Linux Socket Filter

Analysis and Evaluation

Vasileios P. Kemerlis

Network Security Lab
Computer Science Department
Columbia University
New York, NY

05/06/2010

Outline

Outline

Packet Filter
What is it?

- Kernel-level mechanism (typically, but not always)
- Allows direct, **raw**, access to the network interface controller (NIC)
- Integral part of every modern operating system (OS)

Effective mechanism for "tapping" NICs

Packet Filter
Applications

- Historically packet filters facilitated user-space network protocol implementations
- Nowadays they are used mostly for debugging and monitoring

Examples

- Network intrusion detection and prevention (*Snort*, *Bro*)
- Traffic analysis (*tcpdump*, *wireshark*)
- Performance evaluation

Outline

CMU/Stanford Packet Filter
CSPF

- First kernel-level packet filter
- Used a special purpose *stack-based language* for describing arbitrary predicates (*i.e.,* packet selectors)
- Implemented in 4.3BSD UNIX (DEC VAX 11/790, PDP-11)

[1] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The Packet Filter: An Efficient Mechanism for User-level Network Code. In Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP), pages 39–51, Austin, TX, USA, November 1987.

The BSD Packet Filter

BPF

- BPF uses a new *register-based language*
- Maintains the flexibility and generality of CSPF
- Performs better on modern, RISC, machines
- Implemented in 4.3BSD Tahoe/Reno UNIX, 4.4BSD UNIX, HP-UX BSD variants, SunOS 3.5...
- Currently supported by every modern free BSD flavor (*e.g.,* FreeBSD, NetBSD, OpenBSD) as well as by Linux

[2] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In Proceedings of the USENIX Winter Conference, pages 259-269, San Diego, CA, USA, January 1993.

The Mach Packet Filter

MPF

- Kernel-level facility that efficiently dispatches incoming packets to multiple endpoints (*e.g.,* address spaces)
- Flexible and generic (5 additional instructions in BPF)
- Support for multiple active filters (scalable)
  - Exploits structural and logical similarity among different, but not identical filters
  - Identifies filters that have common “prefixes”
  - *Collapses* common filters into one
  - Uses associative matching for dispatching to the final communication endpoint
- Designed for Mach 3.0 (microkernel OS). No ports exist for other OSes, yet

[3] Masanobu Yuhara, Brian N. Bershad, Chris Maeda, and J. Eliot B. Moss. Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages. In Proceedings of the Winter USENIX Technical Conference (USENIX WTC), pages 153–165, San Francisco, CA, USA, January 1994.

Dynamic Packet Filters
DPF

- Kernel-level facility for rapid packet demultiplexing
- New, carefully-designed, *declarative language*
- Aggressive dynamic code generation
- Performance is equivalent, or can exceed, hand-coded demultiplexers
- Active filters are stored into a *prefix tree* data structure
- Designed for Aegis (exokernel OS). No ports exist for other OSes, yet

[4] Dawson R. Engler and M. Frans Kaashoek. DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation. In Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), pages 53–59, Standford, CA, USA, 1996.

The BSD Packet Filter+

BPF+

- MPF, DPF use local optimizations; they do not eliminate global common subexpressions
- Exploits data-flow algorithms for generalized optimization among filters
- Eliminates *redundant* predicates
- Allows for matching header fields against one another
- Can generate native code using just-in-time (JIT) compilation
- Relies upon a refined VM (more GPR, branch instructions can use register values)

[5] Andrew Begel, Steven McCanne, and Susan L. Graham. BPF+: Exploiting Global Data-flow Optimization in a Generalized Packet Filter Architecture. ACM SIGCOMM Computer Communication Review, 29(4):123–134, 1999.

x Packet Filter

xPF

- Need for more elaborate computational capabilities
- Engine for *executing* monitoring applications in kernel-space rather than a demultiplexing mechanism
- Persistent memory (per-filter)
- Support for backward branches
- xPF was implemented in OpenBSD

[6] Sotiris Ioannidis, Kostas G. Anagnostakis, John Ioannidis, and Angelos D. Keromytis. xPF: Packet Filtering for Low-cost Network Monitoring. In Proceedings of the IEEE Workshop on High-Performance Switching and Routing (HPSR), pages 121–126, Kobe, Hyogo, Japan, 2002.

