Network packet capturing for Linux

Skill Level: Introductory

Ashish Chaurasia (achauras@in.ibm.com)
Computer Science Engineer
IBM

13 Apr 2004

This tutorial covers different mechanisms for capturing and manipulating packets on Linux®. Security applications -- such as VPNs, firewalls, and sniffers, and network apps such as routers -- rely on methods like those described here to do their work. Once you have the hang of them, you will rely on them too.

Section 1. Before you start

About this tutorial

Information these days spends much of its time encapsulated in packets, flowing through pipes known as networks. In this tutorial, we will be capturing those packets as they flow, and the platform on which we will be capturing them is Linux.

Most network applications -- from Virtual Private Networks (VPNs) to routers to sniffers -- have some sort of packet capturing mechanism. So anyone involved with writing such software can benefit from this tutorial.

Because several of the packet filtering mechanisms we will examine are kernel modules, we will also briefly cover those, as well as kernel compilation.

We will also review some of the mechanisms that were unsuccessful for me: I could get only a copy of the packet instead of intercepting the original. This discussion can not only save you the trouble of duplicating my efforts, but can also be useful for network applications like sniffers.

Apart from a general familiarity with the different packet capturing mechanisms including firewall hooks, divert socket, and netfilter; readers will also gain knowledge about Linux networking and the TCP/IP stack. There is even some source code to get you started.
Prerequisites

This tutorial is best suited to readers with some experience with system programming, Linux networking, and Linux kernel modules. However, this tutorial aims to keep concepts as simple as possible and to give detailed explanations where appropriate.

Section 2. Introduction to packet capturing

The TCP/IP stack is an important pillar of Linux networking and is architected beautifully. A packet is the carrier of information flow in the TCP/IP stack. Much of the meaningful work in Linux network programming consists of capturing these information-rich packets, and extracting or manipulating the information they contain.

What we mean by packet capturing is some mechanism to catch hold of a packet until our required purpose is solved and then release it, so that it can follow its regular route through any remaining processing. Other terms for the same or similar actions include packet filtering, packet sniffing, network sniffing, and packet or network monitoring.

Packet capturing is the foundation of a large range of network software, including network monitoring tools, VPNs, network analyzers, routers, Intrusion Detection Systems (IDS), and sniffer.

Linux provides several places where packets may be captured, both in userspace and in kernelspace. The figure below shows the path a network packet follows through the TCP/IP stack:

**Figure 1. Path of a packet through the stack**
As you can see in the figure, incoming packets pass through the ethernet layer, the network layer, the TCP stack, and through sockets before they are copied to userspace. At all of these breakpoints the packets are vulnerable to capture. The methods discussed in this tutorial work mostly at the network layer and in userspace.

Due to the evolution of Linux and of the Linux kernel, some of the packet capturing methods covered here work only with a specific kernel. For instance, divert socket works with a patched 2.2.12 kernel, while netfilter works with kernels 2.4.x and 2.6.x. Specifics will always be noted in the discussion of the various methods.

Section 3. Linux loadable kernel module (LKM)

Modules that capture packets

Most of the packet capturing mechanisms we will examine in this tutorial work as Linux kernel modules, and so we will briefly discuss those now. If you already have a strong background in LKM, you may skip to the next section, Compiling the kernel.

What is an LKM?

A loadable kernel module is an extension of the kernel that can be attached to or
removed from the kernel on an as-needed basis. LKMs are basically a software implementation of a device driver, that work with either a real or a virtual hardware device. When loaded into the kernel, they listen for and handle any requests to their device. Because you load only the LKMs that you need, the use of LKMs makes the kernel light, modular, flexible, and scalable.

LKM architecture

The architecture of writing a Linux kernel module is as follows:

1. **init_module**
   This initialization function of the LKM is mandatory. This function gets fired when the LKM is loaded into the kernel, so all the initialization is done in this function.

2. **Processing specific functions**
   These functions do the actual work, such as reading, writing, opening, closing the device, and so on.

3. **cleanup_module**
   This mandatory function of LKM gets called when LKM is removed from the kernel. So all cleanup work like freeing up memory and the like should be done here.

Compiling the LKM

After writing an LKM, you can compile it using:

```bash
gcc -c -g <module_name>.c -I/usr/src/linux/include
```

This command gives you a file named `<module_name>.o`, which is your LKM that you can load in the kernel with this command:

```bash
insmod -f <module_name>.o
```

(The `-f` option is for forceful loading.)

