Seminar “User Mode Linux”: Netfilter

Tassilo Horn
heimdall@uni-koblenz.de

03.02.2006

Inhaltsverzeichnis

1 Einführung **3**
1.1 Kurzbeschreibung 3
1.2 Historisches 3

2 Aufbau des Netfilter-Frameworks **4**
2.1 Tabellen (tables) 4
2.2 Ketten (chains) 4
2.3 Verarbeitungsregeln 6

3 Netfilter-Konfiguration mit iptables **8**
3.1 Kommando-Optionen 9
3.2 Muster und Ziel 10
3.3 weitere Tools 11

4 Beispiele **13**
4.1 Logging von Paketen 13
4.2 NAT im Heimnetz 14
4.3 Port-Forwarding / Load-Sharing 16
4.4 BitTorrent 17

5 Das VNUML-Beispiel **18**
5.1 Konfiguration der UML-Guests 18
5.2 Was soll erreicht werden? 19
5.3 Die Konfiguration, die zum Ziel führt 19

6 Literatur **23**

1 Einführung

1.1 Kurzbeschreibung

- **Netfilter**: Kernel-Framework zur Filterung & Manipulation von Netzwerkpaketen
- **`iptables`**: Userland-Tool zur Netfilter-Konfiguration
- **Main Features:**
  - Network Address [and Port] Translation (NA[P]T)
  - Packet Filtering
  - Packet Mangling

Unter Netfilter versteht man ein Framework innerhalb des Linux-Kernels zur Filterung und Manipulation von Netzwerkpaketen. Das zugehörige Userland-Tool zur Netfilter-Konfiguration heißt `iptables`. Beide Begriffe werden häufig synonym verwendet für die Gesamtheit aus Kernel- und Userspacebestandteilen. Der Einfachheit halber wird die auch in dieser Ausarbeitung so gemacht.

Die wichtigsten Features sind Network Address Translation (NAT), Paketfilterung (Packet Filtering) und andere Paketmanipulationen (Packet Mangling).

1.2 Historisches

Geschichte von Netfilter

- Linux 1.0 (1994): Portierung des BSD-Paketfilters `ipfw`
- Linux 2.0 (1996): Erweiterung & Überarbeitung $\Rightarrow$ `ipfwadm`
- Linux 2.2 (1999): Erweiterung & Überarbeitung $\Rightarrow$ `ipchains`
- Linux 2.4 (2001) bis heute: Erweiterung & Überarbeitung $\Rightarrow$ `iptables`

Plege & Weiterentwicklung durch das 1999 gegründete Netfilter-Projekt[1].

Bereits die Linux-Version 1.0 besaß einen Paketfilter, der von BSD portiert wurde. Im Laufe der Jahre wurde dieser mehrmals erweitert und überarbeitet. Seit 1999 wird Netfilter vom Netfilter-Projekt gepflegt und weiterentwickelt.

[1] `http://www.netfilter.org`

2 Aufbau des Netfilter-Frameworks

2.1 Tabellen (tables)

Netfilter gruppiert alle Verarbeitungsregeln gemäß ihrer Funktion in drei Tabellen:

- **filter**: Paketfilter aller Art (default)
- **nat**: Network Address Translation

  Diese Tabelle wird bei allen Paketen, die eine neue Verbindung erzeugen, konsultiert.
- **mangle**: Paketmanipulationen

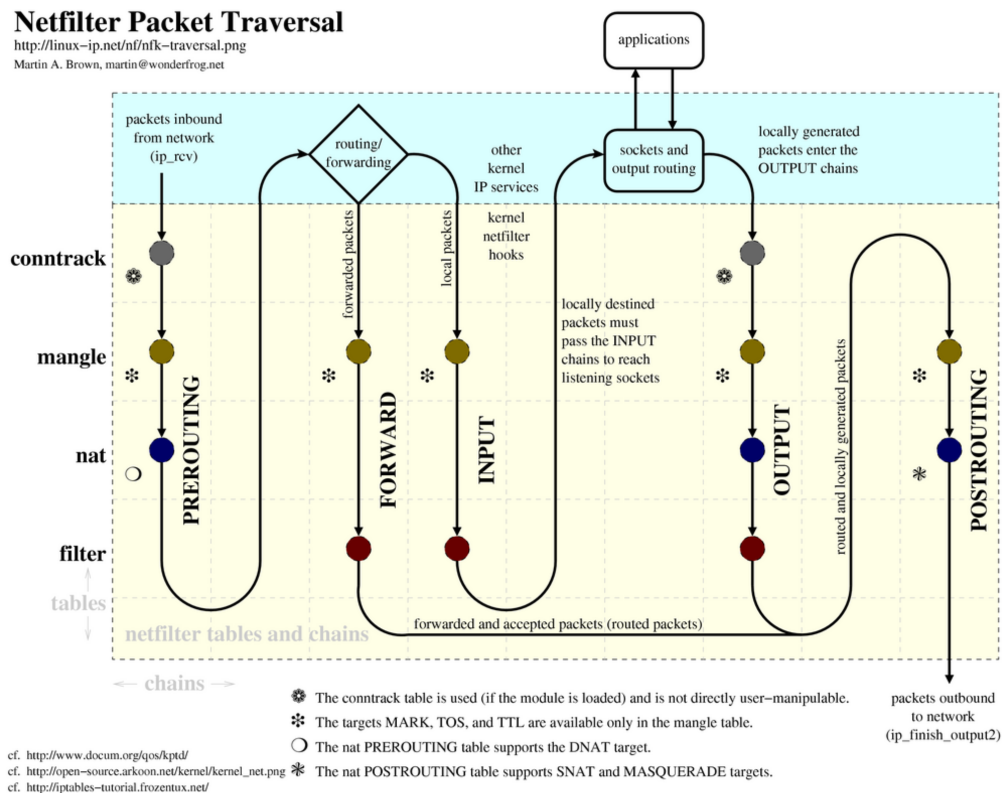
2.2 Ketten (chains)

