IPv6 – Pakete filtern mit Python

Johannes Hubertz
hubertz-it-consulting GmbH
Köln, Germany
<johannes@hubertz.de>

Zusammenfassung

Nach einer kurzen Einführung und Motivation, IPv6-Paketfilter zu nutzen, wird ein Konzept vorgestellt, diese Filter anhand von einfachen Definitions- und Regel-Dateien für beliebig viele Hosts und Router zu erzeugen. Die Mechanismen sind in Linux, BSD-Varianten, OpenSolaris und fast allen Routern enthalten, jedoch stets aufwendig und unterschiedlich zu handhaben. Python-Objekte nehmen Interface- und Routing-Tabellen als Basis, um je ein Shellscript mit den Filterkommandos für jedes Gerät zu generieren; die effektive Erzeugung und Anwendung im eigenen Netz spart Arbeitszeit und vereinfacht die Fehlersuche. Die Mächtigkeit der Objekte wird beispielhaft mit den produzierten ip6tables-Filtern am laufenden Gerät gezeigt. Die Programme sind unter GNU General Public License verfügbar; da pf.conf, ipfilter und ACLs den gleichen Zweck verfolgen, steht deren Implementierung nichts im Wege.

1 Einführung und Motivation

IPv6 wird seit mehr als 15 Jahren[1] spezifiziert, es wird nur selten genutzt und noch seltener effektiv gefiltert. Mittlerweile gibt es weit mehr als 200 RFC, die die Technik detailliert beschreiben. Host-, Switches-, Router,- und Firewall-Hersteller fertigen Geräte, die IPv6 Kommunikation versprechen.

Bis heute hat sich an vielen Stellen gezeigt, dass die 2^{32} Adressen für IPv4 deutlich zu wenig sind. NAT bietet sich als Notlösung zwar an, internen Verkehr zu realisieren, jedoch ist es immer auch mit Komplikationen verbunden. Spätestens bei Netz-Kopplungen werden private Adressen gemäß RFC 1918 problematisch, wenn zwei oder mehr Organisationen nicht disjunkte Adressräume verwenden. Immer wieder gerne wird NAT auch als Sicherheitsgewinn angepriesen, ein trügerisches und fragwürdiges Unterfangen.

Langsam aber stetig werden die offiziellen Adressen knapp, seit 2011 sind alle verbliebenen IPv4-Pools seitens der IANA vergeben; die RIRs[2] (ARIN, LACNIC, RIPE, AfriNIC, APNIC) werden bis 2012 neue Adressen vergeben können, bis auch ihre Pools leer sind. Experten sagen voraus, dass spätestens dann ein schwunghafter Handel mit IPv4 Adressen kurzzeitig einsetzen wird, und auf diese Weise IPv6 über kurz oder lang einfach preisgünstiger als IPv4 wird. Der Preis für IPv6 ist, Stand heute, aufgrund der notwendigen Schulungsmassnahmen und der notwendig neuen Geräte noch zu hoch. Der zuletzt angegebene Grund wird zunehmend hinfällig, da es immer weniger Geräte am Markt gibt, die nicht IPv6-konfigurabel sind.

IPv6 – Vor- und Nachteile

Mit IPv6 sollten für die kommenden Jahrzehnte ausreichend viele Adressen für jedermann vorhanden sein. Und nun die gute Nachricht zuerst: *IPv6 ist genauso sicher wie IPv4.* Die schlechte Nachricht: *IPv6 ist genauso unsicher wie IPv4.*

[1] RFC 1883, Dezember 1995, Quelle: http://www.ipv6forum.org.au/timeline.php
[2] Regional Internet Registry

Die RIRs sehen für jeden Endkunden ein /56 oder /48 vor, also 2^{72} oder 2^{80} Adressen. Damit gehören Engpässe der Vergangenheit an, selbst unter der Prämisse, dass für jedes Netz-Segment mindestens ein /64 zu vergeben ist, um eine automatische Adresszuteilung zu ermöglichen. Alle Empfehlungen gehen in diese Richtung: Möglichst keine kleineren Netze als /64 verwenden, selbst für Transfernetze mit nur 2 aktiven Geräten soll ein /64 verwendet werden. Eine stringente Netzplanung erscheint daher von Beginn an zwingend notwendig.

Ein wie auch immer geartetes NAT ist (heute) nicht vorgesehen. [3] Damit muss grundsätzlich von Ende–zu–Ende Kommunikation ausgegangen werden. ICMPv6 ist essentiell notwendig, ein vollständiges Filtern, wie es bei IPv4 möglich und oft üblich war, führt direkt zur Fehlfunktion.

Weitgehend automatisch soll die Adressvergabe am LAN stattfinden; DHCP kann, muss jedoch nicht verwendet werden. Die mindestens /64 grossen Segmente lassen Scanns, wie sie bei IPv4 oft und gerne verwendet wurden, apriori aussichtslos erscheinen; benötigte Bandbreiten für derart viel Traffic und, noch wesentlicher, der Zeitaufwand machen die Exploration deutlich schwieriger. Natürlich funktioniert ein Portscann bei einzelnen, manuell konfigurierten Hosts, die auch im DNS mit AAAA–RRs auftauchen, ebenso schnell wie bei einer IPv4–Adresse. Spätestens mit dieser Überlegung ist der Punkt erreicht, an dem Paket–Filter als ebenso sinnvoll und notwendig erachtet werden müssen wie bei IPv4. Einige andere Gründe sprechen zusätzlich dafür, Gegenargumente sind nur schwer aufzufinden, wenn überhaupt. So ist z.B. die durch Filterung erforderliche zusätzliche CPU–Last angesichts der am Markt vorhandenen CPU–Leistung absolut irrelevant.