Outline

Linux Socket Filter (LSF)

In a nutshell

- Kernel-level mechanism that allows **raw** access to the NIC
- Added to the Linux kernel with the 2.2 release
- Originally based on BPF (as everything else in the Linux networking stack)
- Currently uses the BPF language (for describing filters), but has a completely different internal architecture

BPF

Architectural overview

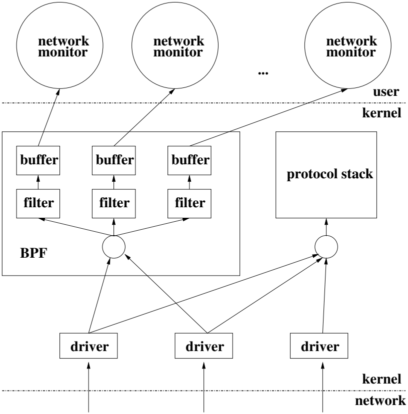


Figure: BPF architecture

```
ldh [12]
jeq #0x800 jt 2 jf 6
ld [26]
jeq #0xd0448b59 jt 12 jf 4
ld [30]
jeq #0xd0448b59 jt 12 jf 13
jeq #0x806 jt 8 jf 7
jeq #0x8035 jt 8 jf 13
ld [28]
jeq #0xd0448b59 jt 12 jf 10
ld [38]
jeq #0xd0448b59 jt 12 jf 13
ret #65535
ret #0
```

Figure: Example of a BPF program for "`host optimus`"

- *tcpdump* monitoring utility (v4.0.0) on Mac OS X 10.6
- `tcpdump -d -i en0 host optimus`

BPF
Usage

1. Open a special-purpose *character-device*, namely `/dev/bpf`*n*, for dealing with raw packets. *n* depends on how many other processes are using BPF and have filters installed
2. Associate the previous device with a network interface by using the `ioctl`(2) system call
3. Set various BPF parameters, such as the buffer size of the filter, and attach some BPF filters to the previous device to receive raw packets selectively. Again, this is done using the `ioctl`(2) system call
4. Read packets from the kernel, or send raw packets, by reading/writing to the corresponding file descriptor of `/dev/bpf` using `read`(2)/`write`(2) system calls

LSF

Usage & differences with BPF

- Utilizes **sockets** for passing/receiving packets to/from the kernel-space
- Filters are attached with the `setsockopt`(2) system call
- Usage in a nutshell:
  1. Create a special-purpose socket (*i.e.*, `PF_PACKET`)
  2. Attach a BPF program to the socket using the `setsockopt`(2) system call
  3. Set the network interface to *promiscuous* mode with `ioctl`(2) (*optionally*)
  4. Read packets from the kernel, or send raw packets, by reading/writing to the file descriptor of the socket using `recvfrom`(2)/`sendto`(2) system calls

Outline

```
static void
attach_filter(void) {
        struct sock_fprog filter;

        if ((sock = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL))) == -1)
                goto err;

        if (ioctl(sock, SIOCGIFFLAGS, &req) == -1)
                goto err;

        req.ifr_flags |= IFF_PROMISC;

        if (ioctl(sock, SIOCSIFFLAGS, &req) == -1)
                 goto err;

        filter.filter = bpf_code;
        filter.len = FT_LEN;
        if (setsockopt(sock,
                        SOL_SOCKET,
                        SO_ATTACH_FILTER,
                        &filter,
                        sizeof(filter)) == -1)
                  goto err;

         return;
err:
        (void)fprintf(stderr, "Error: %s\n", strerror(errno));
        exit(4);
}
```

Figure: LSF usage from user-space

Outline

Kernel Internals

What is next?

- LSF related system call traces (**kernel-space** only)
- Custom annotations with comments (`/* :: :: */`)
- “Irrelevant” functions are pushed towards the right side

socket(2) trace

```
sys_socketcall() { _copy_from_user(); audit_socketcall();
  sys_socket() {
    sock_create() { /* :: socket establishment :: */
      __sock_create() { security_socket_create() { cap_socket_create(); }
        sock_alloc() { /* :: socket struct allocation :: */
          new_inode() { /* :: all sockets are on sockfs :: */
            alloc_inode() {
              sock_alloc_inode() { kmem_cache_alloc(); __init_waitqueue_head(); }
              inode_init_always() { security_inode_alloc() { cap_inode_alloc_security(); }
              __mutex_init(); }
            }
            _raw_spin_lock(); _raw_spin_unlock(); } }
        packet_create() { /* :: PF_PACKET specific; resolved via packet family ops :: */
          capable() { security_capable() { cap_capable(); } }
          sk_alloc() { /* :: sock struct allocation :: */
            sk_prot_alloc() { __kmalloc() { get_slab(); memset(); }
              security_sk_alloc() { cap_sk_alloc_security(); }
            } __init_waitqueue_head(); }
          sock_init_data() { init_timer_key(); } __mutex_init();
          dev_add_pack() { /* :: register reception callback to the network stack :: */
            _raw_spin_lock_bh() {
              local_bh_disable() { __local_bh_disable(); }
            }
            _raw_spin_unlock_bh() { local_bh_enable_ip(); }
          }
          _raw_write_lock_bh() { local_bh_disable() { __local_bh_disable(); } }
          sock_prot_inuse_add();
          _raw_write_unlock_bh() { local_bh_enable_ip(); }
        }
```