Jede dieser Tabellen enthält Ketten (*chains*), welche Sequenzen von Verarbeitungsregeln sind. Die Kette bestimmt *wann* ein Paket zu prüfen ist. Im Regelfall werden alle Verarbeitungsregeln innerhalb einer Kette sequentiell bis zum ersten Treffer abgearbeitet.

Die fünf im Kernel integrierten Ketten (*chains*)

- `PREROUTING`: 1. Kette für alle einlaufenden Pakete (*nat* & *mangle*)
- `FORWARD`: 2. Kette für weiterzuleitende Pakete (*filter* & *mangle*)
- `INPUT`: 2. Kette für an lokale Prozesse adressierte Pakete (*filter* & *mangle*)
- `OUTPUT`: 1. Kette für von lokalen Prozessen stammenden Paketen (*filter*, *nat* & *mangle*)
- `POSTROUTING`: Letzte Kette für alle Pakete, die den Rechner verlassen (*nat* & *mangle*)

Einen besonders guten Überblick über die Tabellen und Ketten des Netfilter-Framworks gibt dieses Diagramm.



Angenommen man öffnet seinen Browser und will `www.google.de` einen Besuch abstatten, so werden die dadurch lokal erzeugten Pakete zuerst die `OUTPUT`- und dann die `POSTROUTING`-Ketten der entsprechenden Tabellen passieren. Die Antwort von Google auf unsere Anforderung passiert als erstes die `PREROUTING`- und dann die `INPUT`-Ketten.

Angenommen unser Rechner hängt als Router zwischen zwei verschiedenen Netzen, und es erreichen uns Pakete, die weitervermittelt werden müssen, so ist der Ablauf der folgende:

1. Die Pakete passieren zuerst die `PREROUTING`-Ketten der Tabellen.
2. Jetzt wird eine positive Routingentscheidung getroffen – das Paket ist also nicht für unseren Rechner bestimmt und muss weitergeleitet werden.
3. Das Paket passiert die `FORWARD`-Ketten.
4. Das Paket passiert die `POSTROUTING`-Ketten und verlässt danach unseren Rechner.

2.3 Verarbeitungsregeln

Wie bereits gesagt enthalten die Ketten eine Reihe von Verarbeitungsregeln, die sequentiell abgearbeitet werden.

Jede Verarbeitungsregel besteht aus:

- **Muster** (*match*): Bestimmt auf welche Pakete eine Regel anzuwenden ist.
- **Ziel** (*target*): Bestimmt was mit einem Paket zu tun ist.

Muster (*matches*)

Wenn ein Paket eine Kette betritt, so wird nacheinander das Muster jeder Verarbeitungsregel mit dem Paket verglichen. Passen Muster und Paket nicht zusammen, so wird das Paket mit dem Muster der nächsten Verarbeitungregel verglichen. Passt das Paket dagegen zum Muster, so werden die Anweisungen des Ziels (*target*) dieser Verarbeitungsregel ausgeführt. In der Regel ist damit die Verarbeitung des Pakets in dieser Kette beendet.

Muster können unter anderem sein

- IP-Adresse
- Portnummer
- MAC-Adresse
- Protokoll (z.B. TCP, UDP, ICMP)
- Interface (auf dem ein Paket eintraf bzw. auf dem es verschickt wird)
- ...

Neben den genannten (generischen) Mustern gibt es noch viele auf bestimmte Anwendungsfälle spezialisierte Matches, die über dynamisch ladbare Erweiterungen eingebunden werden können.

Ziele (*targets*)

Ein **Ziel** gibt an, was mit einem Paket passieren soll. Hier kann man mehrere Gruppen von Zielen unterscheiden:

- **benutzerdefinierte Ketten**: Man kann also z.B. in den vorgegebenen Ketten eine grobe Unterscheidung treffen und dann in mehreren hintereinander geschalteten benutzerdefinierten Ketten immer feiner werden.

