picpics, Fotolia

Building a Netfilter firewall module

SINGING LESSON

We'll show you how to build your own Netfilter extension with this example of a musical firewall. **BY MARK VOGELSBERGER**

Netfilter is the Linux kernel subsystem behind firewall tools such as the famous Iptables. The Netfilter subsystem provides the structure for packet filtering and address translation by offering a series of hooks into the network protocol stack.

You can find many commands, scripts, and front-end applications for accessing the Netfilter subsystem – including tools such as Shorewall and Firestarter, as well as the native Iptables – so you don't have to be a programmer to access the powers of Netfilter. However, if you are ready for a little programming, you can use the built-in Netfilter hooks to create your own custom firewall modules.

This article takes you through the steps of building a custom Netfilter module. The discussion focuses on the example of a "singing" firewall that plays a sound whenever a packet arrives, however, the concepts in this article also applies to your own creative uses of the Netfilter subsystem.

A Singing Firewall

Firewall log entries keep administrators up to date on data packets, but from a physiological and psychological point of view, peering at log files all day can become tedious, and important information can get lost. Of course, you'll need the log files anyway, but why not have a secondary alert system?

The Singwall module described in this article provides a means for the admin to listen to network traffic in real-time. Singwall is a singing firewall extension for Netfilter. The firewall chirps when a packet arrives, and the pitch of the sound varies depending on the port number addressed by the packet.

Hooked

The Netfilter model lets developers program kernel modules that hook into the chain of execution for network packets. The first task the module needs to handle is registering a hook. This registration tells the kernel what the module wants to do. Registration is managed through two appropriately named functions: *nf_register_hook(struct nc_hook_ops*)* and *nf_unregister_hook(struct nc_hook_ops*)*. The latter function removes the hook. It is important to unload the module, as failure to do so could cause fairly messy kernel errors. The crystallization point for registering a hook is the *nc_hook_ops* structure:

```
struct nf_hook_ops {
  struct list_head list;
  nf_hookfn *hook;
  struct module *owner;
  int pf;
  int hooknum;
  int priority;
};
```

This structure provides all the information the kernel needs to deploy the hook.

The first entry in the structure is for management in a linked list. The hook function itself is stored as a pointer in *hook*. The Linux kernel uses this hook function pointer later to call the kernel module. The *owner* entry stores the corresponding module; this entry is also found in numerous other kernel structures.

Choosing a Protocol

The last three entries in the *nc_hook_ops* structure are of particular interest: *pf* specifies the protocol family (OSI network layer) the hook is interested in. *linux/socket.h* contains a list of possible *pf* values.

The *PF_INET* value is reserved for Ipv4, because the Ipv4 protocol system is the network protocol used by the Singwall module. The *hooknum* value specifies the position where the hook latches in. Hooks are numbered, and a macro assigns names to them. The possible values for IPv4 are listed in *linux/netfilter_ipv4.h*.

We want the singing firewall to make a different noise for different incoming and outgoing packet types. Two hooks – *hooknum = NF_IP_LOCAL_OUT* for outgoing packets and *hooknum = NF_IP_LOCAL_IN* for incoming packets – will handle this.

The last entry in the *nf_hook_ops* structure specifies the priority with which the kernel should insert the new hook into the list of existing hooks. Netfilter processes hooks based on their priority. The *priority* values for IPv4 are also detailed in *linux/netfilter_ipv4.h*. We can assign a value of *priority = NF_IP_PRI_FIRST* for the Singwall to put the hook at the top of the list.

Arguments

A hook prototype, or the hook function to be more precise, is defined in *linux/netfilter.h*:

```
typedef unsigned int nf_hookfn(
  unsigned int hooknum,
  struct sk_buff **skb,
  const struct net_device *in,
  const struct net_device *out,
  int (*okfn)(struct sk_buff *)
);
```

The critical argument, and in fact the only relevant argument for Singwall, is second on the list: *skb*. This argument is a pointer to a *sk_buff* structure (a socket buffer). The kernel uses this critical network data structure to transfer data efficiently between the individual network layers.

Packet Analysis

The packet analysis we want the singing firewall to perform can be handled using the socket buffer. The complex *sk_buff* structure is provided by *linux/skbuff.h*. This structure uses a *data* pointer to point to the network data and extracts a variety of management information. In the case of TCP data, the firewall first extracts the TCP header, which in turn gives the firewall the ability to extract the port number.