Ein weiterer wichtiger Grund, Paket–Filter einzusetzen, ist der folgende: IPv6 wird durch einige, marktübliche Systeme benutzt, ohne dass der Benutzer hiervon Kenntnis erlangt. So nutzt bspw. die siebente Version des Redmonder GUI–Systems IPv6 als default. IPv4 dient als Fall-back. Nicht nur die vom Anwender unbemerkte Nutzung, sondern auch andere, nette Features lassen Filterung zwingend notwendig erscheinen. Was wir bei IPv4 als Source–Routing kennen und seit wenigstens 20 Jahren aus Sicherheitsgründen unterbunden werden sollte, ist in IPv6 unter anderem Namen wieder eingebaut worden: Routing–Header. Dieses Feature erlaubt, in einem IPv6–Paket hinter dem IP–Header weitere Header verschiedensten Charakters einzusetzen, die in der Reihenfolge zwar eingeschränkt sind, jedoch zwingend abgearbeitet werden müssen und zwar von jedem Host, der das Paket auf seiner Reise bearbeitet. Hierbei kommen Routing–Header Typ 0 und 2 der Funktionalität Source–Routing nahe. Will man sie unterbinden, kommt nur ein Paket–Filter mit entsprechender Funktionalität und Granularität infrage. Will man kein IPv6–Mobile nutzen, kann man einfach alle Routing–Header unterbinden, da Mobile jedoch nur *mit* Routing–Header Typ 2 funktioniert, muss dann genau der Typ 0 gefiltert werden, will man nicht Tür und Tor für Missbrauch öffnen.

Die Applikationen sind gleich; ob IPv4 oder IPv6 genutzt wird, sollte keine Rolle spielen. Lediglich die Konfiguration entscheidet über den Weg der Daten. Damit sind auch exakt die gleichen Schwachstellen der Applikationen im IPv6–Netzwerk verfügbar wie bei IPv4. Standardisierte Ende–zu–Ende Kommunikation bedeutet dann, dass die Anwendungsprogramme auf den benutzten Workstations mit ihrer lokalen IP–Adresse die Server direkt ansprechen, also kein NAT dazwischen die lokale Adresse verschleiert. Die Spuren in Logdateien lassen infolgedessen einfacher bestimmte, individuelle Auswertungen zu.

All diese mehr oder minder unschönen Eigenschaften lassen eine Filterung also als zwingend notwendig erscheinen. Bleibt noch die Frage offen, an welchen Stellen im Netzwerk Paket–Filter als Schutzmaßnahme aufgestellt werden sollen.

IPv6 – Filtern mit Bordmitteln

In Linux wirkt seit einigen Jahren die NetFilter-Architektur mit `iptables` und `ip6tables`, OpenSolaris, Net– und FreeBSD nutzen `ipfilter` in OpenBSD wird ausschließlich `pf` verwendet. In den neueren Systemen aus Redmond sind auch Paketfiltermechanismen eingebaut, deren Nutzung jedoch den normalen Anwender ebenso wie bei anderen Betriebssystemen überfordert. Alle diese Filtermecha-

[3] drafts zu NAT sind in Diskussion, jedoch sehr umstritten

nismen werden unterschiedlich konfiguriert, die Systeme beherrschen IPv6 einschließlich dynamischem Routing und Services, z.B. quagga als Routing–Daemon, bind9 als DNS–Server, Apache Webserver, NFS, ...

Ausgehend von alltäglichen Bedrohungen ist das Filtern nicht nur wegen der Möglichkeiten um Routing–Header Missbrauch auf jedem IPv6–Gerät sinnvoll, sondern insbesondere auch, um unerwünschte Kommunikation erstens zu unterbinden und zweitens auch die misslungenen Versuche einfach in den jeweiligen Logdateien zu entdecken. Die Konfiguration ist kompliziert, fehleranfällig und aufwendig, die üblichen Administratoren sind jedoch intelligent und infolgedessen faul: Ein geeignetes Werkzeug muss die Fleißarbeit minimieren. Dazu gibt es einige Ansätze, hier nur die wichtigen[4], welche einer freien Lizenz unterliegen:

- NetSPoC
- Firewall Builder
- iscs
- sspe

Sie unterstützen beliebig viele Knoten mit unterschiedlichen Denkmodellen und realisierten Konzepten. NetSPoC ist ein reines Kommandozeilenwerkzeug und geht von der Netzwerk–Topologie aus, Firewall Builder steuert als GUI die einzelnen Geräte, ohne Bezüge untereinander herzustellen, ebenfalls als GUI will iscs die Geräte vollständig administrieren.

Einige Erfahrung im Umgang mit Paket–Filtern mit IPv4 (sspe, simple security policy editor) gibt Grund zu der Annahme, dass auch mit IPv6 die Maßnahmen beherrschbar und einfach handhabbar sein müssen. Die bereits im Zusammenhang [1] gezeigten Vorstellungen lassen sich, bis auf die komplizierte Regulierung des NAT, fast 1:1 auch auf Geräte mit IPv6 anwenden. Die Programmierung muss jedoch aufgrund der seinerzeit impliziten Adressberechnungen von Grund auf neu erfolgen, was angesichts der empfohlenen und allerorten angewandten strikten Trennung von IPv4– und IPv6–Netzen keine negativen Konsequenzen nach sich zieht. Aus Gründen der besseren Lesbarkeit und der Tatsache, dass Programme viel öfter gelesen als geschrieben werden, fiel die Wahl auf Python als objektorientierte Programmiersprache.

Wünsche

Eine Organisation, die ein verzweigtes Netzwerk betreibt und dessen Sicherheit im Fokus hat, wird darum bemüht sein, ihr Administrations-Team auf dem laufenden Stand der Technik zu halten und alle Geräte im Netzwerk zu kennen, deren Funktionen genau zu kennen und zu dokumentieren. Da mit IPv6 vieles automatisch konfiguriert werden kann, sollte auch die Sicherheit, mindestens im Bereich Paket–Filter, weitestgehend automatisiert funktionieren, um Fehler im Tagesgeschäft so weit als möglich zu minimieren.