- **Standardziele**: Es gibt die vier Standardziele `ACCEPT`, `DROP`, `QUEUE` und `RETURN`, die etwas später noch genauer erklärt werden.
- **erweiterte Ziele**: z.B. `LOG` und `REJECT`, und andere, die nur in manchen Ketten bestimmter Tabellen gültig sind

Die vier Standardziele

Es gibt die folgenden Standardziele:

- `ACCEPT`: Das Paket wird akzeptiert und der nächsten Kette zur Bearbeitung übergeben.
- `DROP`: Das Paket wird verworfen (ohne dem Sender eine Rückmeldung zu übergeben).
- `QUEUE`: Das Paket wird der Queue '0' im Userspace übergeben, so dass ein Userspace-Programm das Paket bearbeiten kann. (`ip_queue`-Handler)
  - `NFQUEUE`: Erweiterung von `QUEUE` ab Kernel 2.6.14. Paket kann einer bestimmten Queue im Userspace zugewiesen werden. (`nfnetlink_queue`-Handler)
- `RETURN`: Das Paket gelangt ans Ende der Kette (vgl. `return` in Programmiersprachen als Rückkehr aus einem Unterprogramm).

Bei `RETURN` gelangt das Paket im Gegensatz zu `ACCEPT` nicht zur nächsten Kette, sondern es wird mit der nächsten Regel der vorigen, aufrufenden Kette weitergemacht.

Policies der Standardketten

Den fünf Standardketten `PREROUTING`, `FORWARD`, `INPUT`, `OUTPUT` und `POSTROUTING` kann eine sogenannte **Policy** zugeordnet werden. Diese Policy ist eines der Standardziele. Sie wird angewandt, wenn

- keine Regel der jeweiligen Kette greift, oder
- für das jeweilige Paket das `RETURN`-Ziel angegeben wurde.

Erweiterte Ziele

Zu wichtigsten erweiterten Ziele gehören unter anderem:

- `LOG`: Das Paket wird im Syslog aufgezeichnet und anschließend weiter durch die Kette gereicht. (Kein Abbruch der Kette! *non-terminating target*)
- `REJECT`: Das Paket wird verworfen und der Sender wird darüber mit einem *error packet* informiert. (nur in `INPUT`, `FORWARD` & `OUTPUT` und in von dort aufgerufenen User-Ketten verfügbar)

Zusätzlich gibt es noch erweiterte Ziele, die nur in einzelnen Ketten bestimmter Tabellen und daraus aufgerufenen, benutzerdefinierten Ketten gültig sind. Besonders wichtig sind hier die erweiterten Ziele der *nat*-Tabelle.

Erweiterte Ziele der *nat*-Tabelle

- `DNAT`: steht für Destination NAT, Zieladresse des Pakets wird durch eine angegebene Adresse ersetzt (nur `PREROUTING` & `OUTPUT`)
- `SNAT`: steht für Source NAT, Quelladresse des Pakets wird durch eine angegebene Adresse ersetzt (nur `POSTROUTING`)
- `MASQUERADE`: Spezialform von `SNAT`, Ersetzung der Quelladresse des Pakets durch die Adresse der Schnittstelle, auf dem es den Rechner verlässt (wie `SNAT` nur `POSTROUTING`)
- `REDIRECT`: Leitet das Paket zum lokalen Rechner um, indem die Zieladresse durch die IP des Interfaces, auf dem das Paket eintraf, ersetzt wird (nur `PREROUTING` & `OUTPUT`). Bei lokal erzeugten Paketen wäre die IP dann 127.0.0.1.

Bei den Zielen `DNAT`, `SNAT` und `MASQUERADE` wird die Adressübersetzung gespeichert und auf alle zukünftigen Pakete dieser Verbindung (in beide Richtungen) angewendet. (**connection-tracking**)

3 Netfilter-Konfiguration mit iptables

In diesem Abschnitt kommen wir jetzt endlich zur Praxis und betrachten anhand einiger Beispiele, wie man mit dem Userland-Tool `iptables` Netfilter konfiguriert. Zuerst kümmern wir uns um die Kommandozeilenoptionen von `iptables`.

Der Aufruf von iptables

Im allgemeinen wird `iptables` nach diesem Muster aufgerufen:

```
$ iptables [(-t|--table) table] [weitere Optionen]
```

`table` kann dabei `filter`, `nat` oder `mangle` sein. Wird keine Tabelle angegeben, so wird `filter` als Default angenommen.

3.1 Kommando-Optionen

Regeln hinzufügen

```
$ iptables [(-t|--table) table] \
      (-A|--append) chain rule-spec
      (-I|--insert) chain [rulenum] rule-spec
```

`-A` fügt eine Regel ans Ende der Kette an, `-I` fügt die Regel an der angegebenen Stelle `rulenum` ein (Default: 1, also vorn anfügen).

