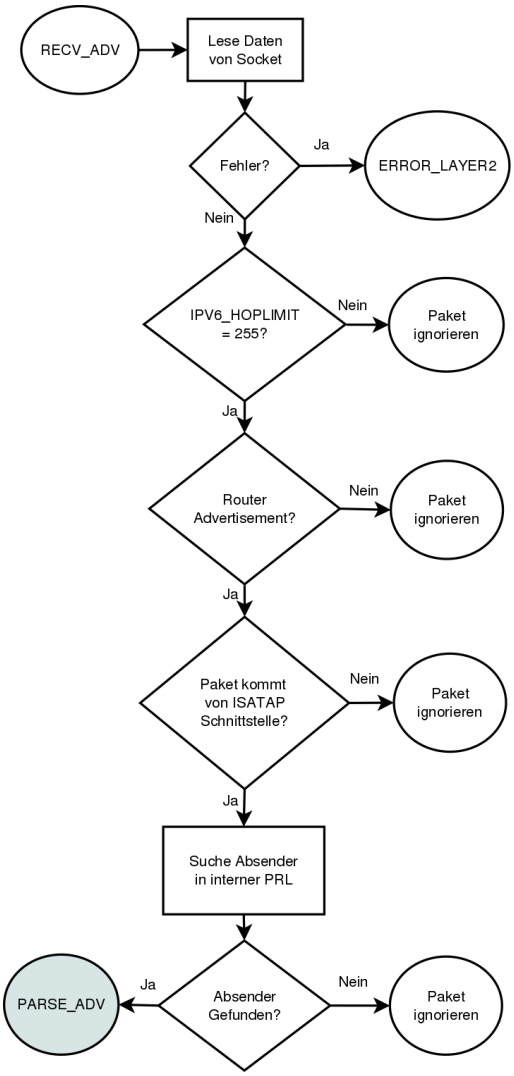


Senden von Router Solicitations Gemäß der Liste in `rfc3330, Abschnitt 3`[8] wird zunächst die IPv4 Adresse eines potentiellen Routers in lokal und global klassifiziert und entsprechend das *u*-Bit der ermittelten IPv6 Adresse gesetzt. In Version 0.9.6 wird zudem bei einer globalen IPv4 Adresse zusätzlich auch die lokale IPv6 Adresse für *Stateless Autoconf* verwendet[4], als Workaround für alte Cisco-IOS Versionen (Kapitel 5.2). Werden zwei IPv6 Adressen zu einer IPv4 Adresse verwendet, werden diese in der internen Liste miteinander verknüpft (**struct** `PRLENTRY::sibling`), sodass bei Erfolg auf einer Adresse die jeweils Andere deaktiviert werden kann.

Aufeinander folgende *Router Solicitations* werden um 4 Sekunden verzögert und der Zähler `rs_sent` erhöht. Nach dem dritten Versuch wird die nächste *RS*-Nachricht um weitere 120 Sekunden verzögert, falls noch immer kein entsprechendes *Router Advertisement* empfangen wurde.

Empfangen von Router Advertisements Über den zuvor erstellten RAW/ICMPv6 Socket können ICMPv6 Nachrichten empfangen werden. Tritt beim Empfang ein Fehler auf, bedeutet das üblicherweise einen Layer-2-Fehler (IPv4), der an den Elternprozess zurückgegeben wird, um die Schnittstelle neu zu initialisieren (Siehe Abbildung 8.8).

Abbildung 8.8.: isatapd-0.9.6 - Empfang von Router Advertisements



Ein empfangenes ICMPv6 Paket muss die folgenden Kriterien erfüllen, damit es weiter verarbeitet wird:

- Die TTL muss 255 betragen. Dadurch wird sichergestellt, dass das Paket direkt auf dem Link gesendet und nicht vorher geroutet wurde.
- ICMPv6 Typ muss ein *Router Advertisement* sein, da über den Socket auch andere

[4] Eine Implementierung, die im Falle einer privaten IPv4 Adresse eine globale IPv6 Adresse konfiguriert, ist nicht bekannt.

Typen, wie ICMPv6 *Echo-Request* oder *Router Solicitations* empfangen werden, die hier nicht von Bedeutung sind.

- Das Paket muss auf der ISATAP-Schnittstelle empfangen worden sein, da über den Sockets *Advertisements* auch von anderen Schnittstellen empfangen werden, die für das Programm uninteressant sind.
- Die IPv6 Adresse des Absenders muss einem Eintrag der internen PRL zugeordnet werden können, damit das *Advertisement* entsprechend verarbeitet werden kann. Da bei ISATAP die Router nicht selbständig *Advertisements* senden können, sind im Normalfall alle *Advertisements* eindeutige Antworten auf gesendete *Router Solicitations*.

In allen anderen Fällen wird das Paket von isatapd ignoriert.

Verarbeiten von Router Advertisements Das Programm sollte *Router Solicitations* in einem so kurzen Abstand senden, dass die Adressen und Routen nicht ihre Gültigkeit verlieren. Hierzu muss das *Advertisement* des Routers verarbeitet und die Lebenszeit der übertragenen Routen für den nächsten Timeout einberechnet werden. Für isatapd sind von einem *Router Advertisement* folgende Werte interessant:

Router Lifetime Gültigkeitsdauer der Default Route auf den Router. Nachdem diese Zeit abgelaufen ist, können keine Pakete mehr über den Router gesendet werden, auch wenn die konfigurierten Adressen noch länger gültig sind. Der Wert ist in Sekunden als vorzeichenloser 16 Bit Wert im *Router Advertisement* enthalten (Siehe `rfc4861, Abschnitt 4.2`). Ein Wert von 0 bedeutet, dass der Router kein Default Router ist; in diesem Falle fließt der Wert nicht in die Berechnung des nächsten Timeouts ein.

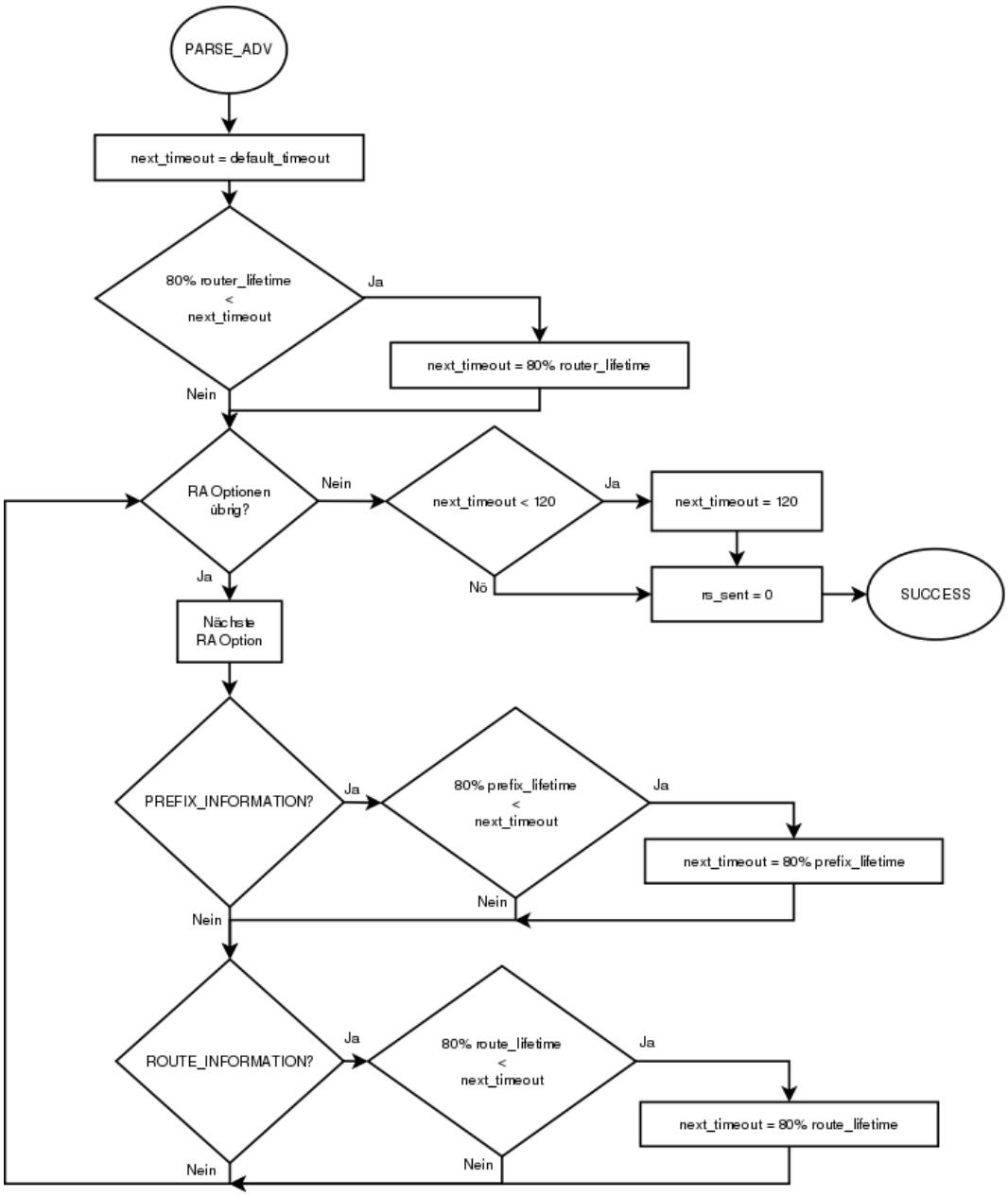
Prefix Preferred Lifetime Einem *Router Advertisement* werden Prefixe als Option mitgesendet (`rfc4861, Abschnitt 4.6.2`). Wichtig für isatapd ist hier die Zeitspanne (in Sekunden), in der die konfigurierte Adresse bei Kommunikation *bevorzugt* wird. Alternativ kann auch die Gültigkeit (Prefix Valid Lifetime) herangezogen werden, der Wert von `0xffffffff` bedeutet *unendlich*.