If you experience any kernel version issues while loading your module, you can resolve them by including the header files of that particular kernel when you load the LKM -- hence the use of `-I/usr/src/linux/include` when we compiled the kernel.

You can also use a Makefile to resolve such versioning problems, but that discussion is outside the scope of this tutorial. You can find more information on LKM programming in Resources) at the end of this tutorial.
Unloading the LKM

Once the module is loaded into the kernel, it will start performing its intended functionality. You can see the list of all the currently loaded LKMs with the command `lsmod`.

You can load or remove the module from the kernel using `rmmod <module_name>`.

---

Section 4. Compiling the kernel

Many of the mechanisms we discuss in this tutorial require setting some kernel options and then recompiling the kernel. So before we begin, we'll review the steps of kernel compilation. Again, if you feel comfortable with kernel compilation already, feel free to skip ahead to the next section, Packet interception: Firewall hooks, which begins our review of packet capturing mechanisms.

To recompile the kernel:

1. `make xconfig`  
   Allows you to set different kernel options. (You can also use `make menuconfig` or `make config`; it's just a matter of preference.)

2. `make dep`  
   Resolves file dependencies for kernel compilation.

3. `make bzImage`  
   Makes the kernel image and stores it in `vmlinuz`.

4. `make install`  
   Copies `vmlinuz` to the `/boot` directory and copies the binaries to the proper place.

5. `make modules`  
   Makes all of the kernel modules.

6. `make modules_install`  
   Installs the modules in the proper place (usually `/lib/modules/<kernel-version>`).

Next you should add the information about the newly compiled kernel to `/etc/lilo.conf` (if it is not already there), and reboot the machine. You will see the new kernel options listed at the start of the reboot process.
Section 5. Packet interception: Firewall hooks

Overview

Firewall hooks were introduced with the 2.2.16 kernel, and were the packet interception method for the run of the 2.2.x kernels. Firewall hooks intercept packets at the IP layer of the TCP/IP stack.

Firewall hooks function as a Linux loadable kernel module (LKM), or pseudo device driver, and can be loaded or unloaded from the kernel according to need.

To use firewall hooking, enable the firewalling options (listed under Networking options) during kernel compilation.

You can use this packet interception mechanism to develop routers, VPNs, packet sniffers, or any other network application that sits at the edge of a network and that requires the capture of packets in real time.

Filling in the structure

The kernel-defined firewall_ops structure is the foundation of firewall hooks. You can use these to specify a vast range of packet policing policies from the very specific to the entirely generic.

The firewall_ops structure is in /usr/src/linux/include/linux/firewall.h and looks like this:

```c
struct firewall_ops
{
    struct firewall_ops next;
    int (*fw_forward)(struct firewall_ops *this, int pf,
                      struct device *dev, void *phdr, void *arg,
                      struct sk_buff **pskb);
    int (*fw_input)(struct firewall_ops *this, int pf,
                    struct device *dev, void *phdr,
                    void *arg, struct sk_buff *pskb);
    int (*fw_output)(struct firewall_ops *this, int pf,
                     struct device *dev, void *phdr,
                     void *arg, struct sk_buff **pskb);
};
```
This mechanism is implemented as an LKM. As such, it requires `init_module` (for initializing the module), `cleanup_module`, and some processing-specific functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
<td>Pointer to next hook</td>
</tr>
<tr>
<td>fw_forward</td>
<td>Function pointer for forwarded packets</td>
</tr>
<tr>
<td>fw_input</td>
<td>Function pointer for incoming packets</td>
</tr>
<tr>
<td>fw_output</td>
<td>Function pointer for outgoing packets</td>
</tr>
<tr>
<td>fw_pf</td>
<td>Protocol family</td>
</tr>
<tr>
<td>fw_priority</td>
<td>Priority of chosen firewalls</td>
</tr>
</tbody>
</table>

These are actually function pointers; the corresponding functions are defined below.

**Functions**

Three functions need to be defined in order to process packets:

**Incoming packet processing:**

```c
static int fw_input(struct firewall_ops *this, int pf, struct device *dev, void *phdr, void *arg, struct sk_buff **pskb);
```

This function is called for all incoming packets. If any processing (also known as "mangling") is needed on incoming packets, such as the addition of an extra field, it can be done here.