Regeln löschen

```
$ iptables [(-t|--table) table] \
      (-D|--delete) chain rule-spec
      (-D|--delete) chain rulenum
      (-F|--flush) [chain]
```

`-D` löscht entweder die mit `rule-spec` angegebene Regel oder die Regel mit der Nummer `rulenum`.

`-F` löscht alle Regeln in der gegebenen Kette. Ist keine Kette gegeben, so werden alle Ketten gelöscht.

Sonstige Kommando-Optionen

```
$ iptables [(-t|--table) table] \
      (-L|--list) [chain]
      (-P|--policy) chain target
      (-R|--replace) chain rulenum rule-spec
```

`-L` listet alle Regeln der gegebenen Kette auf, bzw. aller Ketten, wenn keine Kette angegeben wurde.

`-P` setzt die Policy für eine Kette auf das gegebene Ziel.

`-R` ersetzt in der gegebenen Kette die `rulenum`-te Regel durch die angegebene neue Regel.

Kommandos für benutzerdefinierte Ketten

```
$ iptables [(-t|--table) table] \
      (-N|--new-chain) chain
      (-X|--delete-chain) [chain]
      (-E|--rename-chain) old-chain new-chain
```

Mit `-N` und `-E` können neue benutzerdefinierte Ketten erzeugt werden bzw. umbenannt werden.

Mit `-X` wird die angegebene benutzerdefinierte Kette gelöscht. Ist keine Kette explizit angegeben, so werden alle benutzerdefinierten Ketten der jeweiligen Tabelle gelöscht.

3.2 Muster und Ziel

Wenn wir uns an den Aufbau der Verarbeitungsregeln erinnern, so wissen wir, dass sie aus einem Muster (*match*) und einem Ziel (*target*) bestehen.

Aufbau der Regeln

Eine Regel hat folgenden Aufbau:

```
{match}* (-j|--jump) target
```

Man kann also beliebig viele Muster angeben, die untereinander "verundet" sind, gefolgt von genau einem Ziel.

Das Ziel kann wie bereits erwähnt einer benutzerdefinierte Kette, ein Standardziel oder ein erweitertes Ziel sein.

Die Standardmuster

- `(-p|--protocol) [!] protocol`

  Matcht Pakete eines bestimmten Protokolls (tcp, udp, icmp, all, numerischer Wert des Protokolls oder Wert aus /etc/protocols)

- `(-s|--source) [!] address[/mask]`

  Quelladresse

- `(-d|--destination) [!] address[/mask]`

  Zieladresse

- `(-i|--in-interface) [!] name`

  Schnittstelle, über die das Paket ankam

- `(-o|--out-interface) [!] name`

  Schnittstelle, über die das Paket den Rechner verlässt

- `[!] (-f|--fragment)`

  Matcht alle Paketfragmente ab dem 2. Fragment. Bei ! werden jeweils nur die 1. Fragmente (die Köpfe der Pakets) und unfragmentierte Pakete gematcht.

Das Ausrufezeichen negiert ein Muster.

Neben den genannten Standardmuster verfügt `iptables` über sogenannte **Match Extensions**, die entweder implizit über die Option `-p|--protocol` geladen werden oder explizit über `-m|--match`.

Match Extensions

Beispiele:

```
$ iptables -A INPUT -p tcp --syn --dport 23 \
    -m connlimit --connlimit-above 2 -j REJECT
$ iptables -A INPUT -p tcp --syn --dport 23 \
    -m connlimit ! --connlimit-above 2 -j ACCEPT
```

Nur 2 Telnet-Verbindungen pro Client zulassen. Nur Pakete matchen, bei denen das `SYN`-Bit gesetzt ist und die Bits `ACK`, `RST` und `FIN` nicht gesetzt sind. Dies ist beim Verbindungsaufbau der Fall. Beide Regeln sind äquivalent.

```
$ iptables -A INPUT -m ipv4options --ts -j DROP
```

Alle einlaufenden Pakte mit gesetztem Timestamp-Flag verwerfen.

Diese erweiterten Muster bieten noch viele Funktionen für spezielle Anwendungsgebiete, aber allein das Aufzählen würde den Zeitrahmen des Vortrags sprengen.

3.3 weitere Tools

Neben dem eigentlichen Netfilter-Konfigurationstools `iptables` gibt es noch zwei weitere wichtige Helper-Tools.

- `iptables-save [-c] [-t table]`

  Schreibt die Regeln der gegebenen Tabelle (Default: alle Tabellen) inklusive der Paket- und Byte-Counters, falls `-c` angegeben wurde, nach `stdout`. Die Ausgabe kann dann z.B. in eine Datei umgeleitet werden, und man kann dann beim Bootvorgang dir Regeln automatisch wiederherstellen lassen.

  ```
  $ iptables-save > my_rules
  ```

- `iptables-restore [-c] [-n|--noflush]`

  Löscht alle Tabellen und stellt danach die auf `stdin` gegebenen Regeln wieder her. Falls `-c` angegeben wurde, so werden auch die Werte alle Paket- und Byte-Counter wieder hergestellt. Ist die Option `-n` gegeben, so werden die Tabellen vorher nicht gelöscht.

  ```
  $ cat my_rules | iptables-restore
  ```

Tools für IPv6

Das `iptables`-Tool ist auschließlich zum definieren von Regeln für IPv4 zuständig. Für IPv6 gibt es eigene Tools:

- `ip6tables`

  Das IPv6-Äquivalent zu `iptables`. Die Bedienung entspricht der von `iptables`, jedoch ist der Funktionsumfang noch nicht ganz so umfangreich.