The hook's return values specify what kind of post-processing actions to apply to the packets. The return values, which are listed in *linux/netfilter.h*, include settings such as *NF_DROP* and *NF_ACCEPT*. *NF_DROP* drops the packet after processing, whereas *NF_ACCEPT* lets the packet through. This arrangement makes it quite easy to write a minimal firewall. If the filter does not like the looks of the packet it has just inspected, the hook returns *NF_DROP*, and the kernel takes care of the dirty work. The singing firewall always returns *NF_ACCEPT*, as it simply analyzes the network data without filtering them.

Opus 1

As an overture, let's ask Singwall to tag UDP, TCP, and ICMP packets with tones of different pitches. In case of TCP, a second tone will indicate the port, and thus will tell us the underlying service based on analysis of the TCP header.

The *tcphdr* structure stores the header data. To extract the port number, Singwall reads the *dest* field. The value first has to be converted to the machine's byte order. The *ntohs(unsigned short int netshort)* function handles this, returning the port number as an integer. The

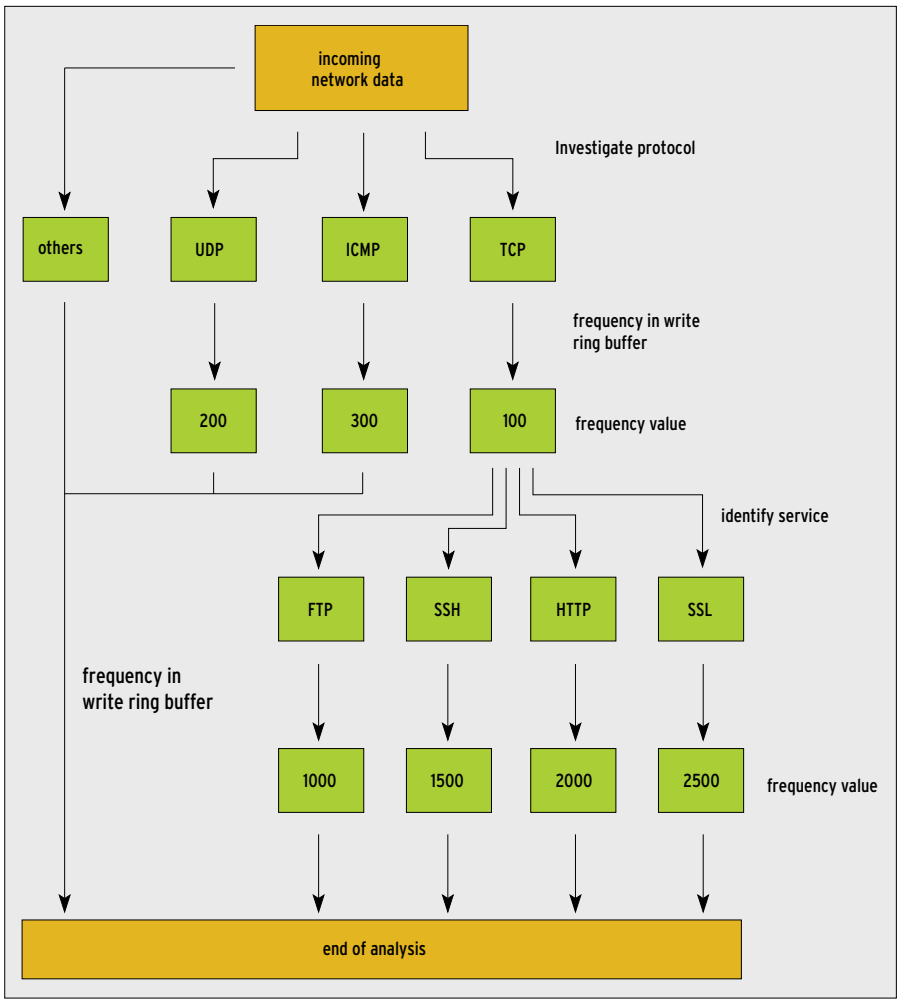

Figure 1: The kernel uses a Netfilter hook to pass incoming packets to Singwall. The module analyzes the protocol, calculates a frequency, and writes the results to a ring buffer.

packet analysis process is shown in Figure 1.

A problem at the hardware/human interface prevents a straightforward implementation. Network traffic is typically so fast that the human ear would not be able to register the individual beeps. To slow things down, Singwall implements a ring buffer with a length of *RING_SIZE*. The hook *fucn* writes the tones to play to this buffer.

Two counters store the position of the next tone to be played (*tone_counter*) and the position of the next free entry (*tone_pos_counter*). If network packets arrive so quickly that the *tone_pos_counter* and *tone_counter* become unsynchronized by more than the *RANGE_SYNC* value, the code resynchronizes, setting *tone_pos_counter* to the value of *tone_counter*.