**Outgoing packet processing:**

```c
static int fw_output(struct firewall_ops *this, int pf, struct device *dev, void *phdr, void *arg, struct sk_buff **pskb);
```

This function is called for all outgoing packets, those originating from the host. Any extra processing of such packets can be done here.

**Forwarded packet processing:**
This function is called for all forwarded packets.

Here are more details on the arguments passed in the above functions.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*this</td>
<td>Pointer to the firewall hook structure</td>
</tr>
<tr>
<td>pf</td>
<td>Stands for protocol family</td>
</tr>
<tr>
<td>*dev</td>
<td>Pointer to the device structure of the ethernet card</td>
</tr>
<tr>
<td>*phdr</td>
<td>Pointer to the buffer of the IP address</td>
</tr>
<tr>
<td>*arg</td>
<td>Extra arguments, if any need to be passed to the function</td>
</tr>
<tr>
<td>**pskb</td>
<td>Pointer to the sk_buff layer of the TCP_IP stack</td>
</tr>
</tbody>
</table>

For each of the functions above, the return values (or actions) on packets may be:

- **Accept**
  This is done by `FW_ACCEPT` macro. It accepts the packets, and packets follow the normal TCP/IP stack path.

- **Reject**
  This is done by `FW_REJECT` macro. It rejects all the packets, and no network traffic goes out or comes in.

- **Redirect**
  This is done by `FW_REDIRECT` macro. It redirects the packets to a particular host.

Now we will assign the functions defined above to the pointers in the `firewall_ops` structure. Once this is done, our `firewall_ops` structure will be complete and the functions will be called by the system itself -- hence the term `callback functions`.

Here are the fields still to be filled in:

```c
struct firewall_ops * next; /*Pointer to the next firewall hook structure */
int fw_pf; /* Protocol family */
int fw_priority; /* Priority of chosen firewalls */
```

The firewall structure should now look something like this:
static struct firewall_ops myOps = {NULL, 
    fw_forward, 
    fw_input, 
    fw_output, 
    PF_INET,  /* Protocol family */ 
    1 /* Priority of chosen firewalls*/
};

Registering with the kernel
Now you need to register the firewall_ops structure with the kernel:

```c
register_firewall(protocol family, struct firewall_ops *);
```

Do this in the LKM's init_module code.

Unregister the firewall_ops structure
When the LKM is unloaded, the cleanup_module function should include

```c
unregister_firewall(protocol family, struct firewall_ops *);
```

Once your LKM is coded, compiled, and loaded into the kernel, your packet interceptor is ready to rock and roll.

Complete source code and usage instructions for this LKM are given on the next two sections.

Source code: Firewall hooks program

```c
/* This is an insertable module that uses the firewall hooks mechanism on 2.2.16 to intercept a packet */
/* gcc -O -c NetFWHook.c -I/usr/src/linux/include*/
/* No one can then telnet,ftp,ping your machine */

/*NetFWHook.c*/
#define MODULE
#define __KERNEL__
```
```c
#include<linux/config.h>
#include<linux/module.h>
#include<linux/version.h>

#include<linux/netdevice.h>
#include<net/protocol.h>
#include<net/pkt_sched.h>
#include<net/tcp.h>
#include<net/ip.h>

#include<linux/if_ether.h>
#include<linux/ip.h>
#include<linux/tcp.h>
#include<linux/skbuff.h>
#include<linux/icmp.h>

#include<linux/kernel.h>
#include<linux/mm.h>
#include<linux/file.h>
#include<linux/firewall.h>
#include<asm/uaccess.h>

//Function for forwarded packets
static int fw_forward(struct firewall_ops *this, int pf, struct device *dev, void *phdr, void *arg, struct sk_buff **pskb)
{
    struct iphdr *hdr = (struct iphdr *)(*pskb)->h.ipiph;
    printk("\n\tfw_forward()() called...");
    printk("\n\tThe source of this packet is: %s", in_ntoa(hdr->saddr));
    return FW_ACCEPT;
}

/*Function for incoming packets*/
static int fw_input(struct firewall_ops *this, int pf, struct device *dev, void *phdr, void *arg, struct sk_buff **pskb)
{
    struct iphdr *iph;
    iph = (struct iphdr *)(*pskb)->h.ipiph;
    printk("\n\tfw_input()() called...");
```
printk("\n\tThe source of this packet is: %s", in_ntoa(iph->saddr));