- `ip6tables-save`

  Äquivalent zu `iptables-save`

- `ip6tables-restore`

  Äquivalent zu `iptables-restore`

iptables-Frontends

Da die Konfiguration mit `iptables` teilweise recht kompliziert ist, gibt es eine ganze Reihe von graphischen Frontends, mit denen viele Standardszenarien schnell konfiguriert werden können.

- **FireHOL**[2] – iptables firewall generator
- **Firestarter**[3] – GUI for iptables firewall setup and monitor
- **KMyFirewall**[4] – Graphical KDE iptables configuration tool
- **Knetfilter**[5] – Manage Iptables firewalls with this KDE app
- **Shorewall**[6] – Full state iptables firewall

4 Beispiele

4.1 Logging von Paketen

Beispiel 1: Logging (1)

Telnet-Verbindungen sollen abgelehnt werden, wobei die Verbindungsversuche geloggt werden sollen.

```
$ iptables -t filter -A INPUT -p tcp --syn \
           --dport 23 -j LOG \
           --log-prefix 'TELNET:'
```

Das veranlasst `iptables` alle Pakete, die an Port 23 (telnet) gerichtet sind an den System-Logger zu übergeben. Zusätzlich wird den jeweiligen Logeinträgen das Präfix `TELNET:` vorangestellt, damit man die Einträge besser unterscheiden kann. Danach sollen die Pakete verworfen werden. Dies geschieht über:

```
$ iptables -t filter -A INPUT -p tcp --syn \
           --dport 23 -j REJECT \
           --reject-with icmp-host-prohibited
```

Hier wird dem Verbindungsanfordernden noch mitgeteilt, dass dieser Host für Telnet gesperrt ist. Statt `REJECT` hätte man auch `DROP` nutzen können. Dann hätte der Verbindungsanfordernde keine Rückmeldung erhalten.

Beispiel 1: Logging (2)

[2] http://firehol.sourceforge.net/
[3] http://www.fs-security.com
[4] http://kmyfirewall.sourceforge.net/
[5] http://expansa.sns.it/knetfilter/
[6] http://www.shorewall.net/

Auf `linux.uni-koblenz.de` an meinen Laptop mit den obigen Regeln:

```
heimdall@penguin2:~ > telnet 141.26.71.39
Trying 141.26.71.39...
telnet: connect to address 141.26.71.39: \
        Connection refused
```

Danach auf meinem Laptop (141.26.71.39):

```
(#:~)- dmesg
TELNET:IN=eth0 OUT= \
  MAC=00:06:5b:b9:9b:55:00:02:b3:8d:c5:ef:08:00
  SRC=141.26.64.104 DST=141.26.71.39 LEN=60 \
  TOS=0x10 PREC=0x00 TTL=64 ID=6091 DF PROTO=TCP \
  SPT=59925 DPT=23 WINDOW=5840 RES=0x00 SYN URGP=0
```

4.2 NAT im Heimnetz

In diesem Abschnitt soll gezeigt werden, wie man den Linux-Router im lokalen Heimnetz konfigurieren muss, damit alle Rechner ins Internet können. Der Router nimmt dabei Network Address Translation vor.

Das Szenario ist wie folgt: Ein Linux-Router ist direkt per Einwahlverbindung mit dem Internet verbunden. Er bezieht also seine IP bei einem Provider wie 1&1, T-Online oder Arcor. Alle anderen Rechner im lokalen Netz haben nur ihre lokalen IPs aus dem privaten Adressbereich. Will man nun von einem dieser Rechner im WWW browsen, beispielsweise die Homepage der Uni besuchen, so kommt das Anfragepaket zuerst beim NAT-Router an. Dieser tauscht die Quelladresse, also die lokale IP des Rechners, gegen seine eigene, vom Provider zugewiesene Adresse aus. Trifft nun das Response-Paket vom Uni-Webserver beim NAT-Router ein, so tauscht er die Zieladresse, also die vom Provider zugewiesene IP, wieder gegen die lokale IP des anfragenden Rechners aus und leitet das Paket an diesen weiter.

Von außen sieht es also so aus, dass nur ein einziger Rechner – der NAT-Router – mit dem Internet kommuniziert. Er dient den anderen lokalen Rechnern als Proxy.

