A Scalable Packet Filtering Concept

Markus Schade Daniel Schreiber

January 31, 2006

Abstract

The aim of this work is to show and document a different approach for packet filtering in medium scale setups. We will demonstrate a concept which uses the iptables MARK target, introduced as one of the new targets since the first release of iptables, to assign traffic to individual users and how to group them into service levels with a common rule set. In the second part we will combine the rule set with Linux Quality of Service capabilities.

Introduction

Many people have written articles [6][7] and tutorials[11][5], performed benchmarks[10] and covered the formerly new subject extensively[3]. Nevertheless most articles where concerned with home user setups[9], not enterprise sized rule sets. And very few publications[8] presented a feasible and complete concept how to build a scalable rule set arguably because iptables has a linear packet classification algorithm. Nevertheless iptables is still a very mature and good packet filter that can deliver good performance even today. A performance comparison[10] between nf-hipac[4], ipset[1] and iptables showed that as long as the amount of iptables rules is limited (less than 512 rules), iptables can deliver equal speed. Performance drops below 50% compared to nf-hipac and ipset with more than 2000 rules. A scalable firewall concept must address this issue.

Since there are many good introductory articles[5][11] to the iptables basics, we will cover only a very small part. After a short revision, we will introduce a firewall concept which could theoretically scale up to over 65000 hosts, and group firewalled hosts into over 4000 different categories. With the ability to discern between users, we can also implement bandwidth limitation based on the traffic volume generated by each one. The challenge however is to keep the amount of rules small, which effectively limits the total number of rules to less than 2000, or even less than 1000 depending on performance requirements.

The Basics

The filter Table

Upon entering a network interface, each packet traverses a number of tables and potentially several chains in each table. If it is destined for the host itself, it will enter the `INPUT` chain of the *filter* table and, if permitted, will be handled by a local process assuming, that there is one listening. Any local generated packet will be sent through the *filter* `OUTPUT` chain and leave through a network device if no filtering is performed.

For a gateway, whose purpose is to control the traffic going in and out of a network, these chains are of course secondary. So we will focus on the chain, which is carrying out the packet filter part of our firewalling strategy: the `FORWARD` chain. Since the early days (1993) Linux has been capable of forwarding IP packets [1]. To enable IP forwarding, we have to tell the kernel to do so:

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

It is generally a good idea to enable forwarding **after** we have set up rules in the `FORWARD` chain or changed the chain policy to DROP.

The mangle Table

Linux would not be Linux, if that was all there is to it. Netfilter provides more means to manipulate packets. The built-in *mangle* table can be used in many ways to control and even alter packets passing through our packet filter. For the time being, we will pick the `FORWARD` chain of the *mangle* table, which is not only different from the `FORWARD` chain of the *filter* table, but also gets examined before. Later on we will discuss how to setup the assignments to our quality of service policy in the `POSTROUTING` chain of the *mangle* table. For a summary of the complete chain traversal refer to figure 1.

The issue of Network Address Translation or NAT is left out in our discussion to reduce complexity. Nevertheless it should pose no problem to adopt the proposed setup to a NAT situation.