2 Konzept

Filtermechanismen sollen vom verwendeten Betriebssystem weitestgehend unabhängig beschrieben sein; die Beschreibung soll möglichst syntaktisch einfach sein, um Änderungen oder eine Revision so einfach wie möglich zu erlauben. Eine abstrakte, vom eingesetzten Filter-Gerät losgelöste Darstellung, ist in Firewalls üblich, jedoch sind diese zumeist nur mit einem einzigen Betriebssystem verfügbar. Eine zentrale Administration, die beliebig viele Geräte als Paketfilter konfiguriert, und so eine verteilte und konsistente „Policy“ unternehmensweit durchsetzt, gleichzeitig verschiedene Betriebssystem und deren Filtermechanismen nutzt, soll nachfolgend beschrieben werden.

Alle den Filtergenerator betreffenden Dateien sind in einem Verzeichnisbaum eines lokalen Benutzers abgelegt, der Zugang zu den Zielmaschinen per `ssh` muss außerhalb der `adm6`-Umgebung per

[4] Es besteht kein Anspruch auf Vollständigkeit, Hinweise werden gerne angenommen

ssh-key automatisch zur Verteilung bereitgestellt sein. Als Anwender sollte ein Benutzer auf dem zur Administration vorgesehenen Rechner angelegt sein, im weiteren Verlauf `adm6` genannt.

Außer der eigentlichen Konfigurationsdatei befinden sich alle dazugehörigen Dateien in einem Verzeichnisbaum, dessen Wurzel in eben dieser Konfiguration festgelegt wird (s.a. Abb. 1). Insbesondere im Unterverzeichnis `~/adm6/desc` ist für jede zu administrierende Maschine je ein Unterverzeichnis, in dem die dazugehörigen Dateien abgelegt werden: Routingtabelle und Interfacekonfiguration sind essentiell wichtig, um Filter generieren zu können. Die Ausgabe wird ebenfalls hier abgelegt.

Ort	Inhalt(e)
~/.adm6.conf	Konfigurations-Datei
~/adm6	adm6-Wurzelverzeichnis
~/adm6/bin	ausführbare Programme
~/adm6/etc	Definitionen, allg. Regeln
~/adm6/etc/rules	allgemeine Regeln
~/adm6/etc/hostnet6	Host- und Netz-Definitionen
~/adm6/desc	Verzeichnisse pro Zielmaschine
~/adm6/desc/dns	Maschinenspezifisches für dns
~/adm6/desc/dns/routing	Routingtabelle für dns
~/adm6/desc/dns/interfaces	Interfacekonfiguration für dns
~/adm6/desc/dns/output	Output für dns

Abbildung 1: adm6-Verzeichnisstruktur

In der Datei `~/etc/hostnet6` (Abb. 2) sind alle Definitionen abgelegt, eine Gruppierung ist durch Namensgleichheit möglich. Gruppen von Gruppen sind nicht vorgesehen, um die Komplexität der Programme einfach zu halten.

Name	CIDR Adresse	# Kommentar
any	2000::/3	# outside and inside
many	::/0	# anybody incl. mcast
admin	2001:db8:f002:1::23/128	# 1st admins workstation
admin	2001:db8:dada:3::23/128	# 2nd admins workstation
nagios	2001:db8:dada:3::5711/128	# nagios server
ns	2001:db8:f002:1::53/128	# 1st domain name server
ns	2001:db8:abba:2::53/128	# 2nd domain name server
www	2001:db8:f002:3::80/128	# internet web server
sfd	2001:db8:f002:1::2010/128	# software-freedom-day srv
intra	2001:db8:dead:beef::443/128	# intranet web server
office-cgn	2001:db8:abcd:2::/64	# office cologne
office-muc	2001:db8:abcd:3::/64	# office munich
office-bln	2001:db8:abcd:7::/64	# office berlin
offices	2001:db8:abcd:2::/64	# office cologne
offices	2001:db8:abcd:3::/64	# office munich
offices	2001:db8:abcd:7::/64	# office berlin
ripe-net	2001:610:240:22::c100:68b/128	# ripe.net web-server
www-kame-net	2001:200:dff:fff1:216:3eff:feb1:44d7/128	# orange.kame.net

Abbildung 2: Host- und Netzdefinitionen `~/adm6/etc/hostnet6`

Die Namen aus diesen Definitionen werden in der Regel-Datei (Abb. 3) benutzt, um in jeder Zeile (Regel) die Quelle und das Ziel zu benennen. Alle weiteren Informationen in der Zeile reglementieren den Verkehr zwischen Quell- und Zieladresse. Als Default-Policy kommt nur `Drop` infrage, diese Einstellung ist in den Programmen implizit gesetzt. Sie kann vom Anwender durch eine explizite Regel ausser Kraft gesetzt werden, welche einfach jede Art von Verkehr von Allen zu Allen erlaubt. Die Reihenfolge und die Bedeutung der Elemente pro Zeile ist fixiert, einfache Leerzeichen oder Tabulatoren trennen die Felder. Die Systematik ist einfach beschrieben, das „#“-Zeichen dient der Trennung

zwischen Inhalten und Kommentaren. Quelle und Ziel sind je eine oder mehrere Zeilen aus der Datei `~/etc/hostnet6` (Abb. 2). Mit der Spalte **Protokoll** ist nichts anderes gemeint als der *Next Header* im IPv6-Header. Anschließend sollte die Spalte **dport** der Einfachheit numerisch besetzt werden, damit spielt dann eine evtl. abweichende Datei `/etc/protocols` auf der Zielmaschine keine Rolle mehr. Das Feld „Aktion“ kann sinnvoll mit drei verschiedenen Werten besetzt werden: **accept, drop** oder **reject,** was fast selbsterklärend ist. Reject wird durch `reject --with-port-unreachable` in ip6tables abgebildet, drop ist zu bevorzugen, um Außenstehenden auch keine Informationen über unerwünschte Kommunikation zu geben. Verschiedene optionale Angaben sind in jeder Regel möglich, als Beispiel möge **NOIF** dienen; dieser Zusatz verhindert, dass entsprechend der Routingtabelle die Schnittstellen für Ein- und Ausgangsverkehr im ip6tables-Kommando mit vorgegeben werden. In jeder Zeile ist ein Kommentar möglich, die Erfahrung mit IPv4-Regelwerken lässt dies sinnvoll erscheinen.