The ring buffer is shown in Figure 2a. This approach ignores many packets in case of heavy traffic, but as field tests show, ignoring packets does not necessarily mean you will miss hearing any connections.

The singing firewall implementation we have looked at thus far had no trouble following a normal web surfing session. Although *ping 127.0.0.1 -f* or mass downloads led to multiple resyncs, this did not spoil the sing-along session.

Singwall uses the PC speaker to output tones. The code tells the speaker to squawk at the required frequency (although we have had a few issues with mute speakers on newer hardware). We decided on a tone length of 20 milliseconds – this is a compromise between audibility and fast handling, which is important to avoid missing more packets than necessary.

Of course, we need to avoid a situation where the firewall is blocking the

Listing 1: Singwall

```
001 #include <linux/version.h>
002 #include <linux/module.h>
003 #include <linux/kernel.h>
004 #include <linux/netfilter.h>
005 #include <linux/netfilter_
    ipv4.h>
006 #include <linux/init.h>
007 #include <linux/tcp.h>
008 #include <asm/io.h>
009 #include <linux/inet.h>
010
011 #if LINUX_VERSION_CODE >=
    KERNEL_VERSION(2,6,16)
012 #include <linux/in.h>
013 #include <linux/ip.h>
014 #endif
015
016 #define RING_SIZE 500
017 #define SYNC_RANGE 100
018 MODULE_LICENSE("GPL");
019
020 static wait_queue_head_t wq;
021 static int thread_id;
022 static DECLARE_COMPLETION(on_
    exit);
023
024 static struct nf_hook_ops
    nfho_out,nfho_in;
025
026 u16 tone[RING_SIZE];
027 int tone_counter;
028 int tone_pos_counter;
029
030 void update_tone_pos_
    counter(void) {
031   if(tone_pos_counter<RING_
    SIZE-1)
    tone_pos_counter++;
032   else tone_pos_counter=0;
033 }
034
035 void update_tone_
    counter(void) {
036   if(tone_counter<RING_SIZE-
    1) tone_counter++;
037   else tone_counter=0;
038 }
039
040 void check_syncing(void) {
041   int counter;
042   if ((abs((tone_pos_counter
    - tone_counter+
    RING_SIZE)
043           % RING_SIZE))>SYNC_
    RANGE)
044   {
045     counter=tone_counter;
046
047     while (((abs((counter -
    tone_counter+
    RING_SIZE)
048                 % RING_
    SIZE))<=SYNC_RANGE))
049     {
050       tone[(counter%RING_
    SIZE)-1]=0;
051       counter++;
052       pr_debug(" %d
    ",(counter%RING_SIZE)-1);
053     }
054     pr_
    debug("SingingFirewall:
    Resyncing %d<---%d!\n",
055       tone_counter,tone_pos_
    counter);
056     tone_pos_counter=tone_
    counter;
057   }
058 }
059
060 static int thread_code(void
    *data) {

[...]

092 }
093
094 unsigned int hook_func(
095   unsigned int hooknum,
096   struct sk_buff **skb,
097   const struct net_device
    *in,
098   const struct net_device
    *out,
099   int (*okfn)(struct sk_buff
    *))
100 {
101   struct tcphdr *thead;
102   struct sk_buff *sk=*skb;
103   u16 port;
104   char check;
105
106   check=0;
107   if (sk->nh.iph->protocol ==
    IPPROTO_TCP)
```

network traffic while it outputs a 20 millisecond tone. A kernel thread that processes the ring buffer and outputs the individual tones provides the parallelism we need to ensure that the traffic will continue without interruption. The thread includes a *while()* loop that runs every 10 milliseconds. This example does this without locking between parallel actions – in the worst case, the firewall might miss a few tones, or trip up over its own toes, which is acceptable from a musical point of view.

The thread, and thus the infinite loop, both stop when the module is unloaded. Figure 2b shows how the hook function and the threads cooperate to handle the ring buffer.

Let the Music Play!

Listing 1 implements the concepts I just described. The module needs two include files on kernel version 2.6.16 or newer. The *if* block in Lines 11 through 14 takes care of including these additional files. The module then goes on to the define values for ring buffer management: *RING_SIZE* and *SYNC_RANGE*. These values can be modified to reflect your own requirements.

If the *DEBUG* macro is defined when you launch the compiler, the system will output messages in */var/log/messages*. Note that the module will log the values for *tone_counter* and *tone_pos_counter*, thus creating an enormous number of messages in the logfile. This log data is useful for debugging, as it shows you how the module synchronizes the ring buffer if it can't keep up with network traffic.