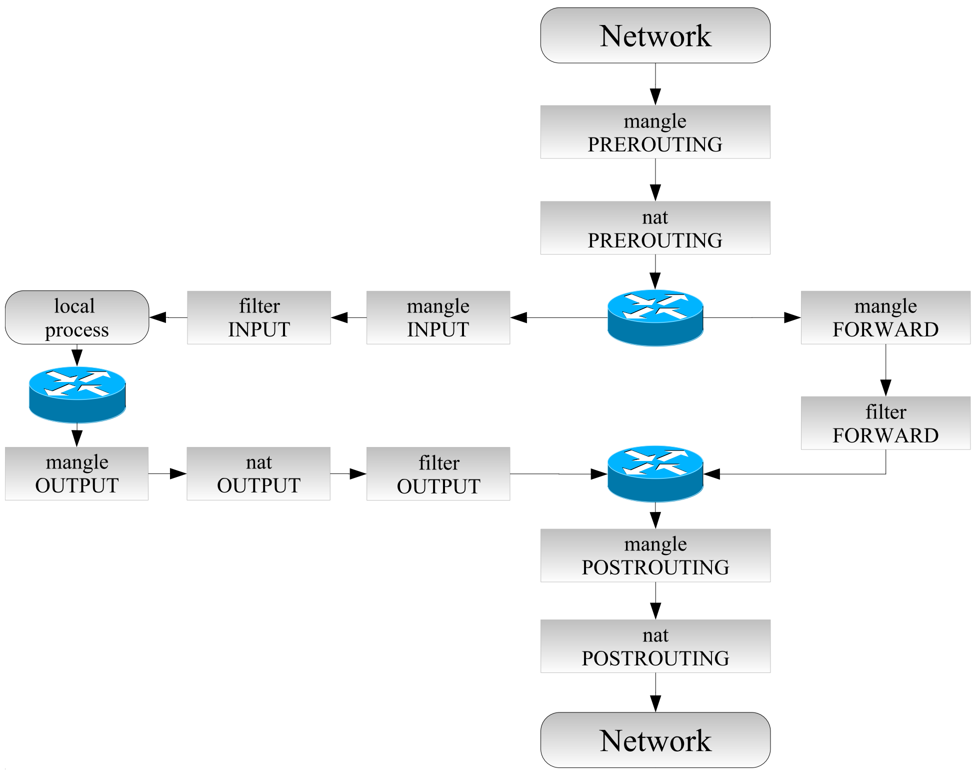

Figure 1: netfilter chain traversal

[1] `net/inet/route.c` is included since Kernel 0.99.10 which was released in 1993

Network

The task for our filter setup will be a rather common one. We have a site with a large number of hosts on a branch of a campus network. The hosts are separated into different subnets and share a common uplink to the campus backbone. Our box guards this gateway and controls in- and outgoing traffic. For the sake of simplicity, we assume that the internet begins just beyond the interface of our Linux gateway (Figure 2). Any additional requirements such as a DMZ are easily taken care of.

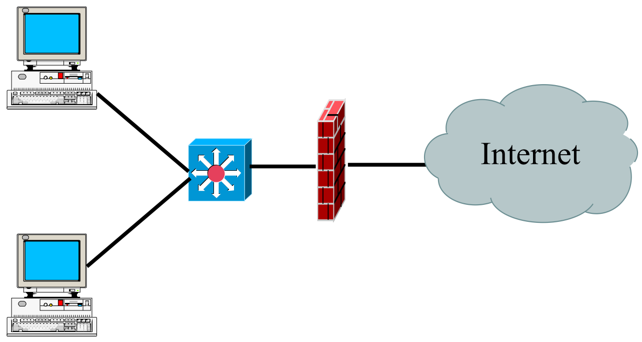

Figure 2: Network topology

Connection Tracking

With the introduction of iptables in Linux 2.4, the ip_conntrack module, which implements connections tracking, finally provided the ability to do stateful filtering. A common setup will contain the following rules:

```
1 iptables -P FORWARD -j DROP
2 iptables -N Forward_New
3 iptables -A FORWARD -m state --state ESTABLISHED,RELATED -j ACCEPT
4 iptables -A FORWARD -m state --state NEW -j Forward_New
```

This first rule sets the policy of the `FORWARD` chain to DROP. The second one creates a new chain called `Forward_New`. In rule 3 we allow all packets, belonging to already established or related connections, to pass. The last one finally jumps to the chain, created in rule 2, for all packets classified as NEW by the ip_conntrack module. This works for both TCP and UDP packets, even though UDP is stateless.

These rules are entirely sufficient to permit existing connections to continue. However, no new connections will be able to pass through, so we have to fill the `Forward_New` chain with some meaningful rules. We will try to give some examples, what can be done. First we will perform some general filtering:

```
iptables -N Forward_All
iptables -A Forward_All -p icmp \
    --icmp-type echo-request -m limit --limit 10/s -j ACCEPT
iptables -A Forward_All -p icmp --icmp-type echo-request -j DROP
```

