Packet Filtering, Traffic Shaping and Firewalling#

(Derived from the NetBSD Documentation)

Packet Filters#

In principle there are two pseudo devices involved with packet filtering. *pf* is involved in filtering network traffic and *bpf* is an interface to capture and access raw network traffic. These will be discussed briefly from the point of view of their attachment to rest of the stack.

pf is implemented using pfil hooks, while *bpf* is implemented as a tap in all the network drivers.

Packet Filter Interface#

pfil is a purely in-stack interface to support packet filtering hooks. Packet filters can register hooks which should be called when packet processing is taking place; in essence *pfil* is a list of callbacks for certain events. In addition to being able to register a filter for incoming and outgoing packets, *pfil* provides support for interface attach/detach and interface change notifications. These modules are called Loadable Shared Modules (lsm) in Neutrino nomenclature or Loadable Kernel Modules (lkm) in BSD nomenclature.

There are two levels of registration required with io-pkt. The first allows the user supplied module to connect into the io-pkt framework and access the stack infrastructure. The second is the standard NetBSD mechanism for registering functions with the appropriate layer that sends / receives packets.

In Neutrino, shared modules are dynamically loaded into the stack. This can be done by specifying them on the command line when io-pkt is started or they can be subsequently added to an existing io-pkt process by using the "mount" command.

The application module must include an initial module entry point defined as

```
#include "sys/io-pkt.h"
#include "nw_datastruct.h"

int mod_entry(void *dll_hdl, struct _iopkt_self *iopkt, char *options)
{

}
```

The calling parameters to the entry function are:

void *dll_hdl: An opaque pointer which identifies the shared module within io-pkt.

struct _iopkt_self *iopkt: A structure used by the stack to reference it's own internals

options: The options string passed by the user to be parsed by this module.

Note that the include files are not installed as OS header files and you must include them from the relevant place in the networking source tree.

This is followed by the registration structure which the stack will look for after "dlopen"-ing the module to retrieve the entry point.

```
struct _iopkt_lsm_entry IOPKT_LSM_ENTRY_SYM(mod) =
  IOPKT_LSM_ENTRY_SYM_INIT(mod_entry);
```

This entry point registration is used by all shared modules, regardless of which layer the remainder of the code is going to hook into.

For registering with the pfil layer, the following functions are used

```
#include <sys/param.h>
#include <sys/mbuf.h>
#include <net/if.h>
#include <net/pfil.h>


struct pfil_head *
pfil_head_get(int af, u_long dlt);

struct packet_filter_hook *
pfil_hook_get(int dir, struct pfil_head *ph);

int
pfil_add_hook(int (*func)(), void *arg, int flags, struct pfil_head *ph);

int
pfil_remove_hook(int (*func)(), void *arg, int flags, struct pfil_head *ph);

int
(*func)(void *arg, struct mbuf **mp, struct ifnet *, int dir);
```

head_get returns the start of the appropriate pfil hook list used for the hook functions. "af" can be either "PFIL_TYPE_AF" (for an address family hook) or "PFIL_TYPE_IFNET" (for an interface hook).

If specifying PFIL_TYPE_AF, the Data Link Type (dlt) entry is a protocol family. The current implementation has filtering points for only AF_INET (IPv4) or AF_INET6 (IPv6).

When using the interface hook (PFIL_TYPE_IFNET), dlt is a pointer to a network interface structure. All hooks attached in this case will be in reference to the specified network interface.

Once the appropriate list head is selected, the pfil_add_hook routine can then be used to add a hook to the filter list.

pfil_add_hook takes a filter hook function, an opaque pointer (which is passed into the user supplied filter "arg" function), a flags value (described below) and the associated list head returned by pfil_head_get.

The flags value indicates when the hook function should be called and may be one of:

- PFIL_IN call me on incoming packets
- PFIL_OUT call me on outgoing packets
- PFIL_ALL call me on all of the above

When a filter is invoked, the packet appears just as if it came off the wire. That is, all protocol fields are in network byte order. The filter returns a non-zero value if the packet processing is to stop, or 0 if the processing is to continue.

