Next Previous Contents

1. Introduction

Hi guys.

This document is a journey; some parts are well-traveled, and in other areas you will find yourself almost alone. The best advice I can give you is to grab a large, cozy mug of coffee or hot chocolate, get into a comfortable chair, and absorb the contents before venturing out into the sometimes dangerous world of network hacking.

For more understanding of the use of the infrastructure on top of the netfilter framework, I recommend reading the Packet Filtering HOWTO and the NAT HOWTO. For information on kernel programming I suggest Rusty's Unreliable Guide to Kernel Hacking and Rusty's Unreliable Guide to Kernel Locking.

1.1 What is netfilter?

netfilter is a framework for packet mangling, outside the normal Berkeley socket interface. It has four parts. Firstly, each protocol defines "hooks" (IPv4 defines 5) which are well-defined points in a packet's traversal of that protocol stack. At each of these points, the protocol will call the netfilter framework with the packet and the hook number.

Secondly, parts of the kernel can register to listen to the different hooks for each protocol. So when a packet is passed to the netfilter framework, it checks to see if anyone has registered for that protocol and hook; if so, they each get a chance to examine (and possibly alter) the packet in order, then discard the packet (`NF_DROP`), allow it to pass (`NF_ACCEPT`), tell netfilter to forget about the packet (`NF_STOLEN`), or ask netfilter to queue the packet for userspace (`NF_QUEUE`).

The third part is that packets that have been queued are collected (by the ip_queue driver) for sending to userspace; these packets are handled asynchronously.

The final part consists of cool comments in the code and documentation. This is instrumental for any experimental project. The netfilter motto is (stolen shamelessly from Cort Dougan):

```
``So... how is this better than KDE?''
```

(This motto narrowly edged out `Whip me, beat me, make me use ipchains').

In addition to this raw framework, various modules have been written which provide functionality similar to previous (pre-netfilter) kernels, in particular, an extensible NAT system, and an extensible packet filtering system (iptables).

1.2 What's wrong with what we had in 2.0 and 2.2?

1. No infrastructure established for passing packet to userspace:
   - Kernel coding is hard
   - Kernel coding must be done in C/C++
   - Dynamic filtering policies do not belong in kernel
   - 2.2 introduced copying packets to userspace via netlink, but reinjecting packets is slow, and subject to `sanity' checks. For example, reinjecting packet claiming to come from an existing interface is not possible.
2. Transparent proxying is a crock:
   - We look up **every** packet to see if there is a socket bound to that address
   - Root is allowed to bind to foreign addresses

- Can't redirect locally-generated packets
- REDIRECT doesn't handle UDP replies: redirecting UDP named packets to 1153 doesn't work because some clients don't like replies coming from anything other than port 53.
- REDIRECT doesn't coordinate with tcp/udp port allocation: a user may get a port shadowed by a REDIRECT rule.
- Has been broken at least twice during 2.1 series.
- Code is extremely intrusive. Consider the stats on the number of #ifdef CONFIG_IP_TRANSPARENT_PROXY in 2.2.1: 34 occurrences in 11 files. Compare this with CONFIG_IP_FIREWALL, which has 10 occurrences in 5 files.

3. Creating packet filter rules independent of interface addresses is not possible:
   - Must know local interface addresses to distinguish locally-generated or locally-terminating packets from through packets.
   - Even that is not enough in cases of redirection or masquerading.
   - Forward chain only has information on outgoing interface, meaning you have to figure where a packet came from using knowledge of the network topography.
4. Masquerading is tacked onto packet filtering:

   Interactions between packet filtering and masquerading make firewalling complex:

   - At input filtering, reply packets appear to be destined for box itself
   - At forward filtering, demasqueraded packets are not seen at all
   - At output filtering, packets appear to come from local box

5. TOS manipulation, redirect, ICMP unreachable and mark (which can effect port forwarding, routing, and QoS) are tacked onto packet filter code as well.
6. ipchains code is neither modular, nor extensible (eg. MAC address filtering, options filtering, etc).
7. Lack of sufficient infrastructure has led to a profusion of different techniques:
   - Masquerading, plus per-protocol modules
   - Fast static NAT by routing code (doesn't have per-protocol handling)
   - Port forwarding, redirect, auto forwarding
   - The Linux NAT and Virtual Server Projects.
8. Incompatibility between CONFIG_NET_FASTROUTE and packet filtering:
   - Forwarded packets traverse three chains anyway
   - No way to tell if these chains can be bypassed
9. Inspection of packets dropped due to routing protection (eg. Source Address Verification) not possible.
10. No way of atomically reading counters on packet filter rules.
11. CONFIG_IP_ALWAYS_DEFRAG is a compile-time option, making life difficult for distributions who want one general-purpose kernel.

1.3 Who are you?

I'm the only one foolish enough to do this. As ipchains co-author and current Linux Kernel IP Firewall maintainer, I see many of the problems that people have with the current system, as well as getting exposure to what they are trying to do.

1.4 Why does it crash?

Woah! You should have seen it **last** week!

Because I'm not as great a programmer as we might all wish, and I certainly haven't tested all scenarios, because of lack of time, equipment and/or inspiration. I do have a testsuite, which I encourage you to contribute to.

Next Previous Contents