    return FW_ACCEPT;
}

/*Function for outgoing packets*/
static int fw_output(struct firewall_ops *this, int pf, struct device *dev, void *phdr, void *arg, struct sk_buff **pskb)
{
    struct iphdr *iph;
    iph = (struct iphdr*)(*pskb)->h.ipiph;
    printk("\n\tfw_output() called...");
    printk("\n\tThis packet is destined for: %s", in_ntoa(iph->daddr));
    return FW_ACCEPT;
}

/*Filling the firewall_ops structure*/
static struct firewall_ops myOps =
{NULL, fw_forward, fw_input, fw_output, PF_INET int fw_pf; /* Protocol family*/, int fw_priority; /* Priority of chosen firewalls */};

/*First function to be called at the time of loading of module*/
int init_module(void)
{
    /*registering the firewall_ops structure*/
    if(register_firewall(PF_INET, &myOps) < 0)
    {
        printk("\n\terror...firewall main aag lag gaye!!");
        return -1;
    }

    else
    {
        printk("\n\tfirewall registered");
    }
    return 0;
}

/*Function that is called when the module is unloaded*/
void cleanup_module(void)
/*Unregistering the firewall_ops structure*/
    if(unregister_firewall(PF_INET,&myOps)<0)
    {
        printk("\n\n\tError....Firewall can't be unregistered");
    }
    else
    {
        printk("\n\n\tFirewall unregistered");
    }
}

Using the Firewall hooks program

1. Compile and run the firewall hooks program:
   gcc -c -O NetFWHook.c -I/usr/src/linux/include/

2. You will get a file named NetFWHook.o. Insert it into the kernel with the following command:
   /sbin/insmod -f NetFWHook.o

3. To see messages generated by this LKM, run the dmesg command.

Section 6. Packet interception: Netfilter

Overview

Netfilter is the packet interception mechanism provided in kernels 2.4.x and 2.6.x, and replaces ipchains, firewall hooks, and other methods used with kernel 2.2.x. Netfilter, too, is available as an LKM.

To use netfilter, set the Packet Filtering option at the time of kernel compilation.

You can use the netfilter mechanism for the same types of applications that firewall
hooks excel at: routers, packet sniffers, and other entities that sit at the edge of a network and access the traffic flow.

Netfilter on patrol

Netfilter can nab a packet at several well-defined points of its path through the TCP/IP protocol stack:

- **NF_IP_PRE_ROUTING**
  The packet is held after the initial sanity check (checksums and the like)

- **NF_IP_LOCAL_IN**
  The packet is apprehended if it is intended for the local host

- **NF_IP_FORWARD**
  The packet is detained if it is intended for some other machine

- **NF_IP_LOCAL_OUT**
  This purloins locally created packets whose destination is the outside world

- **NF_IP_POST_ROUTING**
  This is the final hook, after which the packet is put on the wire

As a packet traverses the TCP/IP protocol stack, the protocol calls the netfilter framework with the packet and hook number. Hooks may also be assigned priority.

Return values for the functions include:

- **NF_ACCEPT**
  The packet continues the normal TCP/IP path

- **NF_DROP**
  The packet is dropped; no further processing

- **NF_STOLEN**
  The packet has been taken; no further processing

- **NF_QUEUE**
  Queue the packet (usually for userspace handling)

- **NF_REPEAT**
  Fire this hook again

Steps

Netfilter works in much the same way as firewall hooks. A structure that you register with the kernel as an LKM calls process-specific functions (hooks). Any netfilter-based packet interceptor must follow the steps used to develop an LKM.
Process-specific functions

The prototype for a process-specific function (or hook) is as follows:

```c
static unsigned int packet_interceptor_hook(unsigned int hook, struct sk_buff **pskb,
const struct net_device *indev, const struct net_device *outdev, int (*okfn) (struct sk_buff *))
```

You can define the fields as:

- **hook**
  The number of the hook you are interested in; for instance, NF_IP_LOCAL_OUT, NF_IP_LOCAL_IN, NF_IP_FORWARD, and so on.

- **pskb**
  A pointer to a packet container in the TCP/IP stack; for example, sk_buff.