socket(2) trace (cont'd)

```
        module_put();
        security_socket_post_create() { cap_socket_post_create(); }
      }
    }
    sock_map_fd() { /* :: install the socket descriptor in the process :: */
     sock_alloc_file() { /* :: file struct allocation :: */
       alloc_fd() { _raw_spin_lock(); expand_files(); _raw_spin_unlock(); }
       d_alloc() { kmem_cache_alloc(); memcpy(); _raw_spin_lock(); _raw_spin_unlock(); }
       d_instantiate() { _raw_spin_lock();
         __d_instantiate() { _raw_spin_lock(); _raw_spin_unlock();
           inotify_d_instantiate() {
             _raw_spin_lock(); _raw_spin_unlock(); } } _raw_spin_unlock();
         security_d_instantiate() { cap_d_instantiate(); } }
       alloc_file() {
         get_empty_filp() { kmem_cache_alloc() { memset(); }
           security_file_alloc() { cap_file_alloc_security(); } } }
     }
     fd_install() { _raw_spin_lock(); _raw_spin_unlock(); }
    }
  }
}
```

`socket`(2)
Summary

1. All network-related system calls are multiplexed via `sys_socketcall`(2); `sys_socket`() is invoked after demultiplexing in `sys_socketcall`() (*net/socket.c*)
2. In turn, `sys_socket`() calls `sock_create`() and `sock_map_fd`(). The latter does the housekeeping for installing the socket file descriptor into the process context
3. `sock_create`() invokes `sock_alloc`() and `packet_create`()
4. `sock_alloc`() allocates a socket structure – a new *inode* is allocated in **sockfs** and its parameters are filled
5. `packet_create`() allocates a sock structure and registers the corresponding packet handler with `dev_add_pack`()

socket(2)
Summary (cont'd)

- packet_create() is a protocol family specific (*i.e.*, PACKET) initialization function (*net/packet/af_packet.c*)
- Registered upon the setup of the protocol family by packet_init(), sock_register()
- Allocates a new sock structure, sets the "sock ops" for the corresponding protocol family, and most importantly, registers packet_rcv() to the network stack (*i.e.*, in ptype_base[] or ptype_all depending on the last parameter passed to socket(2))

setsockopt(2) trace

```
sys_setsockopt() {
  sockfd_lookup_light() { /* :: get the socket struct from the fd :: */ fget_light(); }
  security_socket_setsockopt() { cap_socket_setsockopt(); }
  sock_setsockopt() { /* :: generic handler :: */
      lock_sock_nested() {
      _raw_spin_lock_bh() { local_bh_disable() { __local_bh_disable(); } }
      _raw_spin_unlock();
      local_bh_enable();
    }
    _copy_from_user(); /* :: copy the filter length to kernel-space :: */
    sk_attach_filter() { /* :: attach the filter to the sock struct :: */
      sock_kmalloc() { __kmalloc() { get_slab(); } }
      _copy_from_user(); /* :: copy the filter instructions to kernel-space :: */
      sk_chk_filter(); /* :: filter validation :: */
      local_bh_disable() { __local_bh_disable(); }
      local_bh_enable();
    }
    release_sock() {
      _raw_spin_lock_bh() { local_bh_disable() { __local_bh_disable(); } }
      _raw_spin_unlock_bh() { local_bh_enable_ip(); }
    }
  }
}
```

`setsockopt`(2)

Summary

1. `sys_setsockopt`() is invoked after demultiplexing in `sys_socketcall`() (*net/socket.c*)
2. It resolves the socket structure associated with the file descriptor that was invoked with, does some locking, and then calls `sk_attach_filter`() (*net/core/filter.c*)
3. `sk_attach_filter`() allocates space for the filter, makes a copy from the user-space, and checks for errors by invoking `sk_chk_filter`()
4. If the filter is *syntactically* and *semantically* correct, then it is attached in the sock structure associated with the socket

