Linux Socket Filtering aka Berkeley Packet Filter (BPF)

Notice

This file used to document the eBPF format and mechanisms even when not related to socket filtering. The ../bpf/index.rst has more details on eBPF.

Introduction

Linux Socket Filtering (LSF) is derived from the Berkeley Packet Filter. Though there are some distinct differences between the BSD and Linux Kernel filtering, but when we speak of BPF or LSF in Linux context, we mean the very same mechanism of filtering in the Linux kernel.

BPF allows a user-space program to attach a filter onto any socket and allow or disallow certain types of data to come through the socket. LSF follows exactly the same filter code structure as BSD's BPF, so referring to the BSD bpf.4 manpage is very helpful in creating filters.

On Linux, BPF is much simpler than on BSD. One does not have to worry about devices or anything like that. You simply create your filter code, send it to the kernel via the SO_ATTACH_FILTER option and if your filter code passes the kernel check on it, you then immediately begin filtering data on that socket.

You can also detach filters from your socket via the SO_DETACH_FILTER option. This will probably not be used much since when you close a socket that has a filter on it the filter is automagically removed. The other less common case may be adding a different filter on the same socket where you had another filter that is still running: the kernel takes care of removing the old one and placing your new one in its place, assuming your filter has passed the checks, otherwise if it fails the old filter will remain on that socket.

SO_LOCK_FILTER option allows to lock the filter attached to a socket. Once set, a filter cannot be removed or changed. This allows one process to setup a socket, attach a filter, lock it then drop privileges and be assured that the filter will be kept until the socket is closed.

The biggest user of this construct might be libpcap. Issuing a high-level filter command like *tcpdump -i em1 port 22* passes through the libpcap internal compiler that generates a structure that can eventually be loaded via SO_ATTACH_FILTER to the kernel. *tcpdump -i em1 port 22 -ddd* displays what is being placed into this structure.

Although we were only speaking about sockets here, BPF in Linux is used in many more places. There's xt_bpf for netfilter, cls_bpf in the kernel qdisc layer, SECCOMP-BPF (SECure COMPuting [1]), and lots of other places such as team driver, PTP code, etc where BPF is being used.

[1] Documentation/userspace-api/seccomp_filter.rst

Original BPF paper:

Steven McCanne and Van Jacobson. 1993. The BSD packet filter: a new architecture for user-level packet capture. In Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings (USENIX'93). USENIX Association, Berkeley, CA, USA, 2-2. [http://www.tcpdump.org/papers/bpf-usenix93.pdf]

Structure

User space applications include <linux/filter.h> which contains the following relevant structures:

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

Such a structure is assembled as an array of 4-tuples, that contains a code, jt, jf and k value. jt and jf are jump offsets and k a generic value to be used for a provided code:

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

For socket filtering, a pointer to this structure (as shown in follow-up example) is being passed to the kernel through setsockopt(2).

Example

```
#include <sys/socket.h>
#include <sys/types.h>
#include <arpa/inet.h>
```

```
#include <linux/if_ether.h>
/* ... */

/* From the example above: tcpdump -i em1 port 22 -dd */
struct sock_filter code[] = {
        { 0x28,  0,  0, 0x0000000c },
        { 0x15,  0,  8, 0x000086dd },
        { 0x30,  0,  0, 0x00000014 },
        { 0x15,  2,  0, 0x00000084 },
        { 0x15,  1,  0, 0x00000006 },
        { 0x15,  0, 17, 0x00000011 },
        { 0x28,  0,  0, 0x00000036 },
        { 0x15, 14,  0, 0x00000016 },
        { 0x28,  0,  0, 0x00000038 },
        { 0x15, 12, 13, 0x00000016 },
        { 0x15,  0, 12, 0x00000800 },
        { 0x30,  0,  0, 0x00000017 },
        { 0x15,  2,  0, 0x00000084 },
        { 0x15,  1,  0, 0x00000006 },
        { 0x15,  0,  8, 0x00000011 },
        { 0x28,  0,  0, 0x00000014 },
        { 0x45,  6,  0, 0x00001fff },
        { 0xb1,  0,  0, 0x0000000e },
        { 0x48,  0,  0, 0x0000000e },
        { 0x15,  2,  0, 0x00000016 },
        { 0x48,  0,  0, 0x00000010 },
        { 0x15,  0,  1, 0x00000016 },
        { 0x06,  0,  0, 0x0000ffff },
        { 0x06,  0,  0, 0x00000000 },
};

struct sock_fprog bpf = {
        .len = ARRAY_SIZE(code),
        .filter = code,
};

sock = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
if (sock < 0)
        /* ... bail out ... */

ret = setsockopt(sock, SOL_SOCKET, SO_ATTACH_FILTER, &bpf, sizeof(bpf));
if (ret < 0)
        /* ... bail out ... */

/* ... */
close(sock);
```

The above example code attaches a socket filter for a PF_PACKET socket in order to let all IPv4/IPv6 packets with port 22 pass. The rest will be dropped for this socket.