- *indev & *outdev
  A pointer to the device structures of incoming and outgoing network devices. Every device (for instance, an ethernet card) that is registered in the kernel has a device structure consisting of fields like IRQ, the IO address, and so on. When a machine has one network interface to handle both incoming and outgoing traffic, these two structures will be the same. When these duties are divided between two devices, they will vary.

- (*okfn) (struct sk_buff *)
  This function is called when the hook is activated.

The netfilter structure

The core netfilter structure is defined in /usr/src/include/linux/netfilter.h and looks like this:

```c
struct nf_hook_ops
{
    struct list_head list;

    /* User fills in from here down. */
    nf_hookfn *hook;
    int pf;
    int hooknum;
    /* Hooks are ordered in ascending priority. */
    int priority;
};
```

The parameters are:

- **list**
Netfilter itself is a chain of hooks; this points to the head of the netfilter hooks and is usually set to \{ NULL, NULL \}.

- **hook**
  This function is called when a packet hits a hook point. This is the same function we described earlier, and it must return NF_ACCEPT, NF_DROP, or NF_QUEUE. If NF_ACCEPT, the next hook attached to that point will be called. If NF_DROP, the packet is dropped. If NF_QUEUE, it's queued. The sk_buff pointer is passed in this function, and is filled with packet information like the IP header, the TCP header, and so on. You can use the sk_buff structure pointer to manipulate or delete the packet (to do the latter, simply set the skb pointer to null).

- **pf**
  The protocol family; for instance, PF_INET for IPv4

- **hooknum**
  The number of the hook you are interested in; for instance, NF_IP_LOCAL_IN and the like

Kernel registration

In the `init_module` function of our LKM, we need to register the structure that we filled in with the kernel:

```c
int nf_register_hook(struct nf_hook_ops *req);
```

Here, `nf_hook_ops` is the netfilter operations structure.

Once this structure is registered with the kernel, Linux will call the functions defined here to handle packet processing.

Unregister the netfilter structure

When the LKM is unloaded, the netfilter structure needs to be unregistered from the kernel. This is done within the `cleanup_module` function:

```c
void nf_unregister_hook(struct nf_hook_ops *req);
```

Again, `nf_hook_ops` is the netfilter operations structure.

---

Section 7. Packet interception: Divert socket
Overview

One of the patches for the 2.2.12 kernel introduced a new type of raw socket called *divert socket*, which filters packets based on firewall specifications and sends them to userspace. From there, the packets can undergo processing or simply be reinjected back into the TCP/IP stack.

Divert socket is a special type of raw socket through which packets can be received and sent just like any other socket. Divert socket works with the firewalling functionality of the Linux kernel: you can set Linux firewalling to apply limited policies on incoming, outgoing, and forwarded packets, and redirect them to a given port. Divert socket will listen at that port, and further redirect the packets.

A major drawback of using divert socket is the overhead of copying packets to userspace, which takes time and resources and degrades network performance.

You can use divert socket for all networking applications that don't push hard limits on real-time processing for packets, such as sniffers.

Kernel compilation options

Linux firewalling is a prerequisite to using divert socket, so you should enable this option at kernel compilation. Some options were introduced into the kernel specifically for divert socket, and you should set them too at compile time:

- **Firewalling**
  Enables the kernel's firewalling capabilities

- **IP Firewalling**
  Enables the packet filter capabilities of the kernel

- **IP:divert sockets**
  Enables divert sockets in the kernel

- **IP:divert pass-through**
  If this option is set and no application is listening at the divert socket port, the packet isn't diverted and simply follows its regular path

- **IP:always defragment**
  If set, all defragmentation is done in-kernel, which severely degrades performance

Ipcchains control the path that a network packet follows. To use ipchains with divert socket, you should download, compile, and install the ipchains-1.3.9 patched version of ipchains.

Introduction to ipchains
• **Input chain**
  Is traversed by all incoming packets (packets destined for the host and forwarded packets)

• **Output chain**
  Is traversed by all outgoing packets (packets originating from the host and forwarded packets)

• **Forward chain**
  Is traversed by forwarded packets only

A forwarded packet traverses the chains in this order:

1. Input
2. Forward
3. Output

---

Section 8. Packet interception: Kernel tampering

Overview

Here we will manipulate the kernel source code to extend the kernel's native packet interception capabilities.