Man unterscheidet wie bereits gesagt zwei Arten von NAT:

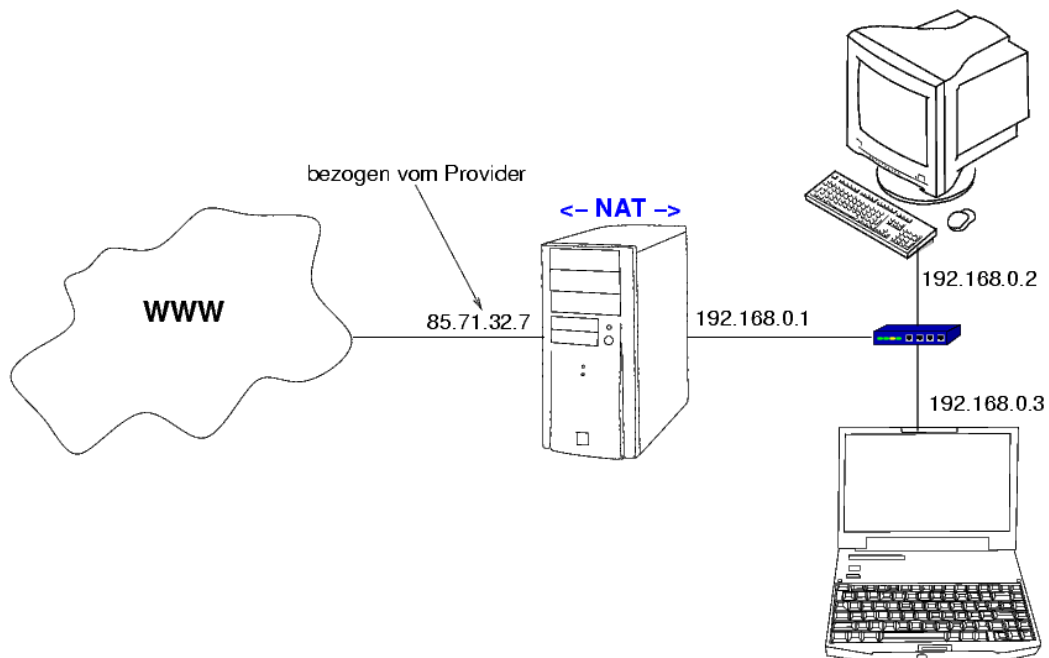
- Source NAT (Ziel: `SNAT`): Die Quelladresse des ersten Pakets wird verändert. Diese NAT-Form wird immer in der Kette `POSTROUTING` angewendet. Eine spezielle Form des Source NAT ist *Masquerading* (Ziel: `MASQUERADE`). Dabei wird die Quelladresse automatisch durch die Adresse des Interfaces, auf dem das Paket den NAT-Router verlässt, vertauscht.

Bei MASQUERADE werden zudem alle Verbindungen und Mappings "vergessen", sobald das Interface abgeschaltet wird. Das ist genau das Verhalten, welches man in Dialup-Umgebungen haben will, denn es ist unwahrscheinlich, dass man bei einer neuen Einwahl die gleiche IP-Adresse zugewiesen bekommt, womit dann alle Übersetzungsregeln falsch wären.

- Destination NAT (Ziel: DNAT): Die Zieladresse des ersten Pakets wird sofort beim Eintreffen (PREROUTING) oder in der OUTPUT-Kette bei lokal erzeugten Paketen verändert. Das sogenannte Port-Forwarding ist beispielsweise eine Form von DNAT.

Ebenfalls wurde bereits erwähnt, dass die Adressübersetzungen nur auf das erste Paket einer Verbindung explizit angewendet werden müssen. Die korrekte Adressübersetzung wird dann gespeichert und auf alle folgenden Pakete dieser Verbindung automatisch angewandt.

Beispiel 2: NAT im Heimnetz (1)



Es soll nun erreicht werden, dass die beiden Rechner im Heimnetz über den NAT-Router ins Internet gelangen können.

Beispiel 2: NAT im Heimnetz (2)

- Match: alle ausgehenden Pakete (`-o <iface>`)
- Target: `MASQUERADE`

```
$ iptables -t nat -A POSTROUTING \
           -o ppp0 -j MASQUERADE
$ echo 1 > /proc/sys/net/ipv4/ip_forward
```

Damit wird der `POSTROUTING`-Kette der `nat`-Tabelle eine neue Regel hinzugefügt, die bewirkt, dass in allen ausgehenden Paketen die Quelladresse mit der Adresse des ausgehenden Interfaces (hier `ppp0`) vertauscht wird. Bei Anwortpaketen wird die Adressübersetzung automatisch in umgekehrter Reihenfolge vorgenommen.

4.3 Port-Forwarding / Load-Sharing

Beispiel 3: Port-Forwarding / Load-Sharing (1)

Ein Server nimmt alle Anfragen von außen entgegen und je nach angefragten Dienst werden die Pakete aufgefächert und an andere, dahinterliegende Rechner weitergeleitet.