Specific Route Lifetime In `rfc4191, Abschnitt 2.3`[23] ist eine zusätzliche Option beschrieben, mit der spezielle Routen übermittelt werden können. Diese Option ist noch nicht weit verbreitet und wird nur selten unterstützt, isatapd benutzt aber bereits die dort mitgesendete *Route Lifetime* für die Berechnung des Intervalls.

Die Zeitspannen werden je auf 80% gekürzt, um mit dem nachfolgenden *Router Solicitation*-Mechanismus zu beginnen *bevor* die Zeit abläuft. Der Zeitabstand sollte zudem 120 Sekunden nicht unterschreiten.

Der Benutzer hat die Möglichkeit, mit der Option `--interval` eine zusätzliche, globale obere Schranke des Intervalls zu definieren, was insbesondere bei kurzlebigen, dynamischen Schnittstellen nützlich sein kann (Wireless LAN, VPN, PPP, ...).

Abbildung 8.9.: isatapd-0.9.6 - Verarbeiten von Router Advertisements



8.3.3. Bedienungsanleitung

Beispiel Zum Aufbau einer ISATAP Verbindung reicht ein einzelner Aufruf von isatapd:

Listing 8.3: Beispielaufruf von isatapd-0.9.6

```
# sudo ./isatapd -vv --interval 1700
isatapd[24993]: Adding internal PDR 141.69.1.6
isatapd[24993]: is0 created (local 141.69.55.43, pmtudisc)
isatapd[24993]: interface is0 up
isatapd[24994]: Adding PDR 141.69.1.6 to kernel
isatapd[24994]: Soliciting fe80::5efe:8d45:106
isatapd[24994]: Soliciting fe80::200:5efe:8d45:106
isatapd[24994]: Advertisement from fe80::5efe:8d45:106
isatapd[24994]:   Router lifetime 1800 sec
isatapd[24994]:   Prefix 2001:7c0:1900:11::/64, lifetime 2592000 sec
isatapd[24994]:  Setting next timeout to 1440 sec
isatapd[24994]:  (Not soliciting fe80::200:5efe:8d45:106 anymore)

^Cisatapd[24993]: signal 2 received, going down.
isatapd[24993]: Solicitation Loop exited with status 101
isatapd[24993]: interface is0 down
isatapd[24993]: is0 deleted
```

Der ISATAP Router wird automatisch aus dem unqualifiziertem DNS Namen `isatap` abgeleitet und hier zu `isatap.fh-weingarten.de` (`141.69.1.6`) erweitert.

Die lokale IPv4 Adresse wird automatisch aus der Routingtabelle hergeleitet. Hier führt die lokale IPv4 Adresse `141.69.55.43` zum Ziel.

Zeitabstand zwischen Solicitations wird auf 80% der Router Lifetime gesetzt. Dies ist der kleinstmögliche Wert, bei dem Konnektivität garantiert werden kann.

Die globale IPv6 Adresse wird nicht weiter beworben, sobald die "Schwester"-Adresse Erfolg hat.

Aufruf Parameter Eine vollständige Liste der möglichen Optionen ist in `man isatapd` aufgeführt. Die Wichtigsten sind:

`--name` Name der ISATAP Schnittstelle. Standard: is0.

`--link` Feste Schnittstelle für ISATAP Link benutzen. Standard: automatisch.

`--mtu` Die MTU des Tunnels setzen. Minimum und empfohlener Wert ist 1280. Standard: automatisch.

`--router` Weitere Router Namen für PRL hinzufügen. Standard: `isatap`.

`--interval` Obere Schranke für *Router Solicitation* Intervall. Kann von isatapd noch weiter verringert werden, wenn die *Router Lifetime* es verlangt. Standard: automagisch.

`--check-dns` Intervall, in dem der *Solicitation-Loop* beendet wird um die PRL neu aufzubauen. In der Praxis hat der geringe Standardwert von 3600 Sekunden kaum Bedeutung, da sich DNS Einträge nur selten ändern.

`--daemon` Das Programm im Hintergrund ausführen.

`--pid` Die PID des Daemons in die Datei schreiben, bevor es im Hintergrund läuft. Nützlich für Start-Skripte von Distributionen.

`--verbose` Mehr Debug Ausgaben.

8.3.4. Bekannte Probleme

Auch das Design von isatapd-0.9.6 besitzt noch eine kleine Unzulänglichkeit:

Nur der Elternprozess kann die PRL neu aufbauen, da zum Setzen und Entfernen von PRL Einträgen im Kernel root-Rechte nötig sind. Läuft die `--check-dns` Zeit ab, muss der Kindprozess beendet werden, die PRL wird neu aufgebaut und der Kindprozess neu gestartet.

Da die aktuellen Timeout Werte aber nicht vom Kind- an den Elternprozess gereicht werden können, fängt nach Aufbau der PRL der *Router Solicitation*-Algorithmus vollständig von vorne an und es werden direkt neue *Solicitations* gesendet. Zudem tauchen "Schwester"-IPv6 Adressen wieder in der internen PRL auf, die zuvor im Kindprozess deaktiviert wurden. Der Parameter `--check-dns` gibt somit eine weitere obere Schranke für das Intervall an, in dem *Router Solicitations* gesendet werden.

Diese Unzulänglichkeit ist nicht kritisch und wird möglicherweise in einer nächsten Version behoben.

8.4. Veröffentlichung und Wartung

Quelltext Der Quelltext von isatapd-0.9.6 wurde auf der Homepage des Autors als Archiv veröffentlicht und wird mit dem unter Linux üblichen *Drei-Schritt* übersetzt und installiert.

`http://www.saschahlusiak.de/linux/isatap.htm`

```
$ tar -zxf isatapd-0.9.6.tar.gz
$ cd isatapd-0.9.6

isatapd-0.9.6 $ ./configure
isatapd-0.9.6 $ make
isatapd-0.9.6 $ sudo make install
```

Debian und Ubuntu Besonderer Dank gilt hier Bernhard Schmidt (LRZ München), der von isatapd bereits während der Entwicklung Pakete für Debian und Ubuntu erstellt und getestet hat und sich als Maintainer bei den Distributionen zur Verfügung stellt. Seine Pakete können unter den nachfolgenden URLs heruntergeladen werden; da sie nur wenige Abhängigkeiten haben, ist der Einsatz auch unter älteren Versionen der Distributionen möglich.

