Berkeley Packet Filter

Linux kernel TLV meetup, 28.02.2016

kerneltlv.com

Kfir Gollan

Agenda

- What is the **B**erkeley **P**acket **F**ilter?
- Writing BPF filters
- Debuging BPF
- Using BPF in user-space applications
- Advanced features of BPF

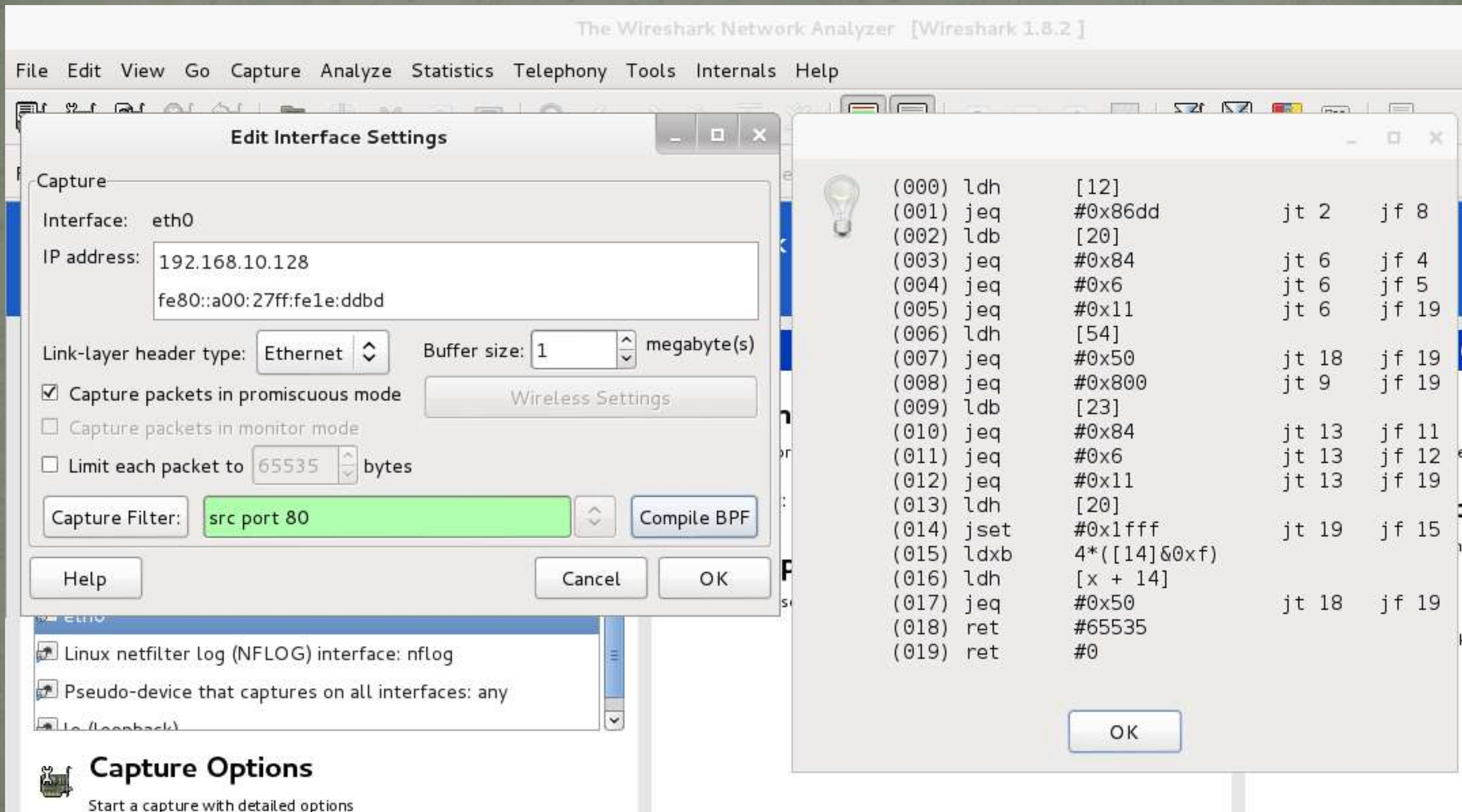
What is the Berkeley Packet Filter?