The kernel source lives in `/usr/src/linux` and the two main files to modify are `ip_input.c` and `ip_output`. These two files are in the `/usr/src/linux/net/ipv4/` directory.

Source file modifications: `ip_input.c`

`ip_input.c` has these two functions, which are invoked for all incoming packets:

```c
ip_local_deliver(struct sk_buff, struct device, struct packet_type)
```

Is executed for all incoming packets regardless of destination

```c
ip_rcv(struct sk_buff, struct device, struct packet_type)
```

Is called for all packets destined for the local machine
• **sk_buff**  
The container for packets in the TCP/IP stack  

• **device**  
The device structure  

• **packet_type**  
The type of packet: incoming or outgoing  

`ip_rcv` performs several tasks that are required at the IP layer including reassembly, checksum calculations, and the like.

Since `ip_rcv` processes all incoming IP packets, it is a good place to add any additional code that you want executed on incoming packets. Your other option is to write your own `ip_rcv` routine.

### Source file modifications: ip_output.c

Like `ip_output`, `ip_output.c` lives in the `/usr/src/linux/net/ipv4/` directory, and it works on outgoing packets. Its four functions are:

- `ip_build_and_sent_pkt(struct sk_buff, struct sock, u32 saddr, u32 daddr, struct ip_options)`
  
  Adds an IP header to `sk_buff` and sends it out

- `ip_queue_xmit(struct sk_buff)`
  
  Queues the packet for sending

- `ip_build_xmit_slow()`
  
  Builds and transmits the packet

- `ip_build_xmit()`
  
  Builds and transmits the packet "fast," an option for unfragmented packets only.

Only `ip_build_xmit_slow` and `ip_build_xmit` process all outgoing IP packets, so any extra processing can be done at either of these stages. As with `ip_input.c`, your other option is to write your own routines, then call the original function from within those. The tactic you choose will depend on what you are implementing.

### Source file modifications using Makefiles
You can implement tampering with the kernel source code dynamically using Makefiles; the Makefile script ensures that the required changes to the source code are made. You can add or remove your packet interceptor with a single command (make install or make uninstall) plus kernel recompilation and rebooting.

Because this mechanism requires tampering with the Linux kernel source code, debugging can quickly become very difficult. Moreover, because this method requires recompilation of the kernel, the process is lengthy, complex, and inflexible.

Section 9. Packet interception: Making copies of packets

Overview

Some packet-capturing mechanisms deliver only copies of the packets instead of the packets themselves. However, as mentioned before, copies of packets can still be useful for developing network sniffer applications.

Protocol handler

The protocol handler is another mechanism that works as an LKM.

The protocol handler applies processing to packets that come via the ethernet card. To work, the protocol handler must be registered with a device structure; in our example, you would register it with the device structure of the ethernet card.

The ethernet card device structure also contains a pointer to an init function, which initializes the device.

Implementing the protocol handler

To implement the protocol handler:

1. Create one instance of the device structure for the ethernet interface and register this with the kernel.
2. Register the handler routine for applying processing on packets with the device structure.

Now any packets that come through the ethernet interface will have the proper processing applied to them. The processing will be applied to a copy of the packet, not to the original. It is not possible to divert the packet from its original path through
the TCP/IP stack, and so only a copy of the packet reaches the protocol handler itself.

Interrupt handler

As its name implies, this mechanism is based on interrupt handling and can also be implemented as an LKM. Whenever a packet comes in at the ethernet card, an interrupt is generated. By writing your own interrupt handler, you can capture the packet and perform any required processing on it.

Fundamentally, whenever a packet traverses the TCP/IP stack, an interrupt is generated, which sets flags in the ethernet card indicating whether the packet is incoming or outgoing. The interrupt is usually interrupt 9, so you can write an interrupt handler to handle interrupt 9 signals and process the packets.

In this case, as with the protocol handler, only a copy of the packet is sent to the interrupt handler, while the original traverses its course through the TCP/IP stack unimpeded.