Debian SID `http://packages.debian.org/sid/isatapd`

Ubuntu Karmic (offiziell) `http://packages.ubuntu.com/karmic/isatapd`

Ubuntu (inoffiziell) `https://launchpad.net/~berni/+archive/ppa`

Nach der Installation sollte die Datei `/etc/defaults/isatapd` angepasst werden bevor das Programm mit `/etc/init.d/isatapd start` gestartet wird.

Gentoo Vom Autor wurde ein einfaches Gentoo Ebuild vorbereitet, das unter [24] bezogen werden kann. Nach der Installation ist die Konfigurationsdatei `/etc/conf.d/isatapd` anzupassen bevor das Programm über `/etc/init.d/isatapd start` gestartet wird.

Eine offizielle Aufnahme in Gentoo steht zum Zeitpunkt der Arbeit noch aus.

Wartung

Nach Bereitstellung einer robusten Lösung, ist davon auszugehen, dass das Programm nur wenig Wartung bedarf. Da das Programm zudem als Open Source veröffentlicht wurde, ist jedem Anwender die Möglichkeit zur Weiterentwicklung gegeben.

Sobald nativem IPv6 der Durchbruch gelingt, ist die Zukunft von Migrationsstrategien wie ISATAP allerdings fraglich.

9. Zusammenfassung und Ausblick

ISATAP ist eine interessante und relativ einfache Technik, IPv6 Datenverkehr durch ein bestehendes IPv4 Netzwerk zu tunneln. Zwar existieren bereits Implementierungen in gängigen Betriebssystem, doch ist bisher keine davon vollständig RFC-konform und robust genug:

Windows XP Sendet Solicitations an Multicast `ff02::2` und konfiguriert bei globaler IPv4 Adresse eine falsche IPv6 Adresse. *Router Advertisements* werden nicht verarbeitet.

Windows Vista/7 Verarbeitet noch immer keine *Router Advertisements* und nimmt Datenverkehr auch von nicht-Routern an.

Cisco-IOS 12.3(26) Setzt bei globaler IPv4 Adresse eine falsche IPv6 Adresse und routet selbst ungültigen *proto-41* Datenverkehr weiter. *Stateless Autoconf* mit Windows Vista/7 schlägt in der Standardkonfiguration fehl.

MacOSX besitzt derzeit noch keine Untersützung für ISATAP.

Linux Alle früheren Userspace Implementierungen waren unbefriedigend.

Seit 2.6.25 ist im Kernel eine ausreichend robuste Implementierung vorhanden, allerdings gab es hierfür vor dieser Arbeit keine Userspace-Programme zur Konfiguration eines Tunnels.

Diese Arbeit beschäftigte sich intensiv mit der Implementierung von ISATAP im Linux Kernel und liefert mit isatapd ein Userspace Programm, mit dem eine sehr komfortable ISATAP Verbindung als Client ermöglicht wird.

isatapd besitzt die wichtigsten in `rfc5214`[7] empfohlenen Features und stellt damit sogar eine bessere Unterstützung bereit, als aktuelle Windows Versionen. Linux ist nahe dran, zu einer Referenzimplementierung von ISATAP zu werden und es ist anzunehmen, dass das Interesse an ISATAP in Zukunft leicht steigt.

Sobald sich IPv6 in der Praxis durchgesetzt hat, wird ISATAP allerdings wieder stark an Bedeutung verlieren, da aktuelle Hardware IPv6 nativ unterstützt und es in modernen Netzen nur sehr wenige Gründe gibt, IPv6 über IPv4 zu tunneln.

Teil IV.

Anhänge

A. Entwickelte Patches für Linux

Die meisten entwickelten Patches wurden auf der Mailingliste netdev@vger.kernel.org eingereicht, sind online abrufbar und wurden zum Teil bereits in den Linux Kernel integriert:

A.1. sit: Fail to create tunnel, if it already exists

Beschreibung Existiert vor Erstellung eines Tunnels bereits ein ähnlicher Tunnel, soll ein Fehler zurückgegeben werden. Ursprüngliches Verhalten gab immer `Erfolg` zurück, sodass der Userspace nicht angemessen reagieren kann.

Status **integriert in Linux 2.6.31**

`http://patchwork.ozlabs.org/patch/27428/`

`http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=8db99e57`

Code

Listing A.1: Patch: Fail to create tunnel if it already exists

```
diff --git a/net/ipv6/sit.c b/net/ipv6/sit.c
index 664ab82..e625041 100644
--- a/net/ipv6/sit.c
+++ b/net/ipv6/sit.c
@@ -165,8 +165,13 @@ static struct ip_tunnel * ipip6_tunnel_locate(struct net *net,
        struct sit_net *sitn = net_generic(net, sit_net_id);

        for (tp = __ipip6_bucket(sitn, parms); (t = *tp) != NULL; tp = &t->next) {
-               if (local == t->parms.iph.saddr && remote == t->parms.iph.daddr)
-                       return t;
+               if (local == t->parms.iph.saddr &&
+                   remote == t->parms.iph.daddr) {
+                       if (create)
+                               return NULL;
+                       else
+                               return t;
+               }
        }
        if (!create)
                goto failed;
```

A.2. strictly restrict incoming traffic to tunnel link device

Beschreibung Ist ein Tunnel an eine physikalische Schnittstelle gebunden, dürfen empfangene Daten nur von dieser Schnittstelle angenommen werden. Somit werden mehrere gleichartige Tunnel möglich, wenn sie an unterschiedliche Schnittstellen gebunden werden.

Status integriert in Linux 2.6.31

http://patchwork.ozlabs.org/patch/27429/

http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=4fddbf5d

Code

Listing A.2: Patch: strictly restrict incoming traffic to tunnel link device

```
diff --git a/net/ipv6/sit.c b/net/ipv6/sit.c
index e625041..dfedd56 100644
--- a/net/ipv6/sit.c
+++ b/net/ipv6/sit.c
@@ -80,7 +80,7 @@ struct sit_net {
 static DEFINE_RWLOCK(ipip6_lock);

 static struct ip_tunnel * ipip6_tunnel_lookup(struct net *net,
-                __be32 remote, __be32 local)
+                struct net_device *dev, __be32 remote, __be32 local)
 {
        unsigned h0 = HASH(remote);
        unsigned h1 = HASH(local);
@@ -89,18 +89,25 @@ static struct ip_tunnel * ipip6_tunnel_lookup(struct net *net,

        for (t = sitn->tunnels_r_l[h0^h1]; t; t = t->next) {
                if (local == t->parms.iph.saddr &&
-                   remote == t->parms.iph.daddr && (t->dev->flags&IFF_UP))
+                   remote == t->parms.iph.daddr &&
+                   (!dev || !t->parms.link || dev->iflink == t->parms.link) &&
+                   (t->dev->flags & IFF_UP))
                        return t;
        }
        for (t = sitn->tunnels_r[h0]; t; t = t->next) {
-               if (remote == t->parms.iph.daddr && (t->dev->flags&IFF_UP))
+               if (remote == t->parms.iph.daddr &&
+                   (!dev || !t->parms.link || dev->iflink == t->parms.link) &&
+                   (t->dev->flags & IFF_UP))
                        return t;
        }
        for (t = sitn->tunnels_l[h1]; t; t = t->next) {
-               if (local == t->parms.iph.saddr && (t->dev->flags&IFF_UP))
+               if (local == t->parms.iph.saddr &&
+                   (!dev || !t->parms.link || dev->iflink == t->parms.link) &&
+                   (t->dev->flags & IFF_UP))
                        return t;
        }
-       if ((t = sitn->tunnels_wc[0]) != NULL && (t->dev->flags&IFF_UP))
+       t = sitn->tunnels_wc[0];
+       if ((t != NULL) && (t->dev->flags & IFF_UP))
                return t;
        return NULL;
 }
@@ -166,7 +173,8 @@ static struct ip_tunnel * ipip6_tunnel_locate(struct net *net,

        for (tp = __ipip6_bucket(sitn, parms); (t = *tp) != NULL; tp = &t->next) {
                if (local == t->parms.iph.saddr &&
-                   remote == t->parms.iph.daddr) {
+                   remote == t->parms.iph.daddr &&
+                   parms->link == t->parms.link) {
                        if (create)
                                return NULL;
                        else
@@ -451,7 +459,10 @@ static int ipip6_err(struct sk_buff *skb, u32 info)
        err = -ENOENT;

        read_lock(&ipip6_lock);
-       t = ipip6_tunnel_lookup(dev_net(skb->dev), iph->daddr, iph->saddr);
+       t = ipip6_tunnel_lookup(dev_net(skb->dev),
+                               skb->dev,
+                               iph->daddr,
+                               iph->saddr);
        if (t == NULL || t->parms.iph.daddr == 0)
                goto out;

@@ -486,8 +497,9 @@ static int ipip6_rcv(struct sk_buff *skb)
        iph = ip_hdr(skb);

        read_lock(&ipip6_lock);
```

```
-       if ((tunnel = ipip6_tunnel_lookup(dev_net(skb->dev),
-                                         iph->saddr, iph->daddr)) != NULL) {
+       tunnel = ipip6_tunnel_lookup(dev_net(skb->dev), skb->dev,
+                                    iph->saddr, iph->daddr);
+       if (tunnel != NULL) {
                secpath_reset(skb);
                skb->mac_header = skb->network_header;
                skb_reset_network_header(skb);
```

