Firewall and Shaping on Broadband SoHo Routers using Linux

An introduction to iptables, iproute2 and tc

Sebastian ’blackwing’ Werner, Erlangen

`blackwing at erlangen dot ccc dot de`

CCC Erlangen

Aims of this Talk

- Showing modern capabilities of Linux 2.4 / 2.6 series kernels
- Giving some solution snippets
- Introduction to `ip`, `tc` and `iptables`

Questions are welcome at *ANY* time!

About iptables

- Since Linux 2.3 branch standard firewalling suite
- First maintained by Paul Russel, now by Harald Welte
- Modular design, easy to extend
- Very large number of matching methods
- Secures System from Network (System Firewall)
- Manages Network Firewalling, Network Address Translation (NAT) and Mangling of Packets

Advantages of iproute2

- `route` is very limited: Only one routing table
- `ifconfig` has only basic features
- `ip` offers bleeding-edge features:
  - Policy based routing
  - Interface address overloading
  - Routing scopes
  - ...
- Uses netlink interface instead of ioctl
- One tool for all network related sysadm

iproute2: tc

The most interesting part of iproute2 is probably `tc`
`tc` is your solution for any QoS related issue.

It offers the only way to manage the queueing discipline of a recent kernel.

Basic concepts of firewalls

- **opt-in filtering**
  deny everything, allow some
- **opt-out filtering**
  deny some, allow everything else
- **dmz public services**
  in separated net
- **nat users get rfc1918**
  space instead of public ips

Time for practics!

So, lets take a look at `iptables`!

Basic iptables usage

- Accept packets that come in via eth0
  ```
  iptables -A INPUT -i eth0 -j ACCEPT
  ```
- Accept tcp packets that come in on eth0 and get routed to eth1
  ```
  iptables -A FORWARD -i eth0 -o eth1 -p
  tcp -j ACCEPT
  ```
- Reject outgoing icmp packets
  ```
  iptables -A OUTPUT -p icmp -j REJECT
  ```

So what the f*** are chains?

- List of rules that are sequentially checked
- Can be user defined or built-in
- Belong to the tables `filter`, `nat` or `mangle`

Ehm... I’m confused! But what are these tables?

Different tables

- **filter**
  Just plain firewalling. Contains the basic chains `INPUT`,`FORWARD` and `OUTPUT`
- **nat**
  Handles NetworkAdressTranslation, Redirects, SNAT and DNAT. Contains `PREROUTING`, `POSTROUTING` and `OUTPUT`
- **mangle**
  Manipulating packet headers. Provides `INPUT`, `OUTPUT`, `FORWARD`, `POSTROUTING`, `PREROUTING` chains.

Targets

- `ACCEPT` Packet is accepted. No further rules are checked.
- `DROP` Packet is dropped.
- `REJECT` Packet is reject, but a message is sent to origin.
- `LOG` Log packet via klog/syslog. All further rules are checked.
- `ULOG` Log packet via netlink socket to userspace.
- `MASQUERADE` NAT this packet.
- `SNAT` Change *source* of packet.
- `DNAT` Change *destination* of packet.
- `REDIRECT` Redirect packet locally.

Advanched Targets

- `IMQ` Pass interface to IMQ device.
- `TCPMSS` Modificate TCP maximum segment size.
- `ECN` Perform explicit congestion notifcation stuff.
- `MARK` Assign a flowid to this packet
- `CONNMARK` Same, but for a connection (connection tracking!)

Matching extensions

`iptables` is very modular. And there are many modules!
`iptables -m` *module-name* `--module-param` *module-arg*

- `state`
  provides stateful firewalling via the connection tracking facility.
- `mark`
  match by marks applied via the `MARK` target
- `tos`
  match by type-of-service field
- `length`
  match by packet length
- `ipp2p`
  match peer2peer traffic (live in a flat-share? you will LOVE this one!)

Stateful firewalling

As mentioned before, there is the connection tracking table, which enables iptables to filter per connection state.
States are

- `NEW`
  A packet that initiates a new connection such as TCP-syn.
- `ESTABLISHED`
  A packet that belongs to a valid connection
- `RELATED`
  Related traffic to a valid connection. e.g. ICMP-error or a passive ftp connection.
- `INVALID`
  A apacket that cannot be identified.

What you should know about NAT

- Simple NAT rewrites the source address in the IP header.
- Some protocols (like ftp) have IP in tcp payload - So they have to be altered, too!
- Lots of proprietiary application protocols are not NAT safe.
- Keeping a connection table needs lots of cpu time and memory.
- Because connections are mapped, the source tcp-port has to be altered too. But the host has only 65000!

Questions?!?!

Got any Questions?

Now, lets get into a real world situation....

Recomended basic settings

- `iptables -P INPUT DROP`
  Drop all incoming packets per default
- `iptables -P FORWARD DROP`
  Drop all forwarding packets per default
- `iptables -A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT`
  Accept established stuff to local machine
- `iptables -A FORWARD -m state --state RELATED,ESTABLISHED -j ACCEPT`
  Accept running connections through the gate
- *... your rules here*

But what about...