Quelle	Ziel	Protokoll	dport	Aktion	Optionen	# Kommentar
admin	any	tcp	22	accept		#
any	ns	udp	53	accept		# test comment
any	ns	udp	53	accept	NOSTATE	# another comment
sfd	ns	udp	53	accept		#
sfd	many	tcp	80	accept		#
many	sfd	udp	123	accept	NTP	#
nagios	offices	udp	0:	accept	NOIF	# nagios may do everything
nagios	offices	tcp	0:	accept	NOIF	# nagios may do everything
office-cgn	office-muc	esp	all	accept		# fully trustworthy
office-muc	office-cgn	esp	all	accept		# fully trustworthy
office-cgn	office-bln	esp	all	accept		# fully trustworthy
office-bln	office-cgn	esp	all	accept		# fully trustworthy

Abbildung 3: Filter-Regeln ~/adm6/etc/rules

Als weiteres Beispiel sei **NOSTATE** angeführt, es verhindert, dass bei tcp und udp die Angabe `-m state -state NEW,ESTABLISHED,RELATED` hinzugesetzt wird. Der Zusatz **NTP** ändert die Source-Ports von `1024:` (default) auf `123`, um ntp-daemons den Verkehr mit Gleichen zu ermöglichen. Weitere Regelzusätze sind möglich, die Reihenfolge der Umsetzung in die ip6tables-Statements ist durch die derzeitige Programmgestaltung fest vorgegeben.

3 Python Objekte

Python wurde als Programmiersprache gewählt, da es neben einfacher Lesbarkeit vollständig implementierte Objektorientierung bietet. Ersteres erscheint wichtig, da Programme viel öfter gelesen als geschrieben werden. Zweiteres ist ebenso wichtig, um die Einfachheit des Programms auch bei komplexeren Anforderungen zu gewährleisten. „Keep it simple“ ist der wichtigste Leitsatz, soll doch das fertige Programm Sicherheit herstellen; hierzu ist die Nachvollziehbarkeit eine unabdingbare Voraussetzung.

Einige voneinander grundverschiedene Dinge sind also mit Python zu lösen, um der Aufgabenstellung gerecht zu werden:

- Konfiguration der Landschaft, Namen aller Zielmaschinen, GUI-Parameter?, ...
- Lesen aller Host- und Netzdefinitionen (Adressen und Namen)
- Lesen und Verarbeiten der Interface- und Routing-Dateien aller Zielrechner
- Lesen und Verarbeiten zusätzlicher Informationen der Zielrechner (paket-mangling etc.)
- Lesen und Auswerten der Filterregeln

Um eine IT-Landschaft in der adm6-Konfiguration abzubilden, sind verschiedene Wege gangbar. Einfache, flache ASCII-Dateien sind im Prinzip komplexen XML-Dateien inhaltlich gleichwertig, jedoch sind solche strukturierten Datencontainer nur mit speziellen Werkzeugen zu handhaben. Soll der angeforderten Einfachheit entsprochen werden, scheidet jede „nicht einfache“ Lösung aus. Zeilenorientiertes 7-Bit ASCII ist mit wahrscheinlich jedem Editor zu bearbeiten und bietet sich daher an.

Um die nachfolgenden Zusammenhänge nachvollziehen zu können, sind die Quelltexte hilfreich. Sie sind als git-Repository [2] verfügbar. Dank git lässt sich auch später noch genau der Entwicklungsstand herstellen, der zum Zeitpunkt des Schreibens dieses Dokuments vorlag.

3.1 Globale Konfiguration

Ausgehend von Geräten, die mit unterschiedlichen Betriebssystemen im Effekt gleiche Filterfunktionen bieten, um Netzverkehr zu regulieren, müssen dem Werkzeug einige Fakten mittels einer Konfigurationsdatei (Abb. 4) vorgegeben sein:

- Gerätename und Adresse
- Betriebssystem
- Router oder Host
- Aktivitätsstatus

Ein globaler Abschnitt dient generellen Einstellungen und Dingen, die persistent über verschiedene „Sitzungen“ vorgehalten werden sollen, z.B. Gerätekürzel, Arbeitsverzeichnis, etc., weitere Abschnitte für je ein Gerät folgen, in denen Name, Kurzbechreibung und die Adresse abgelegt ist, unter der das Gerät erreichbar ist. Die Datei ist bewusst „python-freundlich“ gehalten, d.h. die Syntax entspricht an manchen

```
1  [global]
2  version = 0.1
3  timestamp = 2011-05-13
4  home = /home/hans/adm6/
5  devices = r-ex, ns, obi-wan
6  software = ['Debian', 'OpenBSD', 'OpenSolaris']
7
8  [device#r-ex]
9  desc = external router via ISP to the world
10 os = Debian GNU/Linux, Lenny
11 ip = 2001:db8:f002:1::1
12 fwd = 1
13 asymmetric = 1
14 active = 1
15
16 [device#ns]
17 desc = company dns server
18 os = Debian GNU/Linux, Lenny
19 ip = 2001:db8:f002:1::23
20 fwd = 0
21 active = 1
22
23 [device#obi-wan]
24 desc = gif-tunnel from company to home
25 os = OpenBSD 4.5
26 ip = 2001:db8:f002:1::2
27 fwd = 0
28 active = 1
```

Abbildung 4: adm6-Konfiguration **~/.adm6.conf**

Stellen der pythonischen Schreibweise, z.B. bei Listen. Diese Vorgehensweise ist zwar einfach, aber vielleicht nicht optimal, möglicherweise ist sie sogar unbrauchbar für viele Geräte. Für größere Anzahlen zu administrierender Geräte sollte die Datenhaltung dieser „Stammdaten“ vielleicht in eine Datenbank ausgelagert werden, die Einbindung in Python ist mit einfachen Mitteln realisierbar.