A.3. ipip6_tunnel_del_prl: return err

Beschreibung Funktion `ipip6_tunnel_del_prl` gab statt Fehlercodes immer `Erfolg` zurück.

Status integriert in Linux 2.6.31

`http://patchwork.ozlabs.org/patch/27430/`

`http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=4b279601`

Code

Listing A.3: Patch: ipip6_tunnel_del_prl: return err

```
diff --git a/net/ipv6/sit.c b/net/ipv6/sit.c
index dfedd56..3fd0600 100644
--- a/net/ipv6/sit.c
+++ b/net/ipv6/sit.c
@@ -368,7 +368,7 @@ ipip6_tunnel_del_prl(struct ip_tunnel *t, struct ip_tunnel_prl *a)
        }
 out:
        write_unlock(&ipip6_lock);
-       return 0;
+       return err;
 }

 static int
```

A.4. addrconf: refuse isatap eui64 for INADDR_ANY

Beschreibung `INADDR_ANY` (`0.0.0.0`) als IPv4 Endpunkt eines Tunnels ergibt die nicht gültige IPv6 ISATAP Adresse `fe80::5efe:0:0`, die mit diesem Patch abgelehnt wird.

Status integriert in Linux 2.6.31

`http://patchwork.ozlabs.org/patch/27431/`

`http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=9af28511`

Code

Listing A.4: Patch: refuse isatap eui64 for INADDR_ANY

```
diff --git a/net/ipv6/addrconf.c b/net/ipv6/addrconf.c
index a8218bc..550fb79 100644
--- a/net/ipv6/addrconf.c
+++ b/net/ipv6/addrconf.c
@@ -1520,6 +1520,8 @@ static int addrconf_ifid_infiniband(u8 *eui, struct net_device *dev)
```

```
 int __ipv6_isatap_ifid(u8 *eui, __be32 addr)
 {
+       if (addr == 0)
+               return -1;
        eui[0] = (ipv4_is_zeronet(addr) || ipv4_is_private_10(addr) ||
                  ipv4_is_loopback(addr) || ipv4_is_linklocal_169(addr) ||
                  ipv4_is_private_172(addr) || ipv4_is_test_192(addr) ||
```

A.5. stateless autoconf for isatap

Beschreibung Erste Version von Stateless Autoconf für ISATAP im Linux Kernel.

Status **integriert in Linux 2.6.31**

http://patchwork.ozlabs.org/patch/27432/

http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=64506929

Entfernt für Linux 2.6.32 (siehe Kapitel 7.2.1)

Code

Listing A.5: Patch: stateless autoconf for isatap

```
diff --git a/include/linux/if_tunnel.h b/include/linux/if_tunnel.h
index 5a9aae4..5eb9b0f 100644
--- a/include/linux/if_tunnel.h
+++ b/include/linux/if_tunnel.h
@@ -44,7 +44,7 @@ struct ip_tunnel_prl {
        __u16                   flags;
        __u16                   __reserved;
        __u32                   datalen;
-       __u32                   __reserved2;
+       __u32                   rs_delay;
        /* data follows */
 };

diff --git a/include/net/ipip.h b/include/net/ipip.h
index fdf9bd7..5d3036f 100644
--- a/include/net/ipip.h
+++ b/include/net/ipip.h
@@ -28,11 +28,18 @@ struct ip_tunnel
        unsigned int                    prl_count;      /* # of entries in PRL */
 };

+/* ISATAP: default interval between RS in secondy */
+#define IPTUNNEL_RS_DEFAULT_DELAY      (900)
+
 struct ip_tunnel_prl_entry
 {
        struct ip_tunnel_prl_entry      *next;
        __be32                          addr;
        u16                             flags;
+       unsigned long                   rs_delay;
+       struct timer_list               rs_timer;
+       struct ip_tunnel                *tunnel;
+       spinlock_t                      lock;
 };

 #define IPTUNNEL_XMIT() do {                                            \
diff --git a/net/ipv6/ndisc.c b/net/ipv6/ndisc.c
index 9f061d1..f787ac4 100644
--- a/net/ipv6/ndisc.c
+++ b/net/ipv6/ndisc.c
@@ -658,6 +658,7 @@ void ndisc_send_rs(struct net_device *dev, const struct in6_addr *saddr,
                    &icmp6h, NULL,
                    send_sllao ? ND_OPT_SOURCE_LL_ADDR : 0);
 }
+EXPORT_SYMBOL(ndisc_send_rs);
```

```
 static void ndisc_error_report(struct neighbour *neigh, struct sk_buff *skb)
diff --git a/net/ipv6/sit.c b/net/ipv6/sit.c
index 3fd0600..b3a59bd 100644
--- a/net/ipv6/sit.c
+++ b/net/ipv6/sit.c
@@ -15,6 +15,7 @@
  * Roger Venning <r.venning@telstra.com>:      6to4 support
  * Nate Thompson <nate@thebog.net>:            6to4 support
  * Fred Templin <fred.l.templin@boeing.com>:   isatap support
+ * Sascha Hlusiak <mail@saschahlusiak.de>:     stateless autoconf for isatap
  */

 #include <linux/module.h>
@@ -222,6 +223,44 @@  failed:
        return NULL;
 }

+static void ipip6_tunnel_rs_timer(unsigned long data)
+{
+       struct ip_tunnel_prl_entry *p = (struct ip_tunnel_prl_entry *) data;
+       struct inet6_dev *ifp;
+       struct inet6_ifaddr *addr;
+
+       spin_lock(&p->lock);
+       ifp = __in6_dev_get(p->tunnel->dev);
+
+       read_lock_bh(&ifp->lock);
+       for (addr = ifp->addr_list; addr; addr = addr->if_next) {
+               struct in6_addr rtr;
+
+               if (!(ipv6_addr_type(&addr->addr) & IPV6_ADDR_LINKLOCAL))
+                       continue;
+
+               /* Send RS to guessed linklocal address of router
+                *
+                * Better: send to ff02::2 encapsuled in unicast directly
+                * to router-v4 instead of guessing the v6 address.
+                *
+                * Cisco/Windows seem to not set the u/l bit correctly,
+                * so we won't guess right.
+                */
+               ipv6_addr_set(&rtr,  htonl(0xFE800000), 0, 0, 0);
+               if (!__ipv6_isatap_ifid(rtr.s6_addr + 8,
+                                       p->addr)) {
+                       ndisc_send_rs(p->tunnel->dev, &addr->addr, &rtr);
+               }
+       }
+       read_unlock_bh(&ifp->lock);
+
+       mod_timer(&p->rs_timer, jiffies + HZ * p->rs_delay);
+       spin_unlock(&p->lock);
+
+       return;
+}
+
 static struct ip_tunnel_prl_entry *
 __ipip6_tunnel_locate_prl(struct ip_tunnel *t, __be32 addr)
 {
@@ -280,6 +319,7 @@  static int ipip6_tunnel_get_prl(struct ip_tunnel *t,
                        continue;
                kp[c].addr = prl->addr;
                kp[c].flags = prl->flags;
+               kp[c].rs_delay = prl->rs_delay;
                c++;
                if (kprl.addr != htonl(INADDR_ANY))
                        break;
@@ -329,11 +369,23 @@  ipip6_tunnel_add_prl(struct ip_tunnel *t, struct ip_tunnel_prl *a, int chg)
        }

        p->next = t->prl;
+       p->tunnel = t;
        t->prl = p;
        t->prl_count++;
+
+       spin_lock_init(&p->lock);
+       setup_timer(&p->rs_timer, ipip6_tunnel_rs_timer, (unsigned long) p);
 update:
        p->addr = a->addr;
        p->flags = a->flags;
+       p->rs_delay = a->rs_delay;
+       if (p->rs_delay == 0)
+               p->rs_delay = IPTUNNEL_RS_DEFAULT_DELAY;
+       spin_lock(&p->lock);
```

```
+       del_timer(&p->rs_timer);
+       if (p->flags & PRL_DEFAULT)
+               mod_timer(&p->rs_timer, jiffies + 1);
+       spin_unlock(&p->lock);
 out:
        write_unlock(&ipip6_lock);
        return err;
@@ -352,6 +404,9 @@  ipip6_tunnel_del_prl(struct ip_tunnel *t, struct ip_tunnel_prl *a)
                        if ((*p)->addr == a->addr) {
                                x = *p;
                                *p = x->next;
+                               spin_lock(&x->lock);
+                               del_timer(&x->rs_timer);
+                               spin_unlock(&x->lock);
                                kfree(x);
                                t->prl_count--;
                                goto out;
@@ -362,6 +417,9 @@  ipip6_tunnel_del_prl(struct ip_tunnel *t, struct ip_tunnel_prl *a)
                while (t->prl) {
                        x = t->prl;
                        t->prl = t->prl->next;
+                       spin_lock(&x->lock);
+                       del_timer(&x->rs_timer);
+                       spin_unlock(&x->lock);
                        kfree(x);
                        t->prl_count--;
                }
```