If desired, rules can be added to prevent private IP addresses from leaving the network or to drop connections to or from well-known worm ports.

```
iptables -A Forward_All -s 10.0.0.0/8 -j DROP
iptables -A Forward_All -s 192.168.0.0/16 -j DROP
...

iptables -N worm_hole
iptables -A Forward_All -p tcp --dport 135 -j worm_hole
...
iptables -A worm_hole -j DROP
```

We could add more specific (not user specific) rules for each way in and out of the network, like disallowing externally initiated connections. In our case, site policy requires, that no new connections to user hosts can be established (with the exception of ICMP echo requests, which we have just limited to 10 packets per minute). And finally we have to insert the proper jump rules to those generic chains.

```
iptables -N Forward_World2Home
iptables -A Forward_World2Home -j DROP
...

iptables -A Forward_New -j Forward_All
iptables -A Forward_New -i eth1 -s ! 134.109.0.0/16 -j Forward_World2Home
```

As you may have noticed, we have used DROP instead of REJECT. Sending a proper REJECT is not only RFC-compliant, but also considered to be more polite. There is a lot of discussion about this matter. From our point of view, it is not a question of RFC-compliance, but of self-defense. With the spread of precompiled cracker tools any malicious person can commandeer a botnet of zombie hosts with only limited knowledge. A botnet is able to generate so many faked packets, that the replies, generated by a proper REJECT rule, will either saturated the uplink (without limit rule) and/or end up at the wrong people. If the internal network is secured by other means, such as port security, static ARP and layer 3 ACL's (preventing any tampering with source MAC and IP address), one could afford the politeness of rejecting packets from internal IP's properly. But when dealing with packets coming from external sources such niceness can be an expensive choice.

The last rule in the FORWARD chain of the *filter* table, for now, will be a rule that jumps to a chain, which would normally contain the rules for every host given permission to access the outside world. But we will use a different approach, so we will name this chain `Forward_Classes`, as it will hold rules matching a whole group of users.

```
iptables -N Forward_Classes
# here will be the rules for the different groups
iptables -A Forward_New -j Forward_Classes
```

Introducing Semantics to iptables MARK's

A MARK is a label that can be assigned to packets. These labels are kept in kernel memory while the packet traverses the IP stack. They do not alter the packet data in any way. The chains in the *mangle* table are used to assign such a label to a packet. Later in the *filter* table the label can be used for filtering. Bitmasks can be used to match whole groups of MARK's.

Let us add a rule to the *mangle* table which marks packets from 134.109.108.237 with the MARK 68760:

```
iptables -t mangle -A FORWARD -s 134.109.108.237 -j MARK --set-mark 68760
```

If we want to filter traffic based on that MARK, we have to add a rule to the *filter* table:

```
iptables -A Forward_New -m mark --mark 68760 -j Forward_User.1
```

This seems to be redundant, as we could have just written:

```
iptables -A Forward_New -s 134.109.108.237 -j Forward_User.1
```

But if the person who uses 134.109.108.237 has a second IP, for example 134.109.109.25, we just have to add another rule to the *mangle* table that assigns the other IP address the same iptables MARK. This makes handling of dynamic IP addresses much easier, since the filter rules do not have to be changed.

```
iptables -t mangle -A FORWARD -s 134.109.109.25 -j MARK --set-mark 68760
```

Using MARK's has another advantage: the MARK's can be used by other parts of the Linux IP stack as well. Using the `iproute2` tool, we can make routing decisions based on MARK's and the Linux traffic shaper tool `tc` can be configured to use MARK's for classifying traffic. This helps to reduce the complexity in large scale setups.

However, in the previous example it is not possible to apply the same rules to groups of users without duplication. MARK's are the solution again. We just have to enhance the MARK's with semantics. Every MARK is just a number[2]. Netfilter allows us to apply bitmasks to the MARK's, so we can divide the bit space to implement some semantics. We use the following approach: the lower 16 bit are reserved for user identifiers which is sufficient for over 65000 users. The upper 15 bits identify the network service level and the most significant bit identifies, if bandwidth has to be limited or not. Each network service level is defined by a set of rules that is applied to all users grouped in that level. By shifting around those numbers different requirements can be accommodated.