The setsockopt(2) call to SO_DETACH_FILTER doesn't need any arguments and SO_LOCK_FILTER for preventing the filter to be detached, takes an integer value with 0 or 1.

Note that socket filters are not restricted to PF_PACKET sockets only, but can also be used on other socket families.

Summary of system calls:

- setsockopt(sockfd, SOL_SOCKET, SO_ATTACH_FILTER, &val, sizeof(val));
- setsockopt(sockfd, SOL_SOCKET, SO_DETACH_FILTER, &val, sizeof(val));
- setsockopt(sockfd, SOL_SOCKET, SO_LOCK_FILTER, &val, sizeof(val));

Normally, most use cases for socket filtering on packet sockets will be covered by libpcap in high-level syntax, so as an application developer you should stick to that. libpcap wraps its own layer around all that.

Unless i) using/linking to libpcap is not an option, ii) the required BPF filters use Linux extensions that are not supported by libpcap's compiler, iii) a filter might be more complex and not cleanly implementable with libpcap's compiler, or iv) particular filter codes should be optimized differently than libpcap's internal compiler does; then in such cases writing such a filter "by hand" can be of an alternative. For example, xt_bpf and cls_bpf users might have requirements that could result in more complex filter code, or one that cannot be expressed with libpcap (e.g. different return codes for various code paths). Moreover, BPF JIT implementors may wish to manually write test cases and thus need low-level access to BPF code as well.

BPF engine and instruction set

Under tools/bpf/ there's a small helper tool called bpf_asm which can be used to write low-level filters for example scenarios mentioned in the previous section. Asm-like syntax mentioned here has been implemented in bpf_asm and will be used for further explanations (instead of dealing with less readable opcodes directly, principles are the same). The syntax is closely modelled after Steven McCanne's and Van Jacobson's BPF paper.

The BPF architecture consists of the following basic elements:

Element	Description
A	32 bit wide accumulator
X	32 bit wide X register
M[]	16 x 32 bit wide misc registers aka "scratch memory store", addressable from 0 to 15

A program, that is translated by bpf_asm into "opcodes" is an array that consists of the following elements (as already mentioned):

```
op:16, jt:8, jf:8, k:32
```

The element op is a 16 bit wide opcode that has a particular instruction encoded. jt and jf are two 8 bit wide jump targets, one for condition "jump if true", the other one "jump if false". Eventually, element k contains a miscellaneous argument that can be interpreted in different ways depending on the given instruction in op.

The instruction set consists of load, store, branch, alu, miscellaneous and return instructions that are also represented in bpf_asm syntax. This table lists all bpf_asm instructions available resp. what their underlying opcodes as defined in linux/filter.h stand for:

Instruction	Addressing mode	Description
ld	1, 2, 3, 4, 12	Load word into A
ldi	4	Load word into A
ldh	1, 2	Load half-word into A
ldb	1, 2	Load byte into A
ldx	3, 4, 5, 12	Load word into X
ldxi	4	Load word into X
ldxb	5	Load byte into X
st	3	Store A into M[]
stx	3	Store X into M[]
jmp	6	Jump to label
ja	6	Jump to label
jeq	7, 8, 9, 10	Jump on A == <x>
jneq	9, 10	Jump on A != <x>
jne	9, 10	Jump on A != <x>
jlt	9, 10	Jump on A < <x>
jle	9, 10	Jump on A <= <x>
jgt	7, 8, 9, 10	Jump on A > <x>
jge	7, 8, 9, 10	Jump on A >= <x>
jset	7, 8, 9, 10	Jump on A & <x>
add	0, 4	A + <x>
sub	0, 4	A - <x>
mul	0, 4	A * <x>
div	0, 4	A / <x>
mod	0, 4	A % <x>
neg		!A
and	0, 4	A & <x>
or	0, 4	A | <x>
xor	0, 4	A ^ <x>
lsh	0, 4	A << <x>
rsh	0, 4	A >> <x>
tax		Copy A into X
txa		Copy X into A
ret	4, 11	Return