A.6. Update stateless autoconf for isatap

Beschreibung Verbesserung der ersten Implementierung von *Stateless Autoconf* für ISATAP. Setzt den vorherigen Patch voraus und rüstet eine nahezu vollständig RFC-konforme Implementierung von *Stateless Autoconf* für ISATAP nach.

Status ***nicht* eingereicht** (Siehe Kapitel 7.2.1).

Code

Listing A.6: Patch: Update stateless autoconf for isatap

```
diff --git a/include/net/ipip.h b/include/net/ipip.h
index 5d3036f..09aa436 100644
--- a/include/net/ipip.h
+++ b/include/net/ipip.h
@@ -28,8 +28,7 @@ struct ip_tunnel
        unsigned int                    prl_count;      /* # of entries in PRL */
 };

-/* ISATAP: default interval between RS in secondy */
-#define IPTUNNEL_RS_DEFAULT_DELAY      (900)
+#define MIN_ROUTER_SOLICIT_INTERVAL (HZ * 120)

 struct ip_tunnel_prl_entry
 {
diff --git a/net/ipv6/addrconf.c b/net/ipv6/addrconf.c
index 43b3c9f..448ea80 100644
--- a/net/ipv6/addrconf.c
+++ b/net/ipv6/addrconf.c
@@ -394,7 +394,7 @@ static struct inet6_dev * ipv6_add_dev(struct net_device *dev)
                printk(KERN_INFO
                       "%s: Disabled Multicast RS\n",
                       dev->name);
-               ndev->cnf.rtr_solicits = 0;
+               /* ndev->cnf.rtr_solicits = 0; */
        }
 #endif

@@ -2867,6 +2867,7 @@ static void addrconf_dad_completed(struct inet6_ifaddr *ifp)
        if (ifp->idev->cnf.forwarding == 0 &&
            ifp->idev->cnf.rtr_solicits > 0 &&
            (dev->flags&IFF_LOOPBACK) == 0 &&
+           (dev->priv_flags & IFF_ISATAP) == 0 &&
```

```
33              (ipv6_addr_type(&ifp->addr) & IPV6_ADDR_LINKLOCAL)) {
                    /*
35                   *      If a host as already performed a random delay
diff --git a/net/ipv6/sit.c b/net/ipv6/sit.c
37  index 98b7327..b296d0c 100644
--- a/net/ipv6/sit.c
39  +++ b/net/ipv6/sit.c
@@ -228,34 +228,48 @@ static void ipip6_tunnel_rs_timer(unsigned long data)
41          struct ip_tunnel_prl_entry *p = (struct ip_tunnel_prl_entry *) data;
            struct inet6_dev *ifp;
43          struct inet6_ifaddr *addr;
+
45  +       unsigned long when;

47          spin_lock(&p->lock);
            ifp = __in6_dev_get(p->tunnel->dev);
49
            read_lock_bh(&ifp->lock);
51  +       if (ifp->if_flags & IF_RA_RCVD) {
+               when = HZ * p->rs_delay;
53  +               ifp->if_flags &= ~(IF_RA_RCVD | IF_RS_SENT);
+               for (addr = ifp->addr_list; addr; addr = addr->if_next) {
55  +                       if (!(ipv6_addr_type(&addr->addr) & IPV6_ADDR_LINKLOCAL))
+                               continue;
57  +                       addr->probes = 0;
+               }
59  +               goto out;
+       }
61  +       when = MIN_ROUTER_SOLICIT_INTERVAL;
            for (addr = ifp->addr_list; addr; addr = addr->if_next) {
63                  struct in6_addr rtr;

65                  if (!(ipv6_addr_type(&addr->addr) & IPV6_ADDR_LINKLOCAL))
                            continue;
67
-               /* Send RS to guessed linklocal address of router
69  -                *
-                * Better: send to ff02::2 encapsuled in unicast directly
71  -                * to router-v4 instead of guessing the v6 address.
-                *
73  -                * Cisco/Windows seem to not set the u/l bit correctly,
-                * so we won't guess right.
75  -                */
+               if (addr->probes++ < ifp->cnf.rtr_solicits) {
77  +                       when = ifp->cnf.rtr_solicit_interval;
+               } else {
79  +                       addr->probes = 0;
+                           continue;
81  +               }
+
83                  ipv6_addr_set(&rtr,  htonl(0xFE800000), 0, 0, 0);
                    if (!__ipv6_isatap_ifid(rtr.s6_addr + 8,
85                                          p->addr)) {
                            ndisc_send_rs(p->tunnel->dev, &addr->addr, &rtr);
87  +                       addr->idev->if_flags |= IF_RS_SENT;
                    }
89          }
+out:
91          read_unlock_bh(&ifp->lock);

93  -       mod_timer(&p->rs_timer, jiffies + HZ * p->rs_delay);
+       mod_timer(&p->rs_timer, jiffies + when);
95          spin_unlock(&p->lock);

97          return;
@@ -375,16 +389,30 @@ ipip6_tunnel_add_prl(struct ip_tunnel *t, struct ip_tunnel_prl *a, int chg)
99
            spin_lock_init(&p->lock);
101         setup_timer(&p->rs_timer, ipip6_tunnel_rs_timer, (unsigned long) p);
+       p->rs_timer.expires = 0;
103  update:
+       spin_lock(&p->lock);
105         p->addr = a->addr;
            p->flags = a->flags;
107         p->rs_delay = a->rs_delay;
-       if (p->rs_delay == 0)
109 -               p->rs_delay = IPTUNNEL_RS_DEFAULT_DELAY;
-       spin_lock(&p->lock);
111 -       del_timer(&p->rs_timer);
-       if (p->flags & PRL_DEFAULT)
113 -               mod_timer(&p->rs_timer, jiffies + 1);
+
115 +       if ((p->flags & PRL_DEFAULT) && (p->rs_delay)) {
+               unsigned long when;
```

```
+                struct inet6_dev *ifp= __in6_dev_get(p->tunnel->dev);
+
+                read_lock_bh(&ifp->lock);
+
+                when = net_random() % (ifp->cnf.rtr_solicit_delay ? : 1);
+
+                if (jiffies + HZ * p->rs_delay < p->rs_timer.expires)
+                        when = HZ * p->rs_delay;
+
+                mod_timer(&p->rs_timer, jiffies + when);
+
+                read_unlock_bh(&ifp->lock);
+        } else
+                del_timer(&p->rs_timer);
+
        spin_unlock(&p->lock);
 out:
        write_unlock(&ipip6_lock);
```

