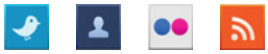

Roll Your Own Firewall with Netfilter

Oct 13, 2003 By Victor Castro (/user/801303)
in

How to create a micro-firewall with kernel modules and packet filtering.

Every self-respecting Linux guru should be familiar with firewalls and how to install and configure them. With this in mind, Linux gurus also should be curious about how firewalls function and how to build a firewall of his or her own. Explaining exactly these two things is the goal of this article. Here, we attempt to write a firewall in less than 60 lines of C code. As impossible as this may sound, it actually is quite simple to do using the power of Linux kernel modules and Netfilter.

Netfilter is a packet filtering subsystem in the Linux kernel stack and has been there since kernel 2.4.x. Netfilter's core consists of five hook functions declared in linux/netfilter_ipv4.h. Although these functions are for IPv4, they aren't much different from those used in the IPv6 counterpart. The hooks are used to analyze packets in various locations on the network stack. This situation is depicted below:

```
  [INPUT] - - - > [1] - - - > [ROUTE] - - - > [3] - - - > [4] - - - > [OUTPUT]
                              |                ^
                              |                |
                              |             [ROUTE]
                              v                |
                             [2]              [5]
                              |                ^
                              |                |
                              v                |
                          [INPUT*]        [OUTPUT*]
[1]  NF_IP_PRE_ROUTING
[2]  NF_IP_LOCAL_IN
[3]  NF_IP_FORWARD
[4]  NF_IP_POST_ROUTING
[5]  NF_IP_LOCAL_OUT
[*]  Network Stack
```

NF_IP_PRE_ROUTING is called right after the packet has been received. This is the hook we are most interested in for our micro-firewall. NF_IP_LOCAL_IN is used for packets that are destined for the network stack and thus has not been forwarded. NF_IP_FORWARD is for packets not addressed to us but that should be forwarded. NF_IP_POST_ROUTING is for packets that have been routed and are ready to leave, and NF_IP_LOCAL_OUT is for packets sent out from our own network stack. Each function has a chance to mangle or do what it wishes with the packets, but it eventually has to return a Netfilter code. Here are the codes that can be returned and what they mean:

- NF_ACCEPT: accept the packet (continue network stack trip)
- NF_DROP: drop the packet (don't continue trip)
- NF_REPEAT: repeat the hook function
- NF_STOLEN: hook steals the packet (don't continue trip)
- NF_QUEUE: queue the packet to userspace

After we write our hook function, we have to register its options with the nf_hook_ops struct located in linux/netfilter.h.

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

The first thing we see in the struct is the list_head struct, which is used to keep a linked list of hooks, but it's not necessary for our firewall. The nf_hookfn* struct member is the name of the hook function that we define. The pf integer member is used to identify the protocol family; it's PF_INET for IPv4. The next field is the hooknum int, and this is for the hook we want to use. The last field is the priority int. The priorities are specified in linux/netfilter_ipv4.h, but for our situation we want NF_IP_PRI_FIRST.

That's enough theory. No one has learned anything from theory alone, so let's learn by crawling into the trenches and writing our first module. Our module should block all traffic in and out of the network stack and should use the NF_IP_PRE_ROUTING and NF_IP_POST_ROUTING hooks. We start by defining our hook function and registering the nf_hook_ops structs in init_module(). Finally, we unregister the hooks in cleanup().

```
#define __KERNEL__
#define MODULE
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
static struct nf_hook_ops netfilter_ops_in; /* NF_IP_PRE_ROUTING */
static struct nf_hook_ops netfilter_ops_out; /* NF_IP_POST_ROUTING */
/* Function prototype in <linux/netfilter> */
unsigned int main_hook(unsigned int hooknum,
                  struct sk_buff **skb,
                  const struct net_device *in,
                  const struct net_device *out,
                  int (*okfn)(struct sk_buff*))
{
return NF_DROP; /* Drop ALL Packets */
}
int init_module()
{
        netfilter_ops_in.hook                   =       main_hook;
        netfilter_ops_in.pf                     =       PF_INET;
        netfilter_ops_in.hooknum                =       NF_IP_PRE_ROUTING;
        netfilter_ops_in.priority               =       NF_IP_PRI_FIRST;
        netfilter_ops_out.hook                  =       main_hook;
        netfilter_ops_out.pf                    =       PF_INET;
        netfilter_ops_out.hooknum               =       NF_IP_POST_ROUTING;
        netfilter_ops_out.priority              =       NF_IP_PRI_FIRST;
        nf_register_hook(&netfilter_ops_in); /* register NF_IP_PRE_ROUTING hook */
        nf_register_hook(&netfilter_ops_out); /* register NF_IP_POST_ROUTING hook */
return 0;
}
void cleanup()
{
nf_unregister_hook(&netfilter_ops_in); /*unregister NF_IP_PRE_ROUTING hook*/
nf_unregister_hook(&netfilter_ops_out); /*unregister NF_IP_POST_ROUTING hook*/
}
```

Let's walk through the code. We start with the regular #define and #include statements and declare our two nf_hook_ops structs, one for what comes in and one for what goes out. We then see our hook function, which passes a few important parameters. The first, hooknum, is a hooktype we already have covered. The second is a pointer to a pointer to a socket kernel buffer. The next two are netdevice pointers; we'll use these later to block and filter interfaces. The last parameter is a pointer to a function that takes in an sk_buff. With that in place, all we do in the hook function is drop all packets by returning **NF_DROP**.