Python bietet mit seinem Konzept „Batteries included“ nach der Installation bereits die Klasse ConfigParser, aus der sich eigene Klassen mit projektspezifischen Erweiterungen ableiten lassen. Um die

gezeigte Konfigurationsdatei (Abb. 4) im Programm zu lesen und auszuwerten, ist **Adm6ConfigParser** als Python-Klasse implementiert. Deren verschiedene Methoden geben jeweils konfigurierte Einzelheiten der Geräte zurück. Ein Schreiben der Konfiguration ist zunächst nicht vorgesehen, hierzu wäre auch eine wesentliche Erweiterung nötig, um die Reihenfolge der Abschnitte in der Datei nicht dem Zufall zu überlassen. Derzeit ist die Pflege dieser Datei nicht per GUI realisiert, jedoch denkbar.

3.2 Netz- und Host-Definitionen

Alle im Zusammenhang mit Filterregeln verwendete Quell- und Zieladressen werden mit beliebigem Namen notiert und in einer Datei **`hostnet6`** vorgehalten. Dabei wird eine Gruppe von Geräten (oder Netzen) durch den gleichen Namen gekennzeichnet. Eine Klasse **`HostNet6`** dient der objektorientierten Nutzung des Dateiinhaltes, die enthaltenen Kommentare werden ignoriert. Intern entsteht durch das Lesen der Datei eine Liste, die als Elemente eine 2-elementige Liste der gelesenen Namen und die Adressen als Instanz eines IPv6Network-Objektes enthält. Dieses Objekt entstammt dem Python-Modul **`ipaddr.py`**, welches durch Google™ der Community bereitgestellt wird. Durch das Exceptionhandling von Python ist es einfach, falsche Einträge in der Datei zu überlesen und so nur (syntaktisch) korrekte Eingabedaten zu verwerten. Mit der Instantiierung des Objektes wird eine leere Liste angelegt, so ist ein mehrfaches Einlesen verschiedener Dateien gleichen Formates nacheinander möglich, im Ergebnis sind dann alle Elemente aller Dateien in der Liste genau einmal vorhanden. So kann es für jedes Gerät eine eigene Definitionsdatei geben, mindestens muss nur genau eine vorhanden sein und kann für alle Geräte alle Definitionen enthalten. Die so gegebene Flexibilität hat in verschiedenen Szenarien ihren Grund. Nach dem Einlesen mit der Methode **`append`** sind die Namen und Adressen in sortierter Reihenfolge zu verwenden, mit der Methode **`show_hostnet6`** kann der gelesene Inhalt in Gänze wieder nach `stdout` ausgegeben werden. Die Methode **`get_addrs`** liefert alle Adressen zu einem gegebenen Namen als Liste von IPv6Network-Objekten.

3.3 Geräte

Eine weitere Klasse **ThisDevice** implementiert die pro Gerät notwendige Datenhaltung, hier sind neben dem Namen und dem Betriebssystem auch Interface-Konfigurationen und Routingtabllen zu nennen. Sie wird im weiteren Verlauf in Ausschnitten gezeigt. Die `__init__`-Funktion sorgt für das Lesen aller Informationen zum Gerät bei der Instantiierung des Objektes. Eine Instanz der Klasse Adm6ConfigParser wird übergeben und dazu benutzt, um in dieser Funktion die Werte aus der Konfiguration zu Lesen.

Interface- und Routing-Informationen

Die Interface-Konfiguration der jeweiligen Geräte wird mit unterschiedlichen Methoden je nach Betriebssystem eingelesen. Sowohl bei Linux (Debian GNU/Linux dient als Beispiel) als auch bei OpenBSD wird die Ausgabe des Befehls **`ifconfig`** als Eingabe ausgewertet. Andere Betriebssystem sollten über ähnliche Werkzeuge verfügen.

Das Lesen der Routingtabllen macht schon deutlich mehr Aufwand, sind diese in den verschiedenen Varianten auf Linux und OpenBSD doch sehr unterschiedlich. Bei Linux wird der Kommandozeilenbefehl **`ip -6 route show`** genutzt, bei OpenBSD **`route -n show`**. Der vorangestellt Unterstrich im Methodennamen deutet an, dass diese nicht von Aussen genutzt werden soll.

Das Geheimnis der Implementierung mit derart wenigen Zeilen liegt in der erneuten Verwendung des Python-Moduls **`ipaddr.py`**. Die Klasse IPv6Network stellt auf einfache Weise einen Test zur Verfügung, der eine Zeichenkette in Hinblick auf eine IPv6-Netzwerkadresse in CIDR-Schreibweise validiert. Dieses Objekt wird in der Funktion `_bsd_routingtab_line` jeweils in Verbindung mit dem Python Exceptionhandling genutzt, um in der Fülle des Inhaltes die Nadel im Heuhaufen zu finden ...

Regeln einlesen

Nachdem im Objekt die Interface- und Routing-Informationen gespeichert sind, müssen auch die Filterregeln gelesen werden; die beiden Methoden **`read_rules`** und **`read_one_rule`** realisieren dies.

Mit der ersten Methode werden im Verzeichnis für das Gerät alle diejenigen Dateien gesucht, deren Namen mit einer Ziffer beginnt und mit der Zeichenkette „`rules`“ endet. So kann per „symlink“ eine oder mehrere Dateien für alle Geräte gleich sein, und auch jedes Gerät eigene Regeldateien haben. In der zweiten Methode wird eine Regeldatei eingelesen, in der Dritten eine Zeile behandelt. Bei der Initialisierung wurde eine leere Liste angelegt, jede Regel wird dieser Liste hinzugefügt.

Einige Methoden zur Ausgabe der eingelesenen Werte sind ebenfalls vorhanden; sie dienten vornehmlich der Kontrolle während der Klassen-Entwicklung.