The module's *init* routine (in Line 140) first sets the two ring buffer counters to zero. *init* then goes on to launch the thread, including queues (Lines 147

Listing 1: Singwall

```
108     {tone[tone_pos_
    counter]=100; check=1;}
109   if (sk->nh.iph->protocol ==
    IPPROTO_UDP)
110     {tone[tone_pos_
    counter]=200; check=1;}
111   if (sk->nh.iph->protocol ==
    IPPROTO_ICMP)
112     {tone[tone_pos_
    counter]=300; check=1;}
113   if (!check) return NF_
    ACCEPT;
114
115   update_tone_pos_counter();
116   check_syncing();
117   pr_debug("Tone Pos Counter:
    %d\n",
    tone_pos_counter);
118
119   check=0;
120   if (sk->nh.iph->protocol ==
    IPPROTO_TCP) {
121     thead=(struct tcphdr *)
    (sk->data + (sk->nh.iph->ihl
    * 4));
122     port=ntohs(thead->dest);
123     switch( port )
124     {
125       case 21 : {tone[tone_
    pos_counter]=1000;check=1;bre
    ak;}
126       case 22 : {tone[tone_
    pos_counter]=1500;check=1;bre
    ak;}
127       case 80 : {tone[tone_
    pos_counter]=2000;check=1;bre
    ak;}
128       case 443: {tone[tone_
    pos_counter]=2500;check=1;bre
    ak;}
129     }
130
131     if (check) {
132       update_tone_pos_
    counter();
133       check_syncing();
134       pr_debug("Tone Pos
    Counter: %d\n",
    tone_pos_counter);
135     }
136   }
137   return NF_ACCEPT;
138 }
139
140 int init_module() {
141   int counter;
142   for (counter=0;
    counter<=RING_SIZE-1;
    counter++)
143     tone[counter]=0;
144   tone_counter=0;
145   tone_pos_counter=0;
146
147   init_waitqueue_head(&wq);
148   thread_id=kernel_
    thread(thread_code, NULL,
    CLONE_KERNEL);
149   if(thread_id==0) return -
    EIO;
150
151   nfho_out.hook     = hook_
    func;
152   nfho_out.hooknum  = NF_IP_
    LOCAL_OUT;
153   nfho_out.pf       = PF_INET;
154   nfho_out.priority = NF_IP_
    PRI_FIRST;
155
156   nfho_in.hook      = hook_
    func;
157   nfho_in.hooknum   = NF_IP_
    LOCAL_IN;
158   nfho_in.pf        = PF_INET;
159   nfho_in.priority  = NF_IP_
    PRI_FIRST;
160
161   nf_register_hook(&nfho_
    out);
162   nf_register_hook(&nfho_in);
163   return 0;
164 }
165
166 void cleanup_module() {
167     if(thread_id) kill_
    proc(thread_id, SIGTERM, 1);
168     wait_for_completion(&on_
    exit);
169     nf_unregister_hook(&nfho_
    in);
170     nf_unregister_hook(&nfho_
    out);
171     outb(inb_p(0x61) & 0xFC,
    0x61);
172 }
```

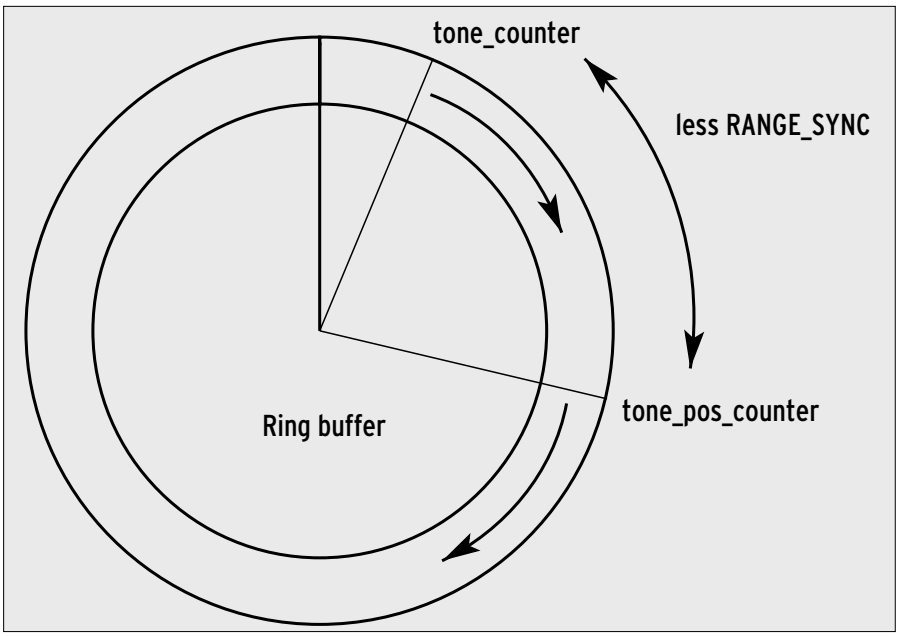

Figure 2a: The hook function stores the pending tones in a ring buffer. Before the buffer overflows, resync drops some data.