- BPF at its base is a way to perform fast packet filtering at the kernel level.
  - Filters are defined by user space
  - Filters are executed in the kernel
- Invented by Steven McCanne and Van Jacobson in 1990. First publication at December 1992.
- Support for BPF in Linux was added by Jay Schulist for the 2.5 development kernel.
  - Many features were added later on.
    For example JIT for BPF was added for the 3.0 kernel (2011)

Why do we need BPF?

- We want to be able to filter packets at the kernel level
  - Discard irrelevant packets at the kernel without copying them to user space.
  - The performance gains when using promiscuous mode are substantial.
- We want to change filters dynamically without recompiling the kernel or using a custom kernel module.
- We want the filters to be architecture independent.

BPF in wireshark



Writing BPF

BPF basics

- BPF defines a set of operations that can be performed on the filtered packet. Each operation gets its own opcode.
- BPF was designed to be protocol indepented, as such it treats the packets as raw buffers. It is up to the filter writer to parse the needed packet headers.
- BPF is based on three building blocks:
  - A: A 32 bit wide accumulator
  - X: A 32 bit wide index register
  - M[]: 16 x 32 bit wide misc registers aka “scratch memory”

BPF instruction types

- LOAD: copy a value into the accumulator or index register.
- STORE: copy either the accumulator or index register to the scratch memory.
- ALU: perform arithmetic or logic operation on the accumulator register.
- BRANCH: alter the flow of control.
- RETURN: terminate the filter and indicate what portion of the packet to save.
- MISC: various operations that doesn’t match the other types.

BPF instruction format

- Each instruction is represented by 64 bits.
  - opcode 16 bits, indicates the instruction type.
  - jt 8 bits, offset of the next instruction for true case ("if" block) – **J**ump **T**rue.
  - jf 8 bits, offset of the next instruction for false case ("else" block) – **J**ump **F**alse.
  - k 32 bits, generic data, used for various purposes.

opcode: 16	jt: 8	jf: 8
k: 32		

BPF instruction set

Instruction	Description	Instruction	Description
ld	Load word into A	add	A + <x>
ldh	Load half-word into A	sub	A - <x>
ldb	Load byte into A	mul	A * <x>
ldx	Load word into X	div	A / <x>
ldxb	Load byte into X	and	A & <x>
st	Store A into M[]	or	A | <x>
stx	Store X into M[]	xor	A ^ <x>
jmp	Jump to label	lsh	A << <x>
jeq	Jump on k == A	rsh	A >> <x>
jgt	Jump on k > A	ret	Return
jge	Jump on k >= A	tax	Copy A into X
jset	Jump on k & A	txa	Copy X into A

BPF addressing modes

Mode	Description
#k	The literal value stored in k.
#len	The length of the packet.
M [k]	The word at offset k in the scratch memory store.
[k]	The byte, half-word or word at byte offset k in the packet.
[x + k]	The byte, half-word or word at byte offset x+k in the packet.
L	Jump to label L
#k, lt. lf	Jump to lt if true, otherwise jump to l f
x	The index register
4 * ([k] & oxf)	Four times the value of the low four bits of the byte at the offset k in the packet

Example headers

- MAC header

Destination MAC	Source MAC	Type
6 bytes	6 bytes	2 bytes

- IP header

<table>
<tr><th>4-bit</th><th colspan="2">8-bit</th><th>16-bit</th><th colspan="2">32-bit</th></tr>
<tr><td>Ver.</td><td colspan="2">Header Length</td><td>Type of Service</td><td colspan="2">Total Length</td></tr>
<tr><td colspan="4">Identification</td><td>Flags</td><td>Offset</td></tr>
<tr><td colspan="2">Time To Live</td><td colspan="2">Protocol</td><td colspan="2">Checksum</td></tr>
<tr><td colspan="6">Source Address</td></tr>
<tr><td colspan="6">Destination Address</td></tr>
<tr><td colspan="6">Options and Padding</td></tr>
</table>