Section 10. Summary of packet interception mechanisms

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Modular?</th>
<th>Scalable?</th>
<th>Performance (TCP/IP throughput)</th>
<th>Insertable/removable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firewall hooks</td>
<td>Yes, you can insert it as an LKM</td>
<td>Yes, you can enhance it by changing the source code of the LKM and recompiling it</td>
<td>Degradation of approximately 50%</td>
<td>Yes, using insmod command</td>
</tr>
<tr>
<td>Netfilter</td>
<td>Yes, you can insert it as an LKM</td>
<td>Yes, you can enhance it by changing the source code of the LKM and recompiling it</td>
<td>Degradation of approximately 50%</td>
<td>Yes, using insmod command</td>
</tr>
<tr>
<td>Divert socket</td>
<td>No, the divert socket patch must be directly applied to the kernel code</td>
<td>Tedious because the whole kernel needs to be compiled</td>
<td>Too slow to be measured</td>
<td>No, because it is not modular</td>
</tr>
<tr>
<td>Kernel tampering</td>
<td>No, the packet interceptor code must be directly integrated into</td>
<td>Tedious because the whole kernel needs to be compiled</td>
<td>Usually faster than LKM methods (if implemented)</td>
<td>Yes, if using a Makefile for installation and uninstallation</td>
</tr>
</tbody>
</table>
Firewall hooks and netfilter are modular, flexible, and dynamically loaded (and removed), and they inflict only a moderate performance toll on your systems.

For applications prone to heavy traffic or bounded by the hard limits of real-time processing, the kernel tampering method would be most suitable.

You can use Divert socket if there is no timing issue with packet processing. Divert socket can be a very useful way of analyzing and controlling network traffic in userspace.
Resources

Learn

• *Linux Device Drivers, 2nd Edition* by Alessandro Rubini and Jonathan Corbet (O'Reilly & Associates, 2001) naturally covers device driver programming; it also discusses the use of Makefiles for resolving version issues.

• *Linux Kernel Internals* by Michael Beck, Harold Bohme, Ulrich Kunitz, Robert Magnus, Mirko Dziadzka, and Dirk Verworner (ACM 1996) explains the inner mechanisms of Linux from process scheduling to memory management and file systems, and also covers the kernel itself.

• The *Divert Sockets mini-HOWTO* has a great deal of information on this method of packet interception.

• For more information on Netfilter, see the netfilter Web site or the *Linux netfilter Hacking HOWTO*.

• For newer kernels, refer to "Linux 2.4 Packet Filtering HOWTO," one of Rusty's Remarkably Unreliable Guides (others include the Networking Concepts HOWTO, the Linux 2.4 NAT HOWTO, and the Netfilter Hacking HOWTO cited above).

• The *Linux iptables HOWTO: using iptables* describes how to use iptables to filter out bad packets for Linux kernels 2.3.15 and beyond.

• The *Linux 2.4 Advanced Routing HOWTO* covers the use of iproute2 -- a good choice when tunneling is needed.

• Get an introduction to TCP/IP and learn how to *set up a LAN using Red Hat Linux*.

• The *Network Address Monitoring and Messaging API* is a Java API for monitoring interface addresses and programmatically responding to their changes.

• The *Guide to IP Layer Network Administration with Linux* includes an excellent section of *Links to Documentation*, covering everything from general networking to iproute2, netfilter, ipchains, and much more. Their *Links to Software* are also quite wonderful.

• *Phrack* magazine covers topics ranging from security and hacking to radio broadcasting and world news -- as well as many interesting articles on the Linux kernel and kernel modules.

• The verse "If a packet hits a pocket on a socket on a port" is taken from *A Grandchild's Guide to Using Grandpa's Computer*.

• For more on LKMs, see the papers, "The Linux Kernel Module Programming Guide" (Peter Jay Salzman and Ori Pomerantz, 2001) and "Linux Loadable Kernel Module HOWTO" (Bryan Henderson, 2004).

• The *Fairly Fast Packet Filter (FFPF)* minimizes packet copying and is intended as an alternative to netfilter.
• Find more tutorials for Linux developers in the developerWorks Linux one.
• Stay current with developerWorks technical events and Webcasts.

Get products and technologies

• Order the SEK for Linux, a two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
• Download IBM trial software directly from developerWorks.

Discuss

• Read developerWorks blogs, and get involved in the developerWorks community.

About the author

Ashish Chaurasia
Ashish Chaurasia is a Computer Science Engineer currently working on Storage Area Network (SAN) file systems at IBM India. He recently completed work on a Virtual Private Network (VPN) project, during which he researched and compiled the resources that led to this tutorial. You can contact him at achauras@in.ibm.com.