through 149). Finally, the hook is registered for incoming and outgoing packets and IPv4. The cleanup routine in Lines 166 reverts these steps, cleanly removing the module from the kernel. Just to be on the safe side, Line 171 contains a special command that disables the PC speaker.

Protocol Checker

The hook function (Lines 94 through 138) checks the protocol, based on the *sk->nh.iph->protocol* entry in the socket buffer passed to it (Lines 107 through 112). If Singwall recognizes the protocol, it sends a frequency to the ring buffer (Line 115).

If the packet is a TCP packet, Lines 120 through 136 generate a second frequency to match the port number. If you like, you can add more services to the port list.

The *update_tone_pos_counter()* (Line 30) and *update_tone_counter()* (Line 35) functions take care of updating the counters. The *check_syncing()* function (Line 40) synchronizes the two pointers. Using a function to do this may not be particularly elegant, but this solution works well, is stable, and clearly delegates responsibilities.

The makefile in Listing 2 helps to build the module in next to no time, and entering the *insmod singwall.ko* as root will load the module into the kernel. The singing firewall then immediately launches into its concert program, actuating the PC loudspeaker in harmony with the network traffic. As predicted, the pitch reflects the nature of the data.

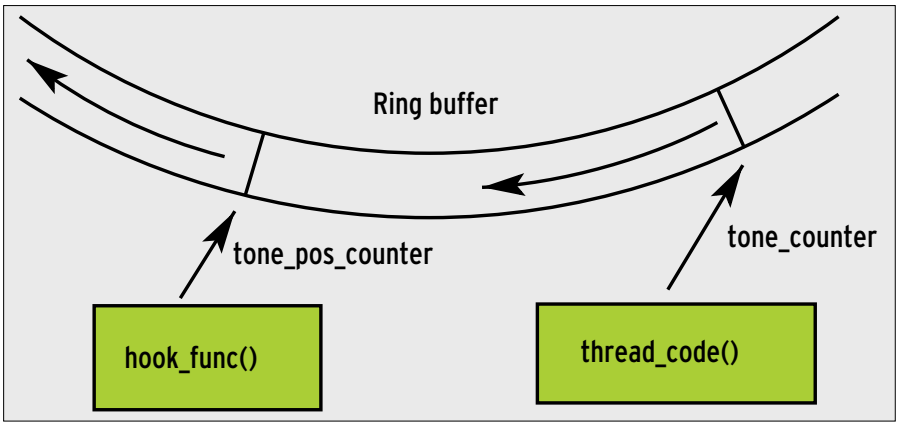

Figure 2b: While the hook function fills the ring buffer with entries, a parallel kernel thread reads the data and plays the tones on the PC speaker.

Listing 2: Makefile

```
01 01
02 #Debug messages?
03 02
04 #EXTRA_CFLAGS+=-DDEBUG
05 03
06 04
07 ifneq ($(KERNELRELEASE),)
08 05
09 obj-m       := singwall.o
10 06
11 else
12 07
13 KDIR    := /lib/modules/
   $(shell uname -r)/build
14 08
15 PWD     := $(shell pwd)
16 09
17 10
18 all:
19 11
20         $(MAKE) -C $(KDIR)
   SUBDIRS=$(PWD) modules
21 12
22 endif
```

It is quite easy to distinguish between encrypted and clear text web pages, for example. The tone for HTTPS connections is higher than the tone for unencrypted websites, and the difference is easy to detect, even if you have no musical training. You'll also find that it is easy to distinguish between SSH and FTP packets.

If you remove the comment tags in Line 2 of the makefile, you can build the module in debug mode. Running the module in debug mode will give you thousands of logfile entries – useful for troubleshooting, but no use at all in normal operations.

Enjoy the concert, and if you grow weary of the song coming from the PC speaker, just run *rmmod singwall* to mute the singer. ■

INFO

[1] Netfiter: *http://www.netfilter.org*

[2] Downloads: *ftp//ftp.linux-magazine.com/Magazine/Downloads/71/Singwall*