A.7. 6to4: honour routing table

Status Vom Upstream **ignoriert**

http://patchwork.ozlabs.org/patch/31868/
http://patchwork.ozlabs.org/patch/31898/

Code

Listing A.7: Patch: 6to4: honour routing table

```
diff --git a/net/ipv6/sit.c b/net/ipv6/sit.c
index 98b7327..8ae3028 100644
--- a/net/ipv6/sit.c
+++ b/net/ipv6/sit.c
@@ -656,8 +656,17 @@  static int ipip6_tunnel_xmit(struct sk_buff *skb, struct net_device *dev)
                        goto tx_error;
        }

-       if (!dst)
-               dst = try_6to4(&iph6->daddr);
+       if (!dst) {
+               struct neighbour *neigh = NULL;
+
+               if (skb_dst(skb))
+                       neigh = skb_dst(skb)->neighbour;
+
+               if (neigh) {
+                       addr6 = (struct in6_addr *)&neigh->primary_key;
+                       dst = try_6to4(addr6);
+               }
+       }

        if (!dst) {
                struct neighbour *neigh = NULL;
```

A.8. allow ip fragmentation when using nopmtudisc to fix package loss

Beschreibung Erlaubt IPv4-Fragmentierung bei gesetztem `IP_DF`-Flag und verhindert damit Paketverlust in manchen Situationen (Kapitel 7.1.7, Abbildung 7.2).

Status **akzeptiert für Linux 2.6.32**:

http://patchwork.ozlabs.org/patch/32033/
http://git.kernel.org/?p=linux/kernel/git/davem/net-next-2.6.git;a=commit;h=8945a808

Code

Listing A.8: Patch: allow ip fragmentation when using nopmtudisc to fix package loss

```
diff --git a/net/ipv6/sit.c b/net/ipv6/sit.c
index 98b7327..386f1e9 100644
--- a/net/ipv6/sit.c
+++ b/net/ipv6/sit.c
@@ -778,7 +778,7 @@ static int ipip6_tunnel_xmit(struct sk_buff *skb, struct net_device *dev)
        iph->version            =       4;
        iph->ihl                =       sizeof(struct iphdr)>>2;
        if (mtu > IPV6_MIN_MTU)
-               iph->frag_off   =       htons(IP_DF);
+               iph->frag_off   =       tiph->frag_off;
        else
                iph->frag_off   =       0;
```

A.9. Behandlung von ICMPv4 Fehlern

Beschreibung Reicht ICMPv4 Fehler als IPv6 über den Tunnel weiter, sofern möglich, sodass Anwendungen angemessen auf Layer-2-Fehler reagieren können.

Status ***nicht*** **eingereicht**

Code

Listing A.9: Patch: Behandlung von ICMPv4 Fehlern

```
diff --git a/net/ipv6/sit.c b/net/ipv6/sit.c
index 98b7327..212e780 100644
--- a/net/ipv6/sit.c
+++ b/net/ipv6/sit.c
@@ -485,6 +485,10 @@ static int ipip6_err(struct sk_buff *skb, u32 info)
        const int code = icmp_hdr(skb)->code;
        struct ip_tunnel *t;
        int err;
+       const int hlen = iph->ihl << 2;
+       int rel_type = 0;
+       int rel_code = 0;
+       int rel_info = 0;

        switch (type) {
        default:
@@ -494,7 +498,6 @@ static int ipip6_err(struct sk_buff *skb, u32 info)
        case ICMP_DEST_UNREACH:
                switch (code) {
                case ICMP_SR_FAILED:
-               case ICMP_PORT_UNREACH:
                        /* Impossible event. */
                        return 0;
                case ICMP_FRAG_NEEDED:
@@ -505,12 +508,16 @@ static int ipip6_err(struct sk_buff *skb, u32 info)
                           rfc2003 contains "deep thoughts" about NET_UNREACH,
                           I believe they are just ether pollution. --ANK
                         */
+                       rel_type = ICMPV6_DEST_UNREACH;
+                       rel_code = ICMPV6_ADDR_UNREACH;
                        break;
                }
                break;
        case ICMP_TIME_EXCEEDED:
                if (code != ICMP_EXC_TTL)
```

```
                        return 0;
+               rel_type = ICMPV6_TIME_EXCEED;
+               rel_code = ICMPV6_EXC_HOPLIMIT;
                break;
        }

@@ -525,14 +532,40 @@ static int ipip6_err(struct sk_buff *skb, u32 info)
                goto out;

        err = 0;
-       if (t->parms.iph.ttl == 0 && type == ICMP_TIME_EXCEEDED)
-               goto out;
+
-       if (time_before(jiffies, t->err_time + IPTUNNEL_ERR_TIMEO))
-               t->err_count++;
-       else
-               t->err_count = 1;
-       t->err_time = jiffies;
+
+       /* Most routers return only 8 bytes of packet payload.
+           It means, that precise relaying of ICMP in the real
+           Internet is absolutely infeasible
+
+           If we get rfc1812 compliant ICMP packages, handle them properly,
+           otherwise set tunnel error state, _if_ it has a remote endpoint.
+           If it does not (6to4, isatap), possibly unrelated ICMPv6 messages
+           might be sent in ipip6_tunnel_xmit, so better ignore it.
+       */
+       if (skb->len - hlen >= sizeof(struct ipv6hdr)) {
+               struct ipv6hdr *iph6 = (struct ipv6hdr *)((u8*)iph + hlen);
+               struct sk_buff *skb2;
+
+               skb2 = skb_clone(skb, GFP_ATOMIC);
+               if (skb2 == NULL)
+                       goto out;
+               skb_pull(skb2, (u8 *)iph6 - skb->data);
+               skb_reset_network_header(skb2);
+               skb2->dev = t->dev;
+               icmpv6_send(skb2, rel_type, rel_code, rel_info, skb2->dev);
+
+               kfree_skb(skb2);
+       } else if (t->parms.iph.daddr) {
+               if (t->parms.iph.ttl == 0 && type == ICMP_TIME_EXCEEDED)
+                       goto out;
+               if (time_before(jiffies, t->err_time + IPTUNNEL_ERR_TIMEO))
+                       t->err_count++;
+               else
+                       t->err_count = 1;
+               t->err_time = jiffies;
+       }
 out:
        read_unlock(&ipip6_lock);
        return err;
```

A.10. fix off-by-one in ipip6_tunnel_get_prl

Beschreibung Behebt einen *off-by-one* Fehler beim Auslesen der PRL über einen entsprechenden `ioctl`. Ohne diesen Patch wird unter Umständen ein Eintrag mehr an den Userspace übergeben, als er speichern kann, worauf üblicherweise eine Zugriffsverletzung folgt, aber auch Kernelspeicher überschrieben werden kann.

Status **integriert für Linux 2.6.32**

`http://patchwork.ozlabs.org/patch/34469/`
`http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=298bf12d`

Code

Listing A.10: Patch: fix off-by-one in ipip6_tunnel_get_prl

```
diff --git a/net/ipv6/sit.c b/net/ipv6/sit.c
index d65e0c4..dbd19a7 100644
--- a/net/ipv6/sit.c
+++ b/net/ipv6/sit.c
@@ -274,7 +274,7 @@  static int ipip6_tunnel_get_prl(struct ip_tunnel *t,

        c = 0;
        for (prl = t->prl; prl; prl = prl->next) {
-               if (c > cmax)
+               if (c >= cmax)
                        break;
                if (kprl.addr != htonl(INADDR_ANY) && prl->addr != kprl.addr)
                        continue;
```

A.11. iproute2: ISATAP potential router list

Beschreibung Bringt *iproute2* den Umgang mit PRLs bei. Der Patch basiert auf: `http://osprey67.com/seal/iproute2_diff.v0_3.txt` (von Fred. L. Templin)