Shellscript erzeugen

Alle diese nun im Objekt abgelegten Informationen werden benötigt, um die zu einer Zeile aus der Regeldatei je nach Betriebssystem ip6tables, pf.conf oder anderweitige Access-Listen zu generieren. Der besseren Übersicht halber wurde alle Funktionalität der eigentlichen Filterung in ein separates Python-Modul filter6.py ausgelagert, daher sei hier nur erwähnt, dass die Methode **`do_rules`** über die Liste der Regeln iteriert und mit Methoden des Filter-Objektes systemspezifische Kommandos generiert.

In der Klasse **`ThisDevice`** sind zwei weitere Methoden vorhanden, die nur internen Berechnungen dienen, **`look_for`** zuerst: Sie ermittelt, ob zu einer gegebenen Adresse im Gerät eine Route vorhanden ist und, wenn ja, welche Zeile der Routingtabelle betroffen ist. Die hierbei benutzte Methode __contains ist aus der Klasse IPv6Network geerbt. Im Erfolgsfall wird Interfacename und Zeilennummer aus der Routingtabelle zurückgegeben. Beide Werte dienen dem Vergleich, ob die durch die Regel vorgegebene Verkehrsbeziehung durch des Gerät verläuft, da nur dann Filterkommandos generiert werden müssen. Andernfalls ist davon auszugehen, dass der Verkehr am Gerät vorbeigeht. Mit der zweiten Methode **`address_is_own`** wird ermittelt, ob eine gegebene Adresse im Gerät konfiguriert ist, also in der Interfacekonfigration enthalten ist. Beide Methoden dienen der Optimierung, um keine unnötigen Filterkommandos zu erzeugen; dies ist noch völlig unabhängig vom verwendeten Betriebssystem des Gerätes.

Das Zusammenspiel zwischen ThisDevice und Filter6 ist mit den Aufrufen in der Hauptschleife dargestellt. Die Methode **`do_rules`** wird mit einer Instanz der Klasse **`IPv6_Filter`** aufgerufen. Diese arbeitet die Regeln ab, indem sie die erforderlichen Berechnungen anstellt und daraus die Ausgabe generiert. Hierzu wird in der Methode **`ThisDevice.do_rules`** die Instanz des Filter6-Objektes genutzt, um die Regeln in ein IPv6_Filter-Objekt einzutragen. Wenn das abgeschlossen ist, wird mit **`IPv6_Filter.mach_output`** alle Ausgabe zu dem Gerät erzeugt. Im Verzeichnis `~/adm6/etc` befinden sich pro unterstützter Betriebssystemvariante je eine Anfangs- und Ende-Datei, bspw. für Debian: `~/adm6/etc/Debian-header` und `-footer`. Diese enthalten vorbereitende und abschließende Kommandos, die dem Einbringen der Filterkommandos dienen. So sind in der Debian-header Datei einige Abkürzungen definiert, die Default-Policy wird gesetzt und einzelne, zwingend erforderliche ICMPv6-Filter werden vorgegeben, um z.B. IPv6-Autoconfiguration nicht zu verhindern. Hier fehlt noch die Erfahrung mit den Details, um exakt auf die wirklich notwendigen Filtermaßnahmen zu beschränken. Routing-Header Typ 0 und 2 werden erst im Nachspann ausgeblendet, um Regeln zu ermöglichen, die gezielt solche wieder erlauben können, aber nicht vorhanden sein müssen.

Da es sich um Shellscripts handelt, sind diese leicht an anderen Bedarf anzupassen. Der Trick mit dem Umbenennen der Chains hat den Vorteil, dass die Policy ständig auf „drop“ bleiben kann, erst die Chains mit den alten und dann die Chains mit den neu eingebrachten Regeln werden umbenannt. Da gleichzeitig mit dem Ändern des jeweiligen Chain-Namens auch der Verweis auf diese so umbenannt wird, dass die Verkettung erhalten bleibt, sind anschließend nur die (kurzen) INPUT-, OUTPUT- und FORWARD-Chains zu verwerfen und dann von Grund auf neu aufzubauen. Da sie kurz sind, dauert dies nur wenige Millisekunden, in denen (je nach Umfeld) auf die vollständige Filter-Funktionalität sicherlich verzichtet werden kann. In Abhängigkeit vom erlaubten Verkehr muss man allerdings darauf achten, dass

keine **ICMPv6-unreachable**-Pakete generiert werden oder diese in den Endgeräten zielgerichtet gefiltert werden, so dass die Applikationen mit Retries über den kurzen Zwischenfall hinwegkommen. Ausgehend von Erfahrungen im IPv4-Umfeld können pro Gerät noch je eine Datei nach dem Header und vor dem Footer eingebunden werden. Diese aus dem sspe-Konzept übernommene Absicht, damit **paket-mangling** zu ermöglichen, ist durch reale Anforderungen hinsichtlich **DNAT** überrollt worden. Nun gibt es in IPv6 (Stand heute) zwar kein NAT, aber solche Dateieinschübe erlauben zusätzliche Flexibilität in Abhängigkeit vom Endgerät. Und auch von den Filterfunktionen unabhängige Skripts lassen sich damit aufrufen. Shell ist eben flexibel.

Somit ergibt sich ein Baukasten aus verschiedenen und unterschiedlichen Aufgaben zugeordneten Schnipseln aus Shellscripts, die aus Header, mangle-startup, den eigentlichen Regelsatz gefolgt von mangle-endup und Footer. Auf den zentralen Abschnitt, den Regelsatz, wird im Folgenden weiter eingegangen.

Regelwerk abarbeiten

Die Iteration über die Liste der Filterregeln erfolgt in einer „for-Schleife“ Die darin benutzte Methode **IPv6_Filter.final_this_rule** erzeugt aus der Liste **self.rule** als Erstes ein Objekt der Klasse **IPv6_Filter_Rule** mit dem kurzen Namen „r“, welche aus der Klasse UserDict abgeleitet ist. Diese Vorgehensweise macht sich zunutze, dass in einem solchen Python-Dictionary assoziativ Pythonobjekte gespeichert werden kann.