Example 1: All IP packets

- To catch all the IP packets over MAC we need to check the type field in the MAC header.

```
        ldh     [12]
        jeq     #ETHERTYPE_IP, L1, L2
L1:     ret     #-1
L2:     ret     #0
```

- Load half-word (2 bytes) from offset 12 of the packet (the type field) into A register.
- Check if A register is equal to #ETHERTYPE_IP
  - If true -> return #-1
  - If false -> return 0

Example 2: IP not from 128.3.112.X

```
        ldh     [12]                        ; A <= ether.type
        jeq     #ETHERTYPE_IP, L1, L3       ; A == #ETHERTYPE_IP ?
L1:     ld      [26]                        ; A = ip.src
        and     #0xffffff00                 ; A = A & 0xffffff00
        jeq     #0x80037000, L3, L2         ; A == 128.3.112.0
L2:     ret     #-1
L3:     ret     #0
```

- Check if the packet type is IP
- "Remove" the lower byte of the src IP
- Check if the src IP matches 128.3.112.X

Debuging BPF

bpf-asm

- A utility used to create bpf binary code (bpf "assembly" compiler").
  - Part of the mainline kernel. tools/net/bpf_asm.c
- Supports two output formats
  - c style output
    ```
    { 0x28,  0,  0,   0x0000000c  },
    { 0x15,  0,  1,   0x00000800  },
    { 0x06,  0,  0,   0xffffffff  },
    { 0x06,  0,  0,   0000000000  },
    ```
  - raw output
    ```
    4,40 0 0 12,21 0 1 2048,6 0 0 4294967295,6 0 0 0,
    ```

bpf-dbg

- A utility used to debug bpf filters
  - Part of the mainline kernel. tools/net/bpf_dbg.c
- Main features
  - pcap files as input for filters.
  - bpf-asm raw output format for bpf filter definition.
  - single-stepping through filters
  - breakpoints
  - internal status (A,X,M, PC)
  - disassemble raw bpf to bpf-asm

Debugging demo

Using BPF in user-space

linux/filter.h – sock_filter

```
struct sock_filter {                 /* Filter block */
        __u16   code;                /* Actual filter code */
        __u8    jt;                  /* Jump true */
        __u8    jf;                  /* Jump false */
        __u32   k;                   /* Generic multiuse field */
};
```

- Filter block is in fact a single BPF instruction.
- Used to pass a filter specifications to the kernel.

```
struct sock_filter code[] = {
        { 0x28, 0, 0, 0x0000000c },
        { 0x15, 0, 8, 0x000086dd },
        { 0x30, 0, 0, 0x00000014 },
        ...
};
```

linux/filter.h – sock_fprog

```
struct sock_fprog {                          /* Required for SO_ATTACH_FILTER. */
        unsigned short len;                  /* Number of filter blocks */
        struct sock_filter  *filter;         /* Filter blocks list */
};
```

- A parameter for setsockopt that allows to attach a filter to a socket.

```
struct sock_fprog bpf = {...};
setsockopt(int fd,
           SOL_SOCKET,
           SO_ATTACH_FILTER,
           &bpf.
           sizeof(bpf));
```

setsockopt flags

- SO_ATTACH_FILTER
  Attach a BPF filter to a socket.
  Note: only a single filter can be attached at a given time.
- SO_DETACH_FILTER
  Remove the currently attached filter from the socket.
- SO_LOCK_FILTER
  Lock a filter on a socket. This is useful for setting a filter and then dropping privileges.
  - Example: create a raw socket, apply a filter, lock it, drop CAP_NET_RAW