For interface hooks, the flags parameter may be:

- PFIL_IFADDR call me on interface reconfig (mbuf ** is ioctl #)
- PFIL_IFNET call me on interface attach/detach (mbuf ** is either PFIL_IFNET_ATTACH or PFIL_IFNET_DETACH)

Following is an example of what a simple pfil hook would look like. It shows when an interface is attached or detached. Upon a detach ("ifconfig if destroy"), the filter is unloaded.

```
#include <sys/types.h>
#include <errno.h>
#include <sys/param.h>
#include <sys/conf.h>
#include <sys/socket.h>
#include <sys/mbuf.h>
#include <net/if.h>
#include <net/pfil.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include "sys/io-pkt.h"
#include "nw_datastruct.h"


static int in_bytes = 0;
static int out_bytes = 0;


static int input_hook(void *arg, struct mbuf **m, struct ifnet *ifp, int dir)
{
   in_bytes += (*m)->m_len;
   return 0;
}

static int output_hook(void *arg, struct mbuf **m, struct ifnet *ifp, int dir)
{
   out_bytes += (*m)->m_len;
   return 0;
}


static int deinit_module(void);
static int iface_hook(void *arg, struct mbuf **m, struct ifnet *ifp, int dir)
{
   printf("Iface hook called ... ");
   if ( (int)m == PFIL_IFNET_ATTACH) {
      printf("Interface attached\n");
      printf("%d bytes in, %d bytes out\n", in_bytes, out_bytes);
   } else if ((int)m == PFIL_IFNET_DETACH) {
      printf("Interface detached\n");
      printf("%d bytes in, %d bytes out\n", in_bytes, out_bytes);
      deinit_module();
   }
   return 0;
}

static int ifacecfg_hook(void *arg, struct mbuf **m, struct ifnet *ifp, int dir)
{

   printf("Iface cfg hook called with 0x%08X\n", (int)(m));

   return 0;
}
```

```
static int deinit_module(void)
{
   struct pfil_head *pfh_inet;

   pfh_inet = pfil_head_get(PFIL_TYPE_AF, AF_INET);
   if (pfh_inet == NULL) {
      return ESRCH;
   }
   pfil_remove_hook(input_hook, NULL, PFIL_IN | PFIL_WAITOK, pfh_inet);
   pfil_remove_hook(output_hook, NULL, PFIL_OUT | PFIL_WAITOK, pfh_inet);

   pfh_inet = pfil_head_get(PFIL_TYPE_IFNET, 0);
   if (pfh_inet == NULL) {
      return ESRCH;
   }

   pfil_remove_hook(ifacecfg_hook, NULL, PFIL_IFNET, pfh_inet);

   pfil_remove_hook(iface_hook, NULL, PFIL_IFNET | PFIL_WAITOK, pfh_inet);
   printf("Unloaded pfil hook\n" );

   return 0;
}



int pfil_entry(void *dll_hdl, struct _iopkt_self *iopkt, char *options)
{
   struct pfil_head *pfh_inet;

   pfh_inet = pfil_head_get(PFIL_TYPE_AF, AF_INET);
   if (pfh_inet == NULL) {
      return ESRCH;
   }
   pfil_add_hook(input_hook, NULL, PFIL_IN | PFIL_WAITOK, pfh_inet);
   pfil_add_hook(output_hook, NULL, PFIL_OUT | PFIL_WAITOK, pfh_inet);

   pfh_inet = pfil_head_get(PFIL_TYPE_IFNET,0);
   if (pfh_inet == NULL) {
      return ESRCH;
   }

   pfil_add_hook(iface_hook, NULL, PFIL_IFNET, pfh_inet);
   pfil_add_hook(ifacecfg_hook, NULL, PFIL_IFADDR, pfh_inet);
   printf("Loaded pfil hook\n" );

   return 0;
}

struct _iopkt_lsm_entry IOPKT_LSM_ENTRY_SYM(pfil) =
  IOPKT_LSM_ENTRY_SYM_INIT(pfil_entry);
```

Can I use pfil hooks to implement my io-net filter?#

pfil_hook is the recommended way to re-write your io-net filter to work in io-pkt.