Einige Einstellungen wie Gerätename, Name der Ausgabedatei, der Text der Regel, Quell- und Ziel-Adresse wie auch diverse Optionen werden direkt im Dictionary gespeichert. Bestimmte Optionen, wie z.B. **NOSTATE** und **NOIF**, und daraus implizierte Eigenschaften können hier schon endgültig gesetzt werden; andere, wie z.B. **FORCED,** führen durch die Reihenfolge in der Programmierung zu folgenreichen Zwischenwerten wie **travers, i_am_s, i_am_d**. Innerhalb des Regeltextes spielt die Reihenfolge der Optionen keine Rolle.

Ein Vorteil mit der gewählten objektorientierten Methodik ist, dass die Methode **__repr__** durch den Anwender definiert werden kann und er so exakt bestimmt, was und wie bei der Ausgabe des Objektes tatsächlich ausgegeben wird.

Die Objekteigenschaften „DisplayList“ und „CommentList“ legen so fest, welche Attribute in welcher Reihenfolge ausgegeben werden; am Beispiel (Abb. 5) wird deutlich, welche Ausgabe durch eine Regel erzeugt wird. Insbesondere die Kommentare darin ermöglichen eine effektive Überprüfung bzw. Entstörung der Python-Klassen. Die generierten ip6tables-Kommandos werden so leicht kontrollier-, bzw. nachvollziehbar, warum sie so erzeugt wurden. Des weiteren kommen sie der Generierung von OpenBSD oder OpenSolaris Filtern zugute, wenn vor einer Zeile ein entsprechendes ip6tables-Kommando als Kommentar zusätzlich ausgegeben wird. So ist ein Vergleich jederzeit schnell und einfach möglich. Unabhängig von der Zielplattform zeigt die Methode **IPv6_Filter_Rule.produce** den einfach strukturierten Verteilmechanismus. Der Wert der Variablen **IPv6_Filter_Rule.[`OS`]** entscheidet, welcher Produzent aufgerufen wird. Dieser, bspw. **produce_Debian,** wird beim Aufruf durch Parameter u. a. gesteuert, ob tatsächlich die ip6tables-Kommandos als Shellscript Befehle oder nur als Kommentar ausgegeben werden. Also sind die Kommentare in gleicher Weise der Weiterentwicklung wie auch im jetzt schon produktiven Betrieb zur Entstörung hilfreich.

Die Programmierung für OpenBSD und OpenSolaris kann beginnen, der Rahmen ist gezeigt. Eine **pf.conf** kann bereits erzeugt werden, Tests stehen aber noch aus. Eine korrekte Funktion kann derzeit noch nicht erwartet werden. Ebenso kann der Anfang, der bereits für proprietäre Systeme aus Redmond gemacht wurde, nicht darüber hinwegtäuschen, dass noch ein weiter Weg bis zu einer vollständigen Funktionalität zurückzulegen ist.

```
# ------------------------------------------------------------------------------ #
# Rule-Nr        : 3                                                             #
# Pair-Nr        : 1                                                             #
# System-Name    : r-ex                                                          #
# OS             : Debian                                                        #
# RuleText       : ['any', 'ns', 'udp', '53', 'accept', 'NOSTATE']               #
# Source         : ::/0                                                          #
# Destin         : 2001:db8:1:1::23/128                                          #
# Protocol       : udp                                                           #
# sport          : 1024:                                                         #
# dport          : 53                                                            #
# Action         : accept                                                        #
# nonew          : False                                                         #
# noif           : False                                                         #
# nostate        : True                                                          #
# insec          : False                                                         #
# i_am_s         : None                                                          #
# i_am_d         : None                                                          #
# travers        : True                                                          #
# source-if      : eth1                                                          #
# source-rn      : 27                                                            #
# src-linklocal  : False                                                         #
# src-multicast  : False                                                         #
# destin-if      : eth3                                                          #
# destin-rn      : 1                                                             #
# dst-linklocal  : False                                                         #
# dst-multicast  : False                                                         #
# producing ip6tables commands for rule: 3 Pair:  1
/sbin/ip6tables -A   forward_new  -o eth1 -s ::/0 -d 2001:db8:1:1::23/128 \
        -p udp --sport 1024: --dport 53 -j ACCEPT
/sbin/ip6tables -A   forward_new  -i eth1 -d ::/0 -s 2001:db8:1:1::23/128 \
        -p udp --dport 1024: --sport 53 -j ACCEPT
```

Abbildung 5: Beispiel: Ausgabeformat einer Regel für Debian

4 Betrieb

Im November 2010 wurde das erste Gerät (Debian GNU/Linux) mit IPv6-Paketfiltern ausgestattet: Ein Webserver war mit tcp/80 für alle Welt, mit tcp/22 nur für ausgewählte Quelladressen erreichbar. DNS-Abfragen zum eigenen Nameserver waren zunächst der einzige abgehende Verkehr. Damit funktionierte die Maschine, jedoch die wunderbaren Debian Update/Upgrade Mechanismen scheiterten. Dank der simplen Gestaltung der Filter war dies jedoch schnell behoben, tcp/80 in alle Welt geschwind zugelassen und das Gerät blieb aktuell. Auf Dauer lief die Uhr ungenau, doch auch dieser Missstand war schnell behoben: udp/123 zu allen Adressen war schnell geöffnet. Die Logs zeigten bald ständige Versuche, das Gerät per ssh zu erreichen. Dies blieb nicht nur wegen der Filter unproblematisch, sondern auch wegen der sshd Einstellung, nur bekannte Schlüssel zur Authentisierung zu nutzen, dauerhaft erfolgreich wie verkehrsarm als Verteidigungsstrategie. Bekanntlich lassen sich 2048-Bit RSA-Schlüssel deutlich schwerer erraten als 8-Buchstaben oder auch längere Passwörter.