Choosing socket() flags correctly

- Choosing the correct flags to the socket is critical for making the filter work properly.
  - The filtered buffer will start in the wanted location in the net stack.
- Selecting the socket domain
  - AF_PACKET – filtering at L2 (e.g ethernet)
  - AF_INET – IPv4 filtering
  - AF_INET6 – IPv6 filtering
- Selecting the socket type
  - SOCK_RAW – raw filtering, no headers are handled by the kernel
  - SOCK_STREAM/SOCK_DGRAM etc – headers are handled by the kernel

PCAP

- libpcap – **P**acket **CAP**ture library
- Provides an easy to use api for packet filtering.
- Supports a high level filtering format which is converted to BPF.
  - pcap_compile – create a bpf filter
  - pcap_setfilter – attach a filter

```
ether dst [mac address]
dst net [ip address]
dst portrange [port1]-[port2]
```

- Look at man 7 pcap-filter for a detailed description.

tcpdump

- user-space packet sniffing program.
- Uses bpf for kernel level filtering (based on pcap)
  - Dump the generated bpf filter
    - -d bpf asm format
    - -dd c format
    - -ddd bpf raw format

```
$ sudo tcpdump  -d "ip and udp“
(000) ldh      [12]
(001) jeq      #0x800          jt 2    jf 5
(002) ldb      [23]
(003) jeq      #0x11           jt 4    jf 5
(004) ret      #65535
(005) ret      #0
```

BPF - Advanced features

JIT

- A just-in-time BPF instruction translation.
  - Note that the translation is performed when attaching the filter via bpf_jit_compile(..).
- BPF instructions are mapped directly to architecture depended instructions.
  - BPF registers are mapped to machine physical registers
- Provides a performance gain
  - About 50ns per packet for simple filters (E5540 @ 2.53GHz).
    The more complex the filter the better performance gains from JIT.
- Supported on x86,x86-64, powerpc. arm and more.

BPF verifier

- Each BPF filter is verified before attaching it to a socket.
  - This is critical, the filters come from userspace!
- The following rules are enforced
  - The filter must not contain references or jumps that are out of range.
  - The filter must contain only valid BPF opcodes.
  - The filter must end with RET opcode.
  - All jumps are forward – loops are not allowed!
- The verification is implemented at sk_chk_filter function in net/core/filter.c (until kernel 3.16), modified after adding seccomp.

BPF extensions

- The Linux kernel also has a couple of BPF extensions that are used along with the class of load instructions.
- The extensions are "overloading" the k argument with a negative offset + a particular extension offset.
- The result of such BPF extensions are loaded into A.

Instruction	Description	Instruction	Description
len	skb->len	queue	skb->queue_mapping
proto	skb->protocol	rxhash	skb->hash
type	skb->pkt_type	rand	Prandom_u32()
poff	Payload start offset	cpu	Executing cpu id
ifidx	skb->dev->ifindex	nla	Netlink attributes
mark	skb->mark	hatype	skb->dev->type

eBPF

- extended BPF is an internal mechanism that can be used only in kernel context (not from userspace!)
- eBPF adds a set of new features:
  - Increased number of registers 10 instead of 2.
  - Register width increased to 64 bit
  - Conditional jf/jt targets replaced with jt/fall-through
  - bpf_call instruction and register passing convention for zero overhead calls from/to other kernel functions.
- Originally designed to be a “restricted C” language that will be architecture independent and JITed in kernel context.
- Eventually used mostly for kernel tracing.

BPF – not only for networking

SecComp

- SECure COMPuting, or seccomp, is a security mechanism available in the linux kernel.
- It applies BPF filtering to syscalls
  - filter the syscall number & its parameters
- It can be used to limit the available syscalls
  - For example strict mode allows only read, write, _exit and sigreturn
- Uses the BPF filters, the filtered buffers are different.
- Look at man 2 seccomp for more details.

Questions?