Firstly, change your entry point function to remove the second arg (dispatch_t) and change the third option to be struct _iopkt_self.

Remove the io_net_dll_entry_t and replace it with the appropriate version of:

```
struct _iopkt_lsm_entry IOPKT_LSM_ENTRY_SYM(mod) =
  IOPKT_LSM_ENTRY_SYM_INIT(mod_entry);
```

"rx_up", "rx_down", "tx_done", "shutdown1", "shutdown2" functions that are part of the io-net API.

The rx_up / rx_down functions are essentially pfil_hook entries with a flag of PFIL_IN or PFIL_OUT respectively. Information about the interface that the packet was received on is contained in the ifp pointer (defined in usr/include/net/if.h. For example, the external interface name is in ifp->if_xname. This can be used to determine where the packet came from). The "cell / endpoint / iface" parameters are essentially wrapped up in the ifp pointer.

There are no advertisement packets used within io-pkt. For information about interfaces being added / removed / reconfigured, an interface hook can be added as described in the sample code above.

Buffering in io-pkt is handled using a different structure than io-net. io-net uses "npkt_t" buffers and io-pkt uses the standard "mbuf" buffers. io-net code will have to be modified to deal with the different buffer format. mbuf usage is documented in many places on the web. The header file which covers the io-pkt mbuf support is in / usr/include/sys/mbuf.h.

The shutdown1 / shutdown2 routines are somewhat similar to the "remove_hook" options above in which the filtering functions are removed from the processing stream. Typically, a filter requiring interaction with the user (e.g. read / write / umount) would export a resource manager interface to provide a mechanism for indicating that the filter has to be removed. pfil_remove_hook calls can then be used after clean up to remove the functions from the data path.

There isn't an equivalent of tx_done for pfil. In io-net, buffers are allocated by endpoints (e.g. a driver or a protocol) and therefore an endpoint specific buffer freeing function needs to be called to release the buffer back to the endpoint's buffer pool. In io-pkt, all buffer allocation is handled explicitly by the stack middleware. This means that any element requiring a buffer goes directly to the middleware to get it and anything freeing a buffer (e.g. the driver) puts it directly back in to the middleware buffer pools. This makes things easier to deal with since you now no longer have to track and manage your own buffer pools as an endpoint. As soon as the code is finished with the buffer, it simply performs an m_freem of the buffer which places it back in the general pool.

There is one downside to the global buffer implementation. You can NOT create a thread using pthread_create which allocates or frees a buffer from the stack since the thread has to modify internal stack stuctures. The locking implemented within io-pkt is optimized to reduce thread context switch times and this means that "non-stack" threads can't lock protect these structures. Instead, the stack provides it's own thread creation function (defined in trunk/sys/nw_thread.h)

```
int nw_pthread_create(pthread_t *tidp, pthread_attr_t *attrp,
    void *(*funcp)(void *), void *arg,
    int flags, int (*init_func)(void *), void *init_arg);
```

The first four threads are the standard arguments to pthread_create. The last three are io-pkt specific. A fairly simplistic example of how to use this function can be found in net/ppp_tty.c.