NIC interrupt trace

```
pcnet32_interrupt() { /* :: IRQ handler :: */
  _raw_spin_lock();
  pcnet32_wio_read_csr(); pcnet32_wio_write_csr();
  pcnet32_wio_read_csr(); pcnet32_wio_write_csr();
  __napi_schedule(); /* :: schedule a NAPI call :: */
  _raw_spin_unlock() { preempt_schedule(); }
}

pcnet32_poll() { /* :: polling function registered to NAPI :: */
  dev_alloc_skb() { /* :: allocate a new skb; does not happen always :: */
    __alloc_skb() { kmem_cache_alloc(); __kmalloc_track_caller() { get_slab(); } }
  }
  skb_put(); /* :: make space :: */
  memcpy(); /* :: copy the received data to the skb :: */
  nommu_sync_single_for_device();
  eth_type_trans() { skb_pull(); }
  netif_receive_skb() { /* :: main reception point :: */ }
  _raw_spin_lock_irqsave();
  dev_kfree_skb_any() { dev_kfree_skb_irq() { raise_softirq_irqoff(); } }
  _raw_spin_unlock_irqrestore();
  _raw_spin_lock_irqsave();
  __napi_complete();
  pcnet32_wio_read_csr(); pcnet32_wio_write_csr(); pcnet32_wio_write_csr();
  _raw_spin_unlock_irqrestore() { /* :: standard boilerplate :: */ }
}
```

interrupt
Summary

1. Every NIC driver registers an IRQ handler upon the initialization of the device (*e.g.,* ifup, ifconfig) – in our case this is `pcnet32_interrupt`() (*drivers/net/pcnet32.c*)
2. `pcnet32_interrupt`() acknowledges the IRQ and schedules a NAPI call. The driver upon loading (*i.e.,* insmod, boot) registers a polling handler for the device to NAPI – `pcnet32_poll`()
3. NAPI invokes the polling function of the driver from a SoftIRQ context
4. `pcnet32_poll`() might allocate a new skb for holding the data received or not. In the latter scenario, the ring buffer is already mapped to skbs and the data have been “DMAed”
5. Finally, `pcnet32_poll`() calls `netif_receive_skb`() that does all the magic

netif_receive_skb() trace

```
netif_receive_skb() {
   packet_rcv() { /* :: drop (by the filter) :: */
     skb_push();
     local_bh_disable() { __local_bh_disable(); }
     sk_run_filter();
     local_bh_enable();
     consume_skb();
   }
   ip_rcv() { /* :: main IP reception point :: */ }
}
netif_receive_skb() {
   packet_rcv() { /* :: accept (by the filter) :: */
     skb_push();
     local_bh_disable() { __local_bh_disable(); }
     sk_run_filter();
     local_bh_enable();
     skb_clone() { kmem_cache_alloc(); __skb_clone() { __copy_skb_header(); } }
     kfree_skb();
     eth_header_parse();
     _raw_spin_lock();
     _raw_spin_unlock();
     sock_def_readable() { /* :: callback for processing data :: */}
   }
   ip_rcv() { /* :: main IP reception point :: */ }
}
```

netif_receive_skb()

Summary

1. netif_receive_skb() takes the skb with the received data and forwards it to the handlers (typically ip_rcv()) registered in the protocol stack – recall dev_add_pack()
2. packet_rcv() is the PACKET protocol family reception handler
3. It resolves the corresponding sock struct, runs the filter that the struct might have attached, and if the skb is accepted it appends a **clone** of the skb to the sock receive queue

recvfrom(2) trace

```
sys_recvfrom() {
  sockfd_lookup_light() { /* :: get the socket struct from the fd :: */ fget_light(); }
  sock_recvmsg() { security_socket_recvmsg() { cap_socket_recvmsg(); }
    packet_recvmsg() { /* :: resolve the PACKET recvmsg callback from proto_ops :: */
      skb_recv_datagram() { /* :: generic; pulls the skb from the receive queue :: */
        __skb_recv_datagram() { _raw_spin_lock_irqsave(); _raw_spin_unlock_irqrestore(); }
      }
      skb_copy_datagram_iovec() { /* :: scatter/gather I/O to user-space; data :: */
        memcpy_toiovec() { copy_to_user(); }
      }
      sock_recv_ts_and_drops(); /* :: timestamping :: */
      memcpy();
      skb_free_datagram() { /* :: dealloc :: */
        consume_skb() {
          __kfree_skb() {
            skb_release_head_state() { sock_rfree(); }
            skb_release_data() { kfree(); }
            kmem_cache_free();
          }
        }
      }
    }
  }
  move_addr_to_user() { /* :: copy the sockaddr struct to user-space :: */
    audit_sockaddr();
    copy_to_user();
  }
}
```