The next table shows addressing formats from the 2nd column:

Addressing mode	Syntax	Description
0	x/%x	Register X
1	[k]	BHW at byte offset k in the packet
2	[x + k]	BHW at the offset X + k in the packet
3	M[k]	Word at offset k in M[]
4	#k	Literal value stored in k
5	4*([k]&0xf)	Lower nibble * 4 at byte offset k in the packet
6	L	Jump label L

Addressing mode	Syntax	Description
7	#k,Lt,Lf	Jump to Lt if true, otherwise jump to Lf
8	x/%x,Lt,Lf	Jump to Lt if true, otherwise jump to Lf
9	#k,Lt	Jump to Lt if predicate is true
10	x/%x,Lt	Jump to Lt if predicate is true
11	a/%a	Accumulator A
12	extension	BPF extension

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

Possible BPF extensions are shown in the following table:

Extension	Description
len	skb->len
proto	skb->protocol
type	skb->pkt_type
poff	Payload start offset
ifidx	skb->dev->ifindex
nla	Netlink attribute of type X with offset A
nlan	Nested Netlink attribute of type X with offset A
mark	skb->mark
queue	skb->queue_mapping
hatype	skb->dev->type
rxhash	skb->hash
cpu	raw_smp_processor_id()
vlan_tci	skb_vlan_tag_get(skb)
vlan_avail	skb_vlan_tag_present(skb)
vlan_tpid	skb->vlan_proto
rand	prandom_u32()

These extensions can also be prefixed with '#'. Examples for low-level BPF:

ARP packets:

```
ldh [12]
jne #0x806, drop
ret #-1
drop: ret #0
```

IPv4 TCP packets:

```
ldh [12]
jne #0x800, drop
ldb [23]
jneq #6, drop
ret #-1
drop: ret #0
```

icmp random packet sampling, 1 in 4:

```
ldh [12]
jne #0x800, drop
ldb [23]
jneq #1, drop
# get a random uint32 number
ld rand
mod #4
jneq #1, drop
ret #-1
drop: ret #0
```

SECCOMP filter example:

```
ld [4]                  /* offsetof(struct seccomp_data, arch) */
jne #0xc000003e, bad    /* AUDIT_ARCH_X86_64 */
ld [0]                  /* offsetof(struct seccomp_data, nr) */
jeq #15, good           /* __NR_rt_sigreturn */
jeq #231, good          /* __NR_exit_group */
jeq #60, good           /* __NR_exit */
jeq #0, good            /* __NR_read */
jeq #1, good            /* __NR_write */
jeq #5, good            /* __NR_fstat */
jeq #9, good            /* __NR_mmap */
jeq #14, good           /* __NR_rt_sigprocmask */
jeq #13, good           /* __NR_rt_sigaction */
jeq #35, good           /* __NR_nanosleep */
bad: ret #0             /* SECCOMP_RET_KILL_THREAD */
good: ret #0x7fff0000   /* SECCOMP_RET_ALLOW */
```

Examples for low-level BPF extension:

Packet for interface index 13:

```
ld ifidx
jneq #13, drop
ret #-1
drop: ret #0
```

(Accelerated) VLAN w/ id 10:

```
ld vlan_tci
jneq #10, drop
ret #-1
drop: ret #0
```

The above example code can be placed into a file (here called "foo"), and then be passed to the bpf_asm tool for generating opcodes, output that xt_bpf and cls_bpf understands and can directly be loaded with. Example with above ARP code:

```
$ ./bpf_asm foo
4,40 0 0 12,21 0 1 2054,6 0 0 4294967295,6 0 0 0,
```

In copy and paste C-like output:

```
$ ./bpf_asm -c foo
{ 0x28,  0,  0, 0x0000000c },
{ 0x15,  0,  1, 0x00000806 },
{ 0x06,  0,  0, 0xffffffff },
{ 0x06,  0,  0, 0000000000 },
```

In particular, as usage with xt_bpf or cls_bpf can result in more complex BPF filters that might not be obvious at first, it's good to test filters before attaching to a live system. For that purpose, there's a small tool called bpf_dbg under tools/bpf/ in the kernel source directory. This debugger allows for testing BPF filters against given pcap files, single stepping through the BPF code on the pcap's packets and to do BPF machine register dumps.