Status eingereicht

`http://saschahlusiak.de/linux/ip_tunnel_isatap_prl.patch`

Code

Listing A.11: Patch: iproute2: ISATAP potential router list

```
diff --git a/ip/iptunnel.c b/ip/iptunnel.c
index 7bee162..338d8bd 100644
--- a/ip/iptunnel.c
+++ b/ip/iptunnel.c
@@ -38,9 +38,10 @@ static void usage(void) __attribute__((noreturn));

 static void usage(void)
 {
-       fprintf(stderr, "Usage: ip tunnel { add | change | del | show } [ NAME ]\n");
+       fprintf(stderr, "Usage: ip tunnel { add | change | del | show | prl } [ NAME ]\n");
        fprintf(stderr, "          [ mode { ipip | gre | sit | isatap } ] [ remote ADDR ] [ local ADDR ]\n");
        fprintf(stderr, "          [ [i|o]seq ] [ [i|o]key KEY ] [ [i|o]csum ]\n");
+       fprintf(stderr, "          [ prl-default ADDR ] [ prl-nodefault ADDR ] [ prl-delete ADDR ]\n");
        fprintf(stderr, "          [ ttl TTL ] [ tos TOS ] [ [no]pmtudisc ] [ dev PHYS_DEV ]\n");
        fprintf(stderr, "\n");
        fprintf(stderr, "Where: NAME := STRING\n");
@@ -324,6 +325,25 @@ static void print_tunnel(struct ip_tunnel_parm *p)
                p->iph.daddr ? format_host(AF_INET, 4, &p->iph.daddr, s1, sizeof(s1))  : "any",
                p->iph.saddr ? rt_addr_n2a(AF_INET, 4, &p->iph.saddr, s2, sizeof(s2)) : "any");

+       if (p->i_flags & SIT_ISATAP) {
+               struct ip_tunnel_prl prl[16];
+               int i;
+
+               memset(prl, 0, sizeof(prl));
+               prl[0].datalen = sizeof(prl) - sizeof(prl[0]);
+               prl[0].addr = htonl(INADDR_ANY);
+
+               if (!tnl_prl_ioctl(SIOCGETPRL, p->name, prl))
+                       for (i = 1; i < sizeof(prl) / sizeof(prl[0]); i++)
+               {
+                       if (prl[i].addr != htonl(INADDR_ANY)) {
+                               printf(" %s %s ",
+                                       (prl[i].flags & PRL_DEFAULT) ? "pdr" : "pr",
+                                       format_host(AF_INET, 4, &prl[i].addr, s1, sizeof(s1)));
+                       }
+               }
+       }
+
        if (p->link) {
                char *n = tnl_ioctl_get_ifname(p->link);
                if (n)
```

```
@@ -464,6 +484,56 @@ static int do_show(int argc, char **argv)
        return 0;
 }

+static int do_prl(int argc, char **argv)
+{
+       struct ip_tunnel_prl p;
+       int count = 0;
+       int devname = 0;
+       int cmd = 0;
+       char medium[IFNAMSIZ];
+
+       memset(&p, 0, sizeof(p));
+       memset(&medium, 0, sizeof(medium));
+
+       while (argc > 0) {
+               if (strcmp(*argv, "prl-default") == 0) {
+                       NEXT_ARG();
+                       cmd = SIOCADDPRL;
+                       p.addr = get_addr32(*argv);
+                       p.flags |= PRL_DEFAULT;
+                       count++;
+               } else if (strcmp(*argv, "prl-nodefault") == 0) {
+                       NEXT_ARG();
+                       cmd = SIOCADDPRL;
+                       p.addr = get_addr32(*argv);
+                       count++;
+               } else if (strcmp(*argv, "prl-delete") == 0) {
+                       NEXT_ARG();
+                       cmd = SIOCDELPRL;
+                       p.addr = get_addr32(*argv);
+                       count++;
+               } else if (strcmp(*argv, "dev") == 0) {
+                       NEXT_ARG();
+                       strncpy(medium, *argv, IFNAMSIZ-1);
+                       devname++;
+               } else {
+                       fprintf(stderr,"%s: Invalid PRL parameter.\n", *argv);
+                       exit(-1);
+               }
+               if (count > 1) {
+                       fprintf(stderr,"One PRL entry at a time.\n");
+                       exit(-1);
+               }
+               argc--; argv++;
+       }
+       if (devname == 0) {
+               fprintf(stderr, "Must specify dev.\n");
+               exit(-1);
+       }
+
+       return tnl_prl_ioctl(cmd, medium, &p);
+}
+
 int do_iptunnel(int argc, char **argv)
 {
        switch (preferred_family) {
@@ -495,6 +565,8 @@ int do_iptunnel(int argc, char **argv)
                    matches(*argv, "lst") == 0 ||
                    matches(*argv, "list") == 0)
                        return do_show(argc-1, argv+1);
+               if (matches(*argv, "prl") == 0)
+                       return do_prl(argc-1, argv+1);
                if (matches(*argv, "help") == 0)
                        usage();
        } else
diff --git a/ip/tunnel.c b/ip/tunnel.c
index 104340d..d1296e6 100644
--- a/ip/tunnel.c
+++ b/ip/tunnel.c
@@ -167,3 +167,19 @@ int tnl_del_ioctl(const char *basedev, const char *name, void *p)
        close(fd);
        return err;
 }
+
+int tnl_prl_ioctl(int cmd, const char *name, void *p)
+{
+       struct ifreq ifr;
+       int fd;
+       int err;
+
+       strncpy(ifr.ifr_name, name, IFNAMSIZ);
+       ifr.ifr_ifru.ifru_data = p;
+       fd = socket(preferred_family, SOCK_DGRAM, 0);
```

```
+       err = ioctl(fd, cmd, &ifr);
+       if (err)
+               perror("ioctl");
+       close(fd);
+       return err;
+}
diff --git a/ip/tunnel.h b/ip/tunnel.h
index 1b58834..0661e27 100644
--- a/ip/tunnel.h
+++ b/ip/tunnel.h
@@ -31,5 +31,6 @@ char * tnl_ioctl_get_ifname(int idx);
 int tnl_get_ioctl(const char *basedev, void *p);
 int tnl_add_ioctl(int cmd, const char *basedev, const char *name, void *p);
 int tnl_del_ioctl(const char *basedev, const char *name, void *p);
+int tnl_prl_ioctl(int cmd, const char *name, void *p);

 #endif
diff --git a/man/man8/ip.8 b/man/man8/ip.8
index e948d03..e12008c 100644
--- a/man/man8/ip.8
+++ b/man/man8/ip.8
@@ -278,7 +278,7 @@ throw " | " unreachable " | " prohibit " | " blackhole " | " nat " ]"
 .IR STATE " ]"

 .ti -8
-.BR "ip tunnel" " { " add " | " change " | " del " | " show " }"
+.BR "ip tunnel" " { " add " | " change " | " del " | " show " | " prl " }"
 .RI "[ " NAME " ]"
 .br
 .RB "[ " mode
@@ -302,6 +302,13 @@ throw " | " unreachable " | " prohibit " | " blackhole " | " nat " ]"
 .B flowlabel
 .IR FLOWLABEL " ]"
 .br
+.RB "[ " prl-default
+.IR ADDR " ] [ "
+.B prl-nodefault
+.IR ADDR " ] [ "
+.B prl-delete
+.IR ADDR " ]"
+.br
 .RB "[ [" no "]" pmtudisc " ]"
 .RB "[ " dev
 .IR PHYS_DEV " ]"
@@ -2178,6 +2185,21 @@ set a fixed encapsulation limit.  Default is 4.
 .RB ( " only IPv6 tunnels " )
 set a fixed flowlabel.

+.SS ip tunnel prl - potential router list (ISATAP only)
+
+.TP
+.BI dev " NAME"
+mandatory device name.
+
+.TP
+.BI prl-default " ADDR"
+.TP
+.BI prl-nodefault " ADDR"
+.TP
+.BI prl-delete " ADDR"
+.RB "Add or delete " ADDR
+as a potential router or default router.
+
 .SS ip tunnel show - list tunnels
 This command has no arguments.
```