So, whats the magic point in the last slide?
Right! All *OUT*GOING traffic is accepted!

This might be a security flaw, cause *any* user could open a socket and connect anything. But regularly this should be no prob.

Enabling local services

- `iptables -A INPUT -m state --state NEW --protocol tcp --destination-port ssh -j ACCEPT`
  Accept incoming ssh
- `iptables -A INPUT -m state --state NEW -p icmp --icmp-type echo-request -j ACCEPT`
  ICMP pings
- `iptables -A INPUT --incoming-interface eth0 -m state --state NEW -p tcp --destination-port 135:139 -j ACCEPT`
  Accept samba stuff

More specific matching

-
  ```
  iptables -A OUTPUT -i ppp0 -p tcp -m
  ipp2p --bit -j REJECT --reject-with
  tcp-reset
  ```
  Deny BitTorrent traffic

-
  ```
  iptables -A INPUT -m limit --limit
  5/minute -m state --state NEW -p tcp
  --destination-port https -j ACCEPT
  ```
  Allow 5 new https connects per minute

-
  ```
  iptables -A INPUT --in-interface eth0 -m
  mac --mac-source 00:0c:8e:13:37:df
  --state NEW -j ACCEPT
  ```
  Allow traffic from mac 00:0c:8e:13:37:df

Pre- and Postrouting stuff

These specific chains apply before or after the kernel routing decision, so this enables a very nice possibility for nice rewrites...

-
  ```
  iptables -t nat -A PREROUTING -p tcp -i
  eth0 --destination-port 80 -j REDIRECT
  --to-port 3128
  ```
  Transparent proxy!

-
  ```
  iptables -t nat -A POSTROUTING -o ppp0 -j
  MASQUERADE
  ```
  NAT outgoing ppp0 traffic

DNAT Stuff

Once you have NAT, you might have a server behind the gate that provides some service... Here ’s the solution...

-
  ```
  iptables -t nat -A PREROUTING -p tcp -i
  ppp0 --destination-port 80 -j DNAT --to
  192.168.99.200:80
  ```
  Forward outside http requests to internal host

-
  ```
  iptables -t nat -A PREROUTING -p tcp -i
  ppp0 --destination-port 80 -j DNAT --to
  192.168.99.1-192.168.99.3
  ```
  Or even better... Loadbalance!

Questions!?

Any Questions?!

... so lets get to TrafficShaping!

Concepts of traffic shaping

Whenever bandwidth is limited, you might want to introduce Quality of Service to ensure that some data is delivered first-class and other just in economy style...
This is in nature of internet: Some data needs to be interactive (ssh, telnet) others is ok, when its delivered bulk-style (ftp, p2p).
So that's where traffic shaping starts:

- When data is sent, it is intermediately buffered
- This buffer is sorted by certain rules

Of classes and qdiscs

- Every interface has a default *root* queue discipline (*qdisc*) and you can simple change this default.
- But then, ALL data would be queued he same way - And so, there would be no difference.
- This is the point where *classes* arise: A tree-like structure is created!
- Then you need to mark the traffic: Here you can use `ip rule` or `iptables`.
- Finally you need to sort the marked traffic into its designated clas/qdisc: `tc filter`

Layout of a shaping solution

- Attach a qdisc to the root-handler (1:0)
- Attach a classifier to this class (1:1)
- This child classifier might address three classes (1:11, 1:12 and 1:13).
- Now you add a handler to this class, to run a queueing discipline on it
- Finally you add a filter for every class.

Simple Classless Queues

- `pfifo_fast`
  As the name says: FirstInFirstOut, that's the default for every interface
- `sfq`
  Stochastic Fairness Queueing - Tries to ensure fair bandwidth allocations.
- `tbf`
  Token Bucket Filter - Allows packets to pass, if they match the rate. Some burst is covered.
- `red`
  Random Early (Detection—Drop) - Randomly drops packets at maximum rate to trigger tcp bandwidth control
- `wrr`
  Weigthed Round Robin - Round robin based on source IPs

Classful Queueing Discipline

- `prio`
  Sort to subclassed based on TypeOfService bit. (Creates 1:1, 1:2, 1:3)
- `cbq`
  Class Based Queueing - Ensures a rate by calculating idle times (Very complex but extremly powerful!)
- `htb`
  Hierarchical Token Bucket - Token based approach to ensure a bandwidth

Questions?!?!

Got any Questions?

Then lets see, how its done.