Bsp.:

- Alle Pakete gehen auf `master.foo.org` ein.
  - Alle ssh-Anfragen werden an `ssh.foo.org` (IP: 192.168.0.2) weitergeleitet.
  - Alle http-Anfragen werden an `www.foo.org` (IP: 192.168.0.3) weitergeleitet. Dieser Webserver lauscht auf Port 8080.
  - Alle ftp-Anfragen werden an `ftp.foo.org` (IP: 192.168.0.4) weitergeleitet.
  - Andere Anfragen werden verworfen.

Obiges Szenario lässt sich leicht mit `DNAT` auf `master.foo.org` einrichten.

Beispiel 3: Port-Forwarding / Load-Sharing (2)

```
# ssh -> 192.168.0.2
$ iptables -t nat -A PREROUTING -p tcp \
  --dport 22 -j DNAT \
  --to-destination 192.168.0.2
```

```
# http -> 192.168.0.3:8080
$ iptables -t nat -A PREROUTING -p tcp \
  --dport 80 -j DNAT \
  --to-destination 192.168.0.3:8080
```

Beispiel 3: Port-Forwarding / Load-Sharing (3)

```
# ftp -> 192.168.0.4
$ iptables -t nat -A PREROUTING -p tcp \
  --dport 21 -j DNAT \
  --to-destination 192.168.0.4

# Alle anderen Pakete verwerfen, indem die Policy
# für die PREROUTING-Kette der nat-Tabelle auf
# DROP gesetzt wird.
$ iptables -t nat --policy PREROUTING DROP
```

4.4 BitTorrent

Beispiel 4: BitTorrent weiterleiten (1)

Das Szenario:

- Ein Linux-Router hängt als NAT-Router zwischen dem lokalen Netz und dem Internet.
- BitTorrent benutzt standardmäßig TCP und UDP auf Port 6881.
- Dieser Port soll an den Rechner mit der IP 10.0.0.2 im lokalen Netz weitergeleitet werden.

Beispiel 4: BitTorrent weiterleiten (2)

```
$ iptables -t nat -A PREROUTING -p tcp -i ppp0 \
    --dport 6881 -j DNAT \
    --to-destination 10.0.0.2:6881

$ iptables -t nat -A PREROUTING -p udp -i ppp0 \
    --dport 6881 -j DNAT \
    --to-destination 10.0.0.2:6881
```

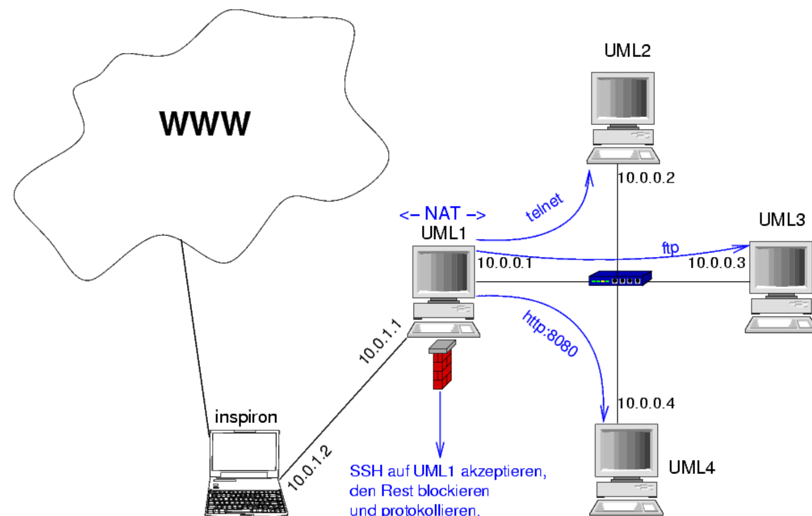
```
$ iptables -A FORWARD -p tcp -i ppp0 -d 10.0.0.2 \
    --dport 6881 -j ACCEPT

$ iptables -A FORWARD -p udp -i ppp0 -d 10.0.0.2 \
    --dport 6881 -j ACCEPT
```

Bei den ersten beiden `iptables`-Aufrufen kann man den Port bei `--to-destination` weglassen, denn standardmäßig leitet `iptables` auf dem Port weiter, auf dem empfangen wurde (hier also 6881).

Außerdem können die beiden letzten Aufrufe weggelassen werden, wenn sonst keine anderen Filterregeln in der `filter`-Tabelle stehen. Standardmäßig werden nämlich alle Pakete akzeptiert.

5 Das VNUML-Beispiel



5.1 Konfiguration der UML-Guests

Alle UML-Guests sind gleich konfiguriert. Nach dem Booten sind keine Filterregeln definiert, IP-Forwarding ist überall aktiviert. Auf allen UML-Guests läuft ein http-server (mini_httpd[7]), ein FTP-Server (Netkit FTP server[8]) und ein Telnet-

[7] `http://www.acme.com/software/mini_httpd/`
[8] `http://www.hcs.harvard.edu/~dholland/computers/netkit.html`

Dämon (Port des OpenBSD-telnetd[9]). Natürlich läuft zusätzlich noch der SSH-Dämon von OpenSSH[10].