Abbildungsverzeichnis

2.1. OSI Modell 5
2.2. Natives IPv6 6

3.1. OSI Modell bei proto-41 Tunnel 9
3.2. Statischer proto-41 Tunnel 10
3.3. Mitschnitt einer proto-41 Kommunikation 11
3.4. Mitschnitt von Kommunikation über IPv4-kompatible Adresse 13
3.5. 6to4-Tunnel Mechanismus 14
3.6. 6over4 Tunnel Mechanismus über Multicast 16
3.7. ISATAP Tunnel Mechanismus 18
3.8. Teredo Tunnel über NAT 20
3.9. Tunnelbroker, z.B. SixXS 22

4.1. Adressklau bei ISATAP ohne Egressfilter 33
4.2. ISATAP Spoofing Szenarien 37
4.3. Kreisrouting bei ISATAP 40

5.1. Windows XP linklokale ISATAP Adresse 44
5.2. Windows XP ISATAP Stateless Autoconf 45
5.3. Windows XP Szenario asynchrone ISATAP Routen 47
5.4. Windows XP Mitschnitt von asynchronen ISATAP Routen 48
5.5. Funktionsweise von isatapd aus dem Miredo Projekt 54

7.1. Hashtabellen und verkettete Listen um Tunnel zu speichern 70
7.2. Problematisches PMTU Szenario bei SIT-Tunnel 89
7.3. IPv4 Routing Loop bei Tunnel 90

8.1. isatapd-0.9.5 - Initialisierung 100
8.2. isatapd-0.9.5 - PRL füllen 101
8.3. isatapd-0.9.5 - Hauptschleife 104
8.4. Trennung von Rechten in isatapd-0.9.6 107
8.5. isatapd-0.9.6 - Initialisierung 108
8.6. isatapd-0.9.6 - Hauptschleife 110
8.7. isatapd-0.9.6 - Router-Solicitation Schleife 112
8.8. isatapd-0.9.6 - Empfang von Router Advertisements 113
8.9. isatapd-0.9.6 - Verarbeiten von Router Advertisements 115

Listings

3.1. Einrichten eines statischen SIT Tunnels unter Linux 10
3.2. Verwendung IPv4-kompatibler IPv6 Adressen unter Linux 12
3.3. Einrichten von 6to4 unter Linux 14

4.1. Stehlen einer ISATAP Adresse über 6to4 35

5.1. 6to4 und Teredo unter Windows XP deaktivieren 43
5.2. ISATAP unter Windows XP konfigurieren 43
5.3. ICMPv4 Paket mit IPv6 als Payload 46
5.4. Empfang von ICMPv4 Nachrichten in Windows Vista 49
5.5. Konfiguration eines ISATAP Routers unter Cisco-IOS 50
5.6. Konfigurierte ISATAP Schnittstelle unter Cisco-IOS 50
5.7. Konfiguration einer korrekten ISATAP Adresse auf einem Cisco Router . 52
5.8. Konfiguration eines GIF-Tunnels für ISATAP unter FreeBSD 53
5.9. Manuelles Senden von Router Solicitations mit 'rdisc6' 55
5.10. Konfiguration eines SIT-Tunnels für ISATAP unter Linux 56
5.11. Konfiguration einer ISATAP Schnittstelle unter Linux mit iproute2 57
5.12. Konfiguration von Linux als ISATAP Router 59
5.13. radvd.conf für Linux als ISATAP Router 59

7.1. Definition struct ip_tunnel_parm (Userspace) 68
7.2. Definition struct iphdr (IPv4 Header) 68
7.3. Beispielcode: Anlegen eines SIT-Tunnels aus dem Userspace 70
7.4. Definition struct ip_tunnel_prl (Userspace) 71
7.5. Beispielcode: Hinzufügen eines PRL Eintrages aus dem Userspace 72
7.6. Definition struct ip_tunnel und struct ip_tunnel_prl_entry (Kernelspace) . 73
7.7. Interface-ID einer ISATAP Schnittstelle berechnen 74
7.8. Überprüfung des Absenders eines ISATAP Datenrahmens 80
7.9. Überprüfung auf IPv6 ISATAP Adresse 81
7.10. Ermittlung der IPv4-Adresse aus dem IPv6 Next-Hop bei ISATAP 84
7.11. Ermittlung der IPv4-Adresse eines 6to4 Ziels 85
7.12. Spezielle Routen auf einer 6to4 Schnittstelle 85

8.1. Router Solicitations aus dem Userspace senden 102
8.2. Definition struct PRLENTRY - Werte für jeden internen PDR 111
8.3. Beispielaufruf von isatapd-0.9.6 116

A.1. Patch: Fail to create tunnel if it already exists 123
A.2. Patch: strictly restrict incoming traffic to tunnel link device 124
A.3. Patch: ipip6_tunnel_del_prl: return err 125
A.4. Patch: refuse isatap eui64 for INADDR_ANY 125
A.5. Patch: stateless autoconf for isatap 126
A.6. Patch: Update stateless autoconf for isatap 128
A.7. Patch: 6to4: honour routing table 130
A.8. Patch: allow ip fragmentation when using nopmtudisc to fix package loss . 131
A.9. Patch: Behandlung von ICMPv4 Fehlern 131
A.10. Patch: fix off-by-one in ipip6_tunnel_get_prl 133
A.11. Patch: iproute2: ISATAP potential router list 133

Literaturverzeichnis

[1] rfc4291, IP Version 6 Addressing Architecture:
http://tools.ietf.org/html/rfc4291

[2] rfc4213, Basic Transition Mechanisms for IPv6 Hosts and Routers:
http://tools.ietf.org/html/rfc4213

[3] rfc3056, Connection of IPv6 Domains via IPv4 Clouds:
http://tools.ietf.org/html/rfc3056
rfc3964, Security Considerations for 6to4:
http://tools.ietf.org/html/rfc3964

[4] rfc2529, Transmission of IPv6 over IPv4 Domains without Explicit Tunnels:
http://tools.ietf.org/html/rfc2529

[5] ISATAP Draft Version 00:
http://tools.ietf.org/html/draft-ietf-ngtrans-isatap-00

[6] rfc4214, Intra-Site Automatic Tunnel Addressing Protocol (obsolete):
http://tools.ietf.org/html/rfc4214 (obsoleted by rfc5214)

[7] rfc5214, Intra-Site Automatic Tunnel Addressing Protocol:
http://tools.ietf.org/html/rfc5214

[8] rfc3330, Special-Use IPv4 Addresses:
http://tools.ietf.org/html/rfc3330

[9] rfc4380, Teredo: Tunneling IPv6 over UDP through Network Address Translations (NATs):
http://tools.ietf.org/html/rfc4380

[10] rfc4941, Privacy Extensions for Stateless Address Autoconfiguration in IPv6:
http://tools.ietf.org/html/rfc4941

[11] rfc4862, IPv6 Stateless Address Autoconfiguration:
http://tools.ietf.org/html/rfc4862

[12] rfc792, Internet Control Message Protocol (ICMPv4):
http://tools.ietf.org/html/rfc792

[13] rfc1812, Requirements for IP Version 4 Routers:
http://tools.ietf.org/html/rfc1812

[14] rfc4443, Internet Control Message Protocol (ICMPv6):
http://tools.ietf.org/html/rfc4443

[15] Gabi Nakibly, Michael Arov: Routing Loop Attacks using IPv6 Tunnels
http://www.usenix.org/events/woot09/tech/full_papers/nakibly.pdf

[16] Erste Unterstützung von ISATAP in Linux-2.6.25:
http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=c7dc89c0

[17] PRL Unterstützung für ISATAP in Linux-2.6.26:
http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=fadf6bf0

[18] rfc4429, Optimistic Duplicate Address Detection (DAD) for IPv6:
http://tools.ietf.org/html/rfc4429

[19] rfc4861, Neighbor Discovery for IP version 6 (IPv6):
http://tools.ietf.org/html/rfc4861

[20] rfc3068, An Anycast Prefix for 6to4 Relay Routers:
http://tools.ietf.org/html/rfc3068

[21] rfc1191, Path MTU Discovery (IPv4):
http://tools.ietf.org/html/rfc1191

[22] rfc1981, Path MTU Discovery for IP version 6
http://tools.ietf.org/html/rfc1981

[23] rfc4191, Default Router Preferences and More-Specific Routes
http://tools.ietf.org/html/rfc4191

[24] isatapd von Sascha Hlusiak
http://www.saschahlusiak.de/linux/isatap.htm