Inside init_module(), we fill in the nf_hook_ops structs and then formally register the hooks with **nf_register_hook()**. In cleanup(), all we do is unregister the two hooks with **nf_unregister_hook()**. To get the module working, compile and load it with the

following commands, assuming the source file is called drop.c:

```
$>cc -c drop.c
$>insmod drop.o
```

Once the module is up and running, open Ethereal or an equivalent program and watch the packets *not* come in. To unload the module, issue the command **$>rmmod drop**.

Now we move on to our main project, called micro-firewall. The firewall is going to filter packets based on IP address, interface and TCP/UDP protocol and/or port. The firewall is a kernel module and is less than 60 lines of code in length. Let's look at the code first then review it:

```
#define __KERNEL__
#define MODULE
#include <linux/ip.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/netdevice.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
#include <linux/skbuff.h>
#include <linux/udp.h>
static struct nf_hook_ops netfilter_ops;
static unsigned char *ip_address = "\xC0\xA8\x00\x01";
static char *interface = "lo";
unsigned char *port = "\x00\x17";
struct sk_buff *sock_buff;
struct udphdr *udp_header;
unsigned int main_hook(unsigned int hooknum,
                  struct sk_buff **skb,
                  const struct net_device *in,
                  const struct net_device *out,
                  int (*okfn)(struct sk_buff*))
{
  if(strcmp(in->name,interface) == 0){ return NF_DROP; }

  sock_buff = *skb;
  if(!sock_buff){ return NF_ACCEPT; }
  if(!(sock_buff->nh.iph)){ return NF_ACCEPT; }
  if(sock_buff->nh.iph->saddr == *(unsigned int*)ip_address){ return NF_DROP; }


if(sock_buff->nh.iph->protocol != 17){ return NF_ACCEPT; }
udp_header = (struct udphdr *)(sock_buff->data + (sock_buff->nh.iph->ihl *4));
if((udp_header->dest) == *(unsigned short*)port){ return NF_DROP; }
return NF_ACCEPT;
}
int init_module()
{
        netfilter_ops.hook              =       main_hook;
        netfilter_ops.pf                =       PF_INET;
        netfilter_ops.hooknum           =       NF_IP_PRE_ROUTING;
        netfilter_ops.priority          =       NF_IP_PRI_FIRST;
        nf_register_hook(&netfilter_ops);

return 0;
}
void cleanup_module() { nf_unregister_hook(&netfilter_ops); }
```

We start with the #define and #include statements. Next, we declare an nf_hook_ops and an IP address (192.168.0.1) in network byte order. We also declare a char* called "lo" for the loopback interface, which we want to block. We also declare a char* for port 23, the telnet port. The last globals are a pointer to a socket kernel buffer and a pointer to a UDP header.

The hook function is where we do the real work. In our first statement, we compare the name of the device the packet came from to our char* interface. If the device is the loopback device, we return **NF_DROP**. In other words, we drop the packet. That's all that is involved with filtering by interface. We easily could have filtered packets from the Ethernet device by replacing <coe>lo with **etho** in the char* interface declaration.

Next, we filter by IP address and use the sk_buff to check for an IP address. We first check to see if we have a valid sk_buff, then we validate the IP packet, and finally we compare IP addresses.

Our last filtering technique is by protocol and/or port. Here we decide to filter by UDP port. First we check to see if we have a valid UDP packet. If we do, we copy the packet's UDP struct to our own. Finally, we compare the packet's UDP port with port 23 (telnet). If all else fails, the hook function returns **NF_ACCEPT** and the packet goes on its merry way through the network stack. Compile and load the firewall with the following commands:

```
$>cc -c firewall.c
$>insmod firewall.o
```

Once again, fire up Ethereal and see if the rules in the firewall holds. The packet filtering we did is straight-forward, and the possibilities are endless if you use your imagination. Unload the module with **$>rmmod firewall**, and things go back to normal.

We've taken a magical trip through the world of Netfilter, and what we got is our micro-firewall. Many possibilities for Netfilter exist, not only with firewalls but for many other useful network tools.

Victor Castro can be reached via his Web site (http://www.geocities.com/victorhugo83) .

Comments

problem with nf (/node/7184/print#comment-343264)
Submitted by fr4nkissimo (not verified) on Sep 24, 2009.

i have a problem, first i show you the code:

```
unsigned int hook_post_routing( unsigned int hooknum,
struct sk_buff **skb,
const struct net_device *in,
const struct net_device *out,
int (*okfn)(struct sk_buff *))
{
sock_buff = *skb;
printk(KERN_ALERT " - Sent new packet %p, %p\n", skb, *skb);

return NF_ACCEPT;}
```

i don't understand why the second value is always 00000000, shouldn't it change value at any time a packet goes?

Ran into Errors.... Please Help...!! (/node/7184/print#comment-328553)