In terms of transmitting packets, the ifp interface structure contains the if_output function that can be used to transmit a "user-built" ethernet packet. The if_output maps to ether_output for an ethernet interface. This function queues the mbuf to the driver queue for sending and subsequently calls if_start to transmit the packet on the queue. The preliminary queuing is implemented to allow traffic shaping to take place on the interface.

pf and Firewalls / NAT#

pf provides roughly the same functionality as "ipfilter": another filtering and NAT suite that also uses the pfil hooks. The portion of pf that loads into io-pkt is found in sys/dist/pf/net. The source for the accompanying userland utilities is located under dist/pf. Some relevant manual pages describing pf are pf(4), pf.conf(5) and pfctl(8). These can be found in our tree under the aforementioned locations or http://man.netbsd.org.

pf is started using the *pfctl* utility which issues a DIOCSTART ioctl. This causes pf to call pf_pfil_attach which runs the necessary pfil attachment routines. The key routines after this are pf_test and pf_test6 which are called for IPv4 and IPv6 packets respectively. These functions test which packets should be sent, received or dropped. The packet filter hooks, and therefore the whole of *pf*, are disabled with the DIOCSTOP ioctl, usually issued with *pfctl -d*.

An excellent reference for using PF is provided with the OpenBSD documentation here ftp://ftp3.usa.openbsd.org/pub/OpenBSD/doc/pf-faq.pdf. Certain portions of the document (related to packet queueing, CARP and others) don't apply to our stack (yet), but the general configuration information is relevant. This document covers both firewalling and NAT configurations that you can apply using pf. Note that while the BSD docs refer to "ioctl", our implementation requires that you use "ioctl_socket" instead.

Berkeley Packet Filter#

The Berkeley Packet Filter (BPF) (sys/net/bpf.c) provides link layer access to data available on the network through interfaces attached to the system. BPF is used by opening a device node, */dev/bpf* and issuing ioctl's to control the operation of the device. A popular example of a tool using BPF is *tcpdump*.

The device */dev/bpf* is a cloning device, meaning it can be opened multiple times. It is in principle similar to a cloning interface, except BPF provides no network interface, only a method to open the same device multiple times.

To capture network traffic, a BPF device must be attached to an interface. The traffic on this interface is then passed to BPF for evaluation. For attaching an interface to an open BPF device, the ioctl BIOCSETIF is used. The interface is identified by passing a *struct ifreq*, which contains the interface name in ASCII encoding. This is used to find the interface from the kernel tables. BPF registers itself to the interfaces *struct ifnet field if_bpf* to inform the system that it is interested about traffic on this particular interface. The listener can also pass a set of filtering rules to capture only certain packets, for example ones matching a given host and port combination.

BPF captures packets by supplying a tapping interface, *bpf_tap*-functions, to link layer drivers and relying on the drivers to always pass packets to it. Drivers honor this request and commonly have code which, along both the input and output paths, does:

```
#if NBPFILTER > 0
   if (ifp->if_bpf)
      bpf_mtap(ifp->if_bpf, m0);
#endif
```

This passes the mbuf to the BPF for inspection. BPF inspects the data and decides if anyone listening to this particular interface is interested in it. The filter inspecting the data is highly optimized to minimize time spent inspecting each packet. If the filter matches, the packet is copied to await being read from the device.

The BPF tapping feature looks very much like the interfaces provided by pfil, so a valid question is are both required? Even though they provide similar services, their functionality is disjoint. The BPF mtap wants to access packets right off the wire without any alteration and possibly copy it for further use. Callers linking into pfil want to modify and possibly drop packets. The "pfil" interface is more analgous to io-net's filter interface.

BPF has quite a rich and complex syntax (e.g. http://www.rawether.net/support/bpfhelp.htm) and is a standard interface that is used by a lot of networking software (see also http://canmore.annwfn.net/freebsd/bpf.html). It should be your interface of first choice when packet interception / transmission is required. It will also be a slightly lower performance interface given that it does operate across process boundaries with filtered packets being copied before being passed outside of the stack domain and into the application domain. "tcpdump" / libpcap operate using the BPF interface to intercept and display packet activity. For those of you currently using something like the "nfm-nraw" interface in io-net, BPF provides the equivalent functionality, with some extra complexity involved in setting things up, but with much more versatility in configuration. Also note, you should use "ioctl_socket" instead of "ioctl" in your source. With the micro-kernel message passing architecture, ioctl calls which have pointers embedded in them need to be handled specially. "ioctl_socket" will default to "ioctl" for functionality which doesn't require special handling.