Outbound shaping example

- Add htb to root qdisc, default is class 50

  ```
  tc qdisc add dev ppp0 root handle 1:  htb
  default 50
  ```

- Set class rate to 510kbit

  ```
  tc class add dev ppp0 parent 1:  classid
  1:1 htb rate 510kbit
  ```

Shaping example (II) - Classes

- Construct a high prio qdisc with 100kbit

  ```
  tc class add dev ppp0 parent 1:1 classid
  1:10 htb rate 100kbit ceil 450kbit burst
  2k quantum 1500 prio 0
  ```

- Construct a mdeium prio qdisc with 400kbit

  ```
  tc class add dev ppp0 parent 1:1 classid
  1:20 htb rate 400kbit ceil 450kbit burst
  2k quantum 1500 prio 0
  ```

- Construct default class

  ```
  tc class add dev ppp0 parent 1:1 classid
  1:50 htb rate 100kbit ceil 450kbit burst
  2k quantum 1500 prio 0
  ```

Shaping example (III) - Qdiscs

- Assign pfifo for highest class

  ```
  tc qdisc add dev ppp0 parent 1:10 handle 10: pfifo
  ```

- Assign enhanced sfq to the second class

  ```
  tc qdisc add dev ppp0 parent 1:20 handle 20: esfq hash src limit 16 perturb 5
  ```

- Assign esfq to the dafult. But with 10 second hash-time.

  ```
  tc qdisc add dev ppp0 parent 1:50 handle 50: esfq hash src limit 16 perturb 10
  ```

Shaping example (IV) - Filtering

- Filter packets with mark 10 to class 10

  ```
  tc filter add dev ppp0 parent 1:0 prio 0
  protocol ip handle 10 fw flowid 1:10
  ```

- These with 20 to class 20

  ```
  tc filter add dev ppp0 parent 1:0 prio 0
  protocol ip handle 20 fw flowid 1:20
  ```

- Redundancy: all others to 50

  ```
  tc filter add dev ppp0 parent 1:0 prio 0
  protocol ip handle 50 fw flowid 1:50
  ```

Shaping example (V) - Marking

You probably asked yourself how you get those *flowids* onto those packets...

-
  ```
  iptables -t mangle -A POSTROUTING -o ppp0 -p icmp -j MARK --set-mark 10
  ```
  Mark icmp packets to 10

-
  ```
  iptables -t mangle -A POSTROUTING -o ppp0 -m tos --tos Minimize-Delay --j MARK --set-mark 10
  ```
  Mark ToS Min-Delay with 10

-
  ```
  iptables -t mangle -A POSTROUTING -o ppp0 -p tcp --destination-port http -j MARK --set-mark 20
  ```
  Mark http stuff with 20

Improving performance

As you might see: Marking every packet in a flow takes a lot of time...
So: Why don't "abuse" connection tracking for storing our marks?!
After marking packets, just save those:

```
iptables -t mangle -A POSTROUTING -o ppp0 -p
tcp -j CONNMARK --save-mark
```

But remember: Before marking packets, restore old marks!

```
iptables -t mangle -A POSTROUTING -o ppp0 -p
tcp -j CONNMARK --restore-mark

iptables -t mangle -A POSTROUTING -o ppp0 -p

tcp -m mark !  --mark 0 -j ACCEPT
```

Questions?!?!

Got any Questions?

This was outbound... Now lets see how inbound is done!

Inbound shaping

Right now, we just saw outbound shaping - cause its pretty easy to just rearrange the sending buffer.

But shaping inbound is way more complex...

One solution is the *Intermediate Queueing Device* which is just a pseudo interface with a queue that gets all designated inbound traffic.

Another solution in the native *ingress* Interface Queue, but this one is pretty limited.

But I'll show you an example for both.

The Intermediate Queueing Device

- First: Get a Kernel patch - Be aware: IMQ is unmaintained!
- Initialised via `modprobe imq numdevs=1`
- Construct classifier as described above: `tc qdisc add dev imq0 handle 1: root htb default 50` and so on.
- Add matching rules to prerouting chain. e.g. `iptables -t mangle -A PREROUTING -i ppp0 -p tcp --source-port http -j MARK --set-mark 20`
- Redirect traffic to IMQ device: `iptables -t mangle -A PREROUTING -i ppp0 -j IMQ`

Conclusion

- **Firewalling**
  `iptables` with its rich pool of match-methods is suitable for any issue in SoHo.
- **Upstream Shaping**
  Egress shaping with htb and sfq is pretty useful and powerful
- **Downstream Shaping**
  Ingress shaping has some very bad limitations. IMQ, the (right now) better solution is unmaintained and a dirty workaround, but running stable!
- **Performance**
  A regular Pentium-3 class server is just idle by shaping a QSC 1024/512 link.

Thank you for your patience!

So, that's it!
I want to say *Thank You* to:

- Prof. Donald E. Knuth for introducing LaTeX
- Frederic Goualard for providing prosper (which was used to write those slides)
- Paul 'Rusty' Rusty for developing iptables and Harald Welte to improve it
- Alexey Kuznetsov for iproute2 and the cbq solution
- RRZE of University Erlangen-Nuremberg for providing various testing equipment