recvfrom(2)
Summary

1. sys_recvfrom() is invoked after demultiplexing in sys_socketcall() (*net/socket.c*)
2. It resolves the socket structure associated with the file descriptor that was invoked with, does some locking, and then calls sock_recvmsg()
3. sock_recvmsg() invokes the protocol specific “recvmsg” variant – packet_recvmsg()
4. packet_recvmsg() pulls the skb from the sock struct receive queue, copies the data in user-space using scatter/gather, fills the corresponding sockaddr struct, and deallocates the skb

sendto(2)
Summary

1. The "send path" is pretty straightforward
2. Similarly to every other socket call, `sys_sendto`() is invoked after demultiplexing in `sys_socketcall`() (*net/socket.c*)
3. It resolves the socket structure associated with the file descriptor that was invoked with, does some locking, and then calls `sock_sendmsg`() (*net/packet/af_packet.c*)
4. `sock_sendmsg`() invokes the protocol specific "sendmsg" variant – `packet_sendmsg`(), `packet_snd`()
5. `packet_snd`() allocates skbs using scatter/gather, checks the corresponding sockaddr struct, and finally invokes `dev_queue_xmit`()

Outline

Methodology
Micro-benchmarks

Filter attach

- start from `sys_setsockopt()`
- different filters sizes

interrupt / poll

- transfer 100MB using `nc`
- start from `packet_rcv()`
- different snaplen values

user-space delivery

- transfer 100MB using `nc`
- start from `sys_recvfrom()`
- different filters sizes

Outline

Testbed

Experimental setup

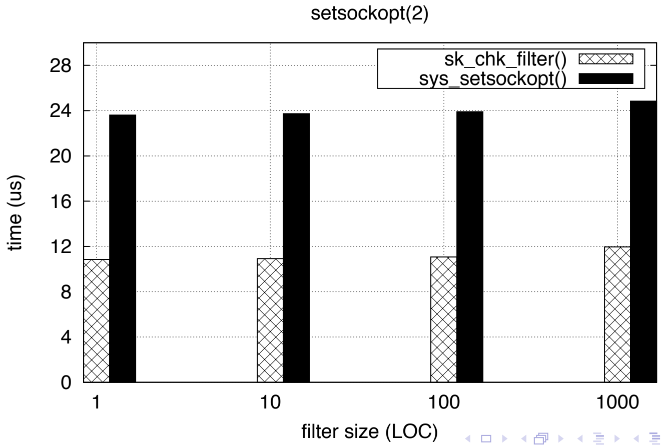
- Intel Core 2 Duo 2.6GHz, 4GB 667MHz DDR2 SDRAM
- GNU/Debian 5.0 (lenny)
- Vanilla 2.6.33.2 Linux kernel; heavily modified config so as to eliminate the driver bloat and enable various kernel-level debugging/tracing options
- Ftrace kernel tracer
- `nc`, `awk`, `gnuplot`, and lots of “glue” code in Bash/C

[7] Tim Bird. Measuring Function Duration with Ftrace. In Proceedings of the Ottawa Linux Symposium (OLS), pages 47–54, Montreal, Canada, July 2009.

Outline

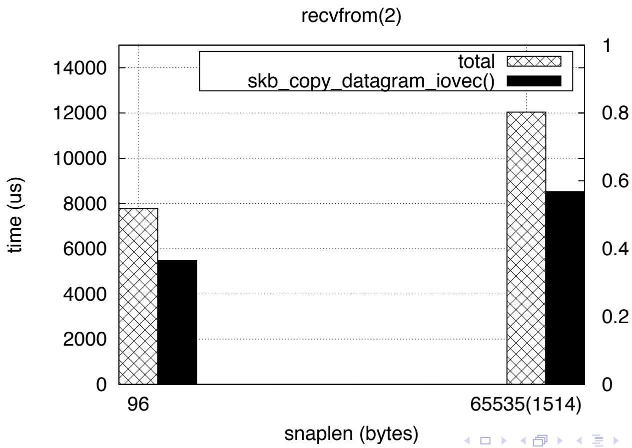
Results

`setsockopt`(2) micro-benchmarks



Results

`recvfrom`(2) micro-benchmarks



Results

interrupt/poll micro-benchmarks