Aufgrund der Logs wurden noch Regeln hinzugefügt, ankommende tcp/445 und udp/137 Pakete zu verwerfen und damit nicht weiterhin in den Logs zu zeigen. Einige andere, häufig verwendete Zielports kamen im Laufe der Zeit hinzu, letztlich blieben in den Logs statistisch verteilte Ports sichtbar und manchmal Fragmente von udp-Paketen. Mangels eines schlüssigen anderen Konzeptes, Fragmente zu handhaben, bleibt derzeit nur das Verwerfen. Dies stellt sicherlich nur eine suboptimale Lösung dar, sie vermeidet jedoch, zusätzliche Komplexität und damit Unsicherheit zuzulassen.

Ende Januar 2011 folgte das zweite Gerät: Der eigene Mail- und DNS-Server mit seiner etwas komplexeren Kommunikationsstruktur. Die Generierung der Paketfilter mit zusätzlichen Regeln mit anderen tcp- und udp-Ports stellte jedoch keine Herausforderung dar, der Regelsatz stieg von ca. 20 Regeln auf etwa 35 Regeln an. Dies ist keine Größenordnung, in der wegen der zusätzlichen CPU-Last mit Problemen zu rechnen ist; die Sache bleibt überschaubar und folgt damit dem ursprünglichen Konzept: Keep it simple!

Mitte März 2011 wurde der eigene IPv6-Router ebenfalls mit Filterregeln versehen, damit stieg die Anzahl Regeln auf etwa 45 an, einige waren nur für Tests und dienten Weiterentwicklung der Pythonscripts. Im Unterschied zu den beiden Endgeräten wurde hier Verkehr gefiltert, der den Router passiert, der Einbau verlief weitgehend unproblematisch und führte schnell zu kleineren Änderungen und Korrek-

turen der Python-Objekte für ip6tables. Auch hier wurden spezielle Regeln eingebaut, um die Logdateien klein zu halten und uninteressante Pakete, z. B. auf tcp/22, tcp/445, udp/137 u.s.w. nur zu verwerfen.

Als im Oktober 2011 ein Providerwechsel stattfand, wurde am neuen Ort asymmetrisches Routing aufgrund der Vorgaben des ISP und der eigenen Vorstellungen nötig. Dies führte zu einer weiteren, zusätzlichen Option in der Konfigurationsdatei und den damit verbundenen Abhängigkeiten. Für Linux, bzw. NetFilter ergab sich nur eine Konsequenz: Stateful Inspection kann bei asymmetrischem Routing nicht funktionieren, daher folgt für ein Gerät mit konfiguriertem asymmetrischen Routing lediglich, dass implizit auf jeder Regel die Option **`NOSTATE`** gesetzt wird. So können die Regeln für die Geräte gemeinsam genutzt werden und somit war auch dieser Spezialfall gelöst. Falls ein Endgerät über mehrere IPv6-Schnittstellen verfügen sollte und eingehende Pakete aufgrund des Routings Antworten auf der anderen Schnittstelle zur Folge haben, muss evtl. die Option **`NOIF`** noch zusätzlich oder anstelle eingebaut werden, um die stateful-inspection eingeschaltet lassen zu können. Es bleibt jedoch einer späteren Implementierung vorbehalten, in solchen Sonderfällen zwischen Forwarding und Input/Output entsprechend zu unterscheiden.

5 Zusammenfassung

Gezeigt wurde eine Methode, IPv6-Paketfilter aus einem Regelsatz konsistent für viele Zielsysteme zu erzeugen, um im Unternehmen auf allen Geräten einheitlich eine Policy zu nutzen. Die Gründe liegen auf der Hand: Wenn IPv6 genutzt wird, sollte es aufgrund der neuen (nicht immer anderen) Risiken gegenüber IPv4 nach Möglichkeit gefiltert werden. Dies sollte in jedem Gerät stattfinden, alle derzeit verfügbaren Betriebssysteme freier oder kommerzieller Natur stellen die notwendigen Mechanismen zur Verfügung. Die Daten eines Unternehmens und dessen Geräte sollten bestmöglich geschützt werden, dies sowohl in Richtung ihrer Inhalte, als auch im Hinblick auf ihr Missbrauchspotenzial gegenüber anderen, möglicherweise auch außerhalb des Unternehmens befindlichen Computern und Daten. Das Ende-zu-Ende Konzept aller IPv6-basierten Kommunikation lässt Filterung auf jeder beteiligten Komponente sinnvoll erscheinen, das vorgestellte Generatorkonzept macht eine einheitliche Policy bzgl. der Filterung einfach realisierbar. Wenngleich es derzeit nur für die Linux NetFilter-Architektur funktioniert und produktiv einsetzbar ist, ist wegen der verwendeten GNU/General Public License der Weg frei zur Mitarbeit und Realisierung auch für proprietäre Systeme. Systeme mit Cisco IOS™ und Windows™ bieten auch starke Bordmittel, um IPv6 Pakete auch in ihrem Zusammenhang (stateful) zu filtern, ergo spricht nichts gegen deren Nutzung im Unternehmen mit einer einheitlichen Policy, diese verspricht mindestens, keine Inkonsistenzen zu enthalten und durch die einheitliche Handhabung auch endbenutzerkompatibel zu sein.

Ausdrücklich erwünscht sind Kritik und Mitarbeit interessierter IPv6- und Pythonkenner, um die Sache gemeinsam voranzutreiben und damit das Netzwerk der Zukunft etwas zuverlässiger zu machen. Einige Aufgaben sind noch zu erledigen:

- **Repository:** einfaches Rollback, Nachvollziehbarkeit
- **Crypto:** verschlüsselte Skripts verhindern Änderungen
- **Datenbank:** beliebige Skalierbarkeit
- **GUI:** einfachere Bedienung

Literatur

[1] Johannes Hubertz, *sspe: simple security policy editor*, GUUG FFG Proceedings 2006, ISBN 978-3-86541-145-7

[2] Quelltexte: http://evolvis.org/projects/adm6