We will assign MARK's to every user independent of his location. To connect the assigned MARK to the users ip addresses, different sources can be used: static mapping, or dynamic IP addresses learned from RADIUS authenticated wireless LAN or 802.1x sessions. Whenever a new IP address becomes valid, we just have to append or alter the marking rule for that IP address. User and group specific rules are applied automatically.

[2]MARK's are stored as *unsigned long* in the kernel

Filtering by Service Level

In the previous section we have seen how to use the MARK operator put a label on each packet. We have used bits of our MARK to signify a service level. All that is left to do, is to use those MARK's as filter criteria for our service level groups.

At the end of the section about connection tracking we have created a chain, which was supposed to hold the rules for each service level. We will now fill this chain:

```
# Service Level 1 - no further restrictions
iptables -N Forward_Class.1
iptables -A Forward_Class.1 -j ACCEPT
iptables -A Forward_Classes -m mark \
        --mark 0x00010000/0x7FFF0000 -j Forward_Class.1
# Service Level 2 - only http
iptables -N Forward_Class.2
iptables -A Forward_Class.2 -p tcp --dport 80 -j ACCEPT
iptables -A Forward_Classes -m mark \
        --mark 0x00020000/0x7FFF0000 -j Forward_Class.1
```

If we need more specific rules within each service level, we can add a another chain and match rule for those cases. Let us assume that within the second service level there is a user who is also permitted FTP:

```
iptables -N Forward_User.2.1
iptables -A Forward_User.2.1 -p tcp --dport 21 -j ACCEPT
iptables -I Forward_Class.2 -m mark \
        --mark 0x00001234/0x0000FFFF -j Forward_User.2.1
```

We only have to accept traffic to port 21 and let connection tracking and its helpers handle the way back. It is not necessary to insert our exception rule in first place. In this case, we are extending access. The other case would be, if we wanted to put special restrictions on a user in service level 1. Here, we must put the match rule first or else it would not have any effect. The rule before is an unconditional ACCEPT, which will end processing rules in this chain.

Using connection tracking

In recent Linux kernels, connection tracking has been extended to marking. Instead of the ipt_MARK module, the ipt_CONNMARK module has to be used for setting marks in the mangle table as well as for matching in the filter table. The MARK target changes to CONNMARK. CONNMARK stores the mark with the connection tracking data structures in the kernel and applies the saved mark to all packets of the connection.

```
1 iptables -t mangle -A FORWARD -j CONNMARK --restore-mark
2 iptables -t mangle -A FORWARD -m connmark ! --mark 0 -j ACCEPT
3 iptables -t mangle -N Marking
4 iptables -t mangle -A FORWARD -j Marking
5 iptables -t mangle -A FORWARD -j CONNMARK --save-mark
```

Rule 1 restores the mark that is saved with the connection tracking data. If a mark can be restored, a value different from zero will be restored. So we stop processing the `FORWARD` chain in the *mangle* table. If it is a new connection, we have to assign the marks in the `Marking` chain and save it into the connection tracking data structures.

Traffic Shaping

If we assign marks to every user, it would be a good, if we could also use them for other purposes. Many sites will have a bandwidth limitation policy based on traffic consumption. As stated before, the iptables MARK is commonly used as criteria for `iproute2` and `tc`. The `tc` tool is a command-line programm with many subcommands, which is used to create queuing disciplines (`tc qdisc`) and classes (`tc class`) for traffic control. There are many different schedulers available in the Linux kernel, such as CBQ or HTB. CBQ (Class Based Queuing) is well known but rather difficult to get right and is therefore error-prone. Better and easier to understand is the Hierarchical Token Bucket or HTB scheduler. For further detail please refer to the Linux Advanced Routing and Traffic Control HOWTO[2].

We will assume that a proper queuing strategy exists with the following properties: Every user is giving a class which is rate limited according to his traffic consumption. The higher the average, the lower the bandwidth. If performance is crucial, users could be grouped to reduce the number of classes. We assume only inbound traffic will be accounted and rate limited. A global factor controls the rate of each user class to ensure a monthly quota is never exceeded.