5.2 Was soll erreicht werden?

1. Der Rechner UML1 soll als NAT-Router dienen.
2. Wenn telnet-Anfragen von außen (also über den Host) eintreffen, so sollen diese an UML2 weitergeleitet werden.
3. Wenn ftp-Anfragen von außen kommen, so sollen diese an UML3 weitergeleitet werden.
4. Wenn http-Anfragen von außen auf Port 80 kommen, so sollen diese an UML4, allerdings auf Port 8080, weitergeleitet werden.
5. Alle Nicht-SSH-Anfragen, die direkt an UML1 gerichtet sind, sollen protokolliert und dann mit Rückantwort an den Anfragenden geblockt werden.

5.3 Die Konfiguration, die zum Ziel führt

[9] ftp://ftp.suse.com/pub/people/kukuk/ipv6/
[10] http://www.openssh.com/

```
### Auf dem Host ############################
#######
# Forwarding aktivieren

(#:~)- echo 1 > /proc/sys/net/ipv4/ip_forward

#######
# neue Route für das Netz 10.0.0.0/8 über
# GW 10.0.1.1

(#:~)- route add -net 10.0.0.0 netmask \
        255.255.255.0 gw 10.0.1.1

### Auf UML1 ################################
#######
# Forwarding aktivieren

$ echo 1 > /proc/sys/net/ipv4/ip_forward

#######
# Default-GW ist der Host (ist normal
# schon da).

$ route add default gw 10.0.1.2

#######
# TODO: NAT einrichten:

$ iptables -t nat -A POSTROUTING -j SNAT \
    --to-source 10.0.1.1

# TESTEN: Auf UML{2,3,4} ein 'ssh bla.foo.org'
# machen und auf dem Host 'tcpdump -i Net1'.
# Dann sieht man, dass die Quelladresse
# 10.0.1.1 (Net1-Iface von UML1) ist und eben
# nicht 10.0.0.{2,3,4}.

#######
# TODO: Telnet an UML2 weiterleiten:

$ iptables -t nat -A PREROUTING -p tcp --dport 23 \
```

```
    -j DNAT --to-destination 10.0.0.2

# TESTEN: Vom Host aus ‘telnet 10.0.1.1’, dann
# anmelden und ‘ifconfig’, und siehe da, wir sind
# auf UML2 (10.0.0.2). Der Prompt zeigt natürlich
# auch schon "uml2".

#######
# TODO: FTP an UML3 weiterleiten:

$ iptables -t nat -A PREROUTING -p tcp --dport 21 \
    -j DNAT --to-destination 10.0.0.3

# TESTEN: Vom Host aus ‘lftp 10.0.1.1’, dann als
# root anmelden mit ‘login root’. Um zu sehen auf
# welchem host wir sind einfach die Datei
# /proc/sys/kernel/hostname anzeigen oder downloaden
# (get hostname) und lokal anzeigen.

#######
# TODO: HTTP an UML4 weiterleiten:

$ iptables -t nat -A PREROUTING -p tcp --dport 80 \
    -j DNAT --to-destination 10.0.0.4:8080

# TESTEN: Mit Brower vom Host auf 10.0.1.1 browsen.

#######
# TODO: Alles außer SSH protokollieren und blocken.

$ iptables -t filter -A INPUT -p tcp -m tcp \
    --dport ! 22 -j LOG --log-prefix "NOT SSH: "

$ iptables -t filter -A INPUT -p tcp -m tcp \
    --dport ! 22 -j REJECT \
    --reject-with icmp-host-prohibited

$ iptables -t filter -P INPUT ACCEPT

# TESTEN: Auf Host ein ‘tcpdump -i Net1’. In einem
# anderen Terminal auf dem Host
```

```
# ‘svn co svn://10.0.1.1’. Auf UML1 werden nun die
# Anfragen mit "NOT SSH" geloggt (dmesg), und der
# tcpdump auf dem Host zeigt auch "ICMP host 10.0.1.1
# unreachable - admin prohibited" als Antwort des
# REJECTs an.

######################################################
# Damit ist unser Beispielszenario komplett.         #
######################################################

#######
# TODO: Bsp. Packet-Mangling: Alle auf dem Host
# erzeugten Pakete sollen die Time To Live 17 bekommen.

host$ iptables -t mangle -A OUTPUT -j TTL --ttl-set 17

# TESTEN: Auf z.B. 10.0.0.2 ein ‘tcpdump -i eth1 -vvv’
# machen, und dann vom Host ein ‘ping 10.0.0.2’. Dann
# sieht man bei dem tcpdump, dass die TTL jetzt 16
# = 17 - 1 ist.
```

6 Literatur

- Geschichtliches: `http://de.wikipedia.org`
- Aufbau und Anwendung: `http://www.netfilter.org` & `man 8 iptables`