We have to tell the kernel which packets should go in each class. One way to do so is the `tc filter` command. It has different match operators such as `cls_u32` or `cls_fw`. The latter matches MARKS's set by iptables. This leaves us with two choices: use `tc` to setup our queuing strategy and assign traffic to classes or use the CLASSIFY target of iptables. There is nothing wrong with `tc filter`, but we want to use the better packet selection abilities of iptables.

The CLASSIFY Target

This target has been in the netfilter patch-o-matic-ng for some time and has reached the mainline 2.6 kernel recently. With this target, it is possible to directly set `skb->priority`, better known as class identifier, with iptables. It is only valid in the `POSTROUTING` chain of the *mangle* table. Making use of the assigned MARK's, we can do the following:

```
iptables -t mangle -A POSTROUTING -o eth1 -m connmark --mark 1234 \
        -j CLASSIFY --set-class 1:4321
```

This will match traffic with the MARK 1234 and assign it to class 1:4321 attached to interface `eth0` (internal interface). This fulfills our premise to rate limit only downstream. To improve performance, a segmentation strategy will limit the number of rules to examine until a match is found.

Performance

We have still left an important part out of the picture: performance. After gathering empirical data by running this setup for some time, we observed that with connection tracking the ratio between ESTABLISHED/RELATED and NEW packets is roughly 30:1. That means only 3,3% of all forwarded packets go through our `Forward_New` chain in the *filter* table. The same applies for the *mangle* table, if CONNMARK is used. In terms of traffic this results in even more impressive numbers: about 99.5% of all forwarded traffic is handled by **3** rules:

1. *mangle* table: restore MARK if exists
2. *mangle* table: MARK restored? → ACCEPT
3. *filter* table: ESTABLISHED/RELATED? → ACCEPT

Summary

In this paper we presented a concept for a packet filter. It seamlessly integrates filtering, dynamic IP address handling and bandwidth limiting in one rule set. However there is still room for improvement: QoS requires prior creation of appropriate qdiscs and classes. Explicitly setting a mark limits the host count to less than 2000 hosts and very few groups with very few rules. It is possible to reduce the number of MARK rules in the *mangle* table by using the IPMARK target instead. IPMARK is currently part of the extra section in netfilters patch-o-matic-ng. In addition to IPMARK, bitwise MARK-operations can further reduce the numer of required rules. Nevertheless, the amount of mark rules should not really matter, as they will only be examined for a very small percentage of packets. However, if short on memory or if serving tens of thousands of users, memory consumption could become an issue. In those cases a different strategy or even a completely different solution such as nf-hipac, ipset or a even a hardware firewall could be more viable.

References

[1] *IP Sets*.
URL `http://ipset.netfilter.org/`

[2] *Linux Advanced Routing and Traffic Control HOWTO*.
URL `http://www.lartc.org`

[3] *Netfilter Documentation*.
URL `http://www.netfilter.org/documentation/index.html`

[4] *nf-HiPAC*.
URL `http://www.hipac.org/`

[5] Oskar Andreasson: *Iptables Tutorial 1.2.0*, 2005.
URL `http://iptables-tutorial.frozentux.net/`

[6] Frank Bernard: *Schnellschutz*, *Linux Magazin*, (6), Jun. 2000.
URL `http://www.linux-magazin.de/Artikel/ausgabe/2000/06/IPTables/iptables.html`

[7] Frank Bernard: *Sieben mal Sieben*, *Linux Magazin*, (7), Jul. 2000.
URL `http://www.linux-magazin.de/Artikel/ausgabe/2000/07/IpTables/IpTables.html`

[8] Chris Clancey, Harry Goldschmitt, John Kastner, Eric Oberlander and Peter Walker: *IPCop Administrative Guide*, 2004.
URL `http://www.ipcop.org/1.4.0/en/admin/html/`

[9] David Coulson: *Mastering IPTables*, 2001.
URL `http://davidcoulson.net/writing/lxf/14/iptables.pdf`

[10] Jósef Kadlecsik and György Pástor: *Netfilter Performance Testing*.
URL `http://people.netfilter.org/kadlec/nftest.pdf`

[11] Rusty Russel: *Linux 2.4 Paket Filtering HOWTO*, 2002.
URL `http://www.netfilter.org/documentation/HOWTO/packet-filtering-HOWTO.html`