Starting bpf_dbg is trivial and just requires issuing:

```
# ./bpf_dbg
```

In case input and output do not equal stdin/stdout, bpf_dbg takes an alternative stdin source as a first argument, and an alternative stdout sink as a second one, e.g. *./bpf_dbg test_in.txt test_out.txt*.

Other than that, a particular libreadline configuration can be set via file "~/.bpf_dbg_init" and the command history is stored in the file "~/.bpf_dbg_history".

Interaction in bpf_dbg happens through a shell that also has auto-completion support (follow-up example commands starting with '>' denote bpf_dbg shell). The usual workflow would be to ...

- load bpf 6,40 0 0 12,21 0 3 2048,48 0 0 23,21 0 1 1,6 0 0 65535,6 0 0 0 Loads a BPF filter from standard output of bpf_asm, or transformed via e.g. `tcpdump -iem1 -ddd port 22 | tr '\n' ','`. Note that for JIT debugging (next section), this command creates a temporary socket and loads the BPF code into the kernel. Thus, this will also be useful for JIT developers.
- load pcap foo.pcap

  Loads standard tcpdump pcap file.
- run [<n>]

bpf passes:1 fails:9

Runs through all packets from a pcap to account how many passes and fails the filter will generate. A limit of packets to traverse can be given.

- disassemble:

  ```
  l0:     ldh [12]
  l1:     jeq #0x800, l2, l5
  l2:     ldb [23]
  l3:     jeq #0x1, l4, l5
  l4:     ret #0xffff
  l5:     ret #0
  ```

  Prints out BPF code disassembly.
- dump:

```
/* { op, jt, jf, k }, */
{ 0x28,  0,  0, 0x0000000c },
{ 0x15,  0,  3, 0x00000800 },
{ 0x30,  0,  0, 0x00000017 },
{ 0x15,  0,  1, 0x00000001 },
{ 0x06,  0,  0, 0x0000ffff },
{ 0x06,  0,  0, 0000000000 },
```

Prints out C-style BPF code dump.

- breakpoint 0:

  ```
  breakpoint at: l0:      ldh [12]
  ```

- breakpoint 1:

  ```
  breakpoint at: l1:      jeq #0x800, l2, l5
  ```

  ...

  Sets breakpoints at particular BPF instructions. Issuing a *run* command will walk through the pcap file continuing from the current packet and break when a breakpoint is being hit (another *run* will continue from the currently active breakpoint executing next instructions):

  - run:

    ```
    -- register dump --
    pc:       [0]                       <-- program counter
    code:     [40] jt[0] jf[0] k[12]    <-- plain BPF code of current instruction
    curr:     l0:   ldh [12]              <-- disassembly of current instruction
    A:        [00000000][0]             <-- content of A (hex, decimal)
    X:        [00000000][0]             <-- content of X (hex, decimal)
    M[0,15]:  [00000000][0]             <-- folded content of M (hex, decimal)
    -- packet dump --                   <-- Current packet from pcap (hex)
    len: 42
        0: 00 19 cb 55 55 a4 00 14 a4 43 78 69 08 06 00 01
    16: 08 00 06 04 00 01 00 14 a4 43 78 69 0a 3b 01 26
    32: 00 00 00 00 00 00 0a 3b 01 01
    (breakpoint)
    >
    ```

  - breakpoint:

    ```
    breakpoints: 0 1
    ```

    Prints currently set breakpoints.

- step [-<n>, +<n>]

  Performs single stepping through the BPF program from the current pc offset. Thus, on each step invocation, above register dump is issued. This can go forwards and backwards in time, a plain *step* will break on the next BPF instruction, thus +1. (No *run* needs to be issued here.)

- select <n>

  Selects a given packet from the pcap file to continue from. Thus, on the next *run* or *step*, the BPF program is being evaluated against the user pre-selected packet. Numbering starts just as in Wireshark with index 1.

- quit

  Exits bpf_dbg.

JIT compiler

The Linux kernel has a built-in BPF JIT compiler for x86_64, SPARC, PowerPC, ARM, ARM64, MIPS, RISC-V and s390 and can be enabled through CONFIG_BPF_JIT. The JIT compiler is transparently invoked for each attached filter from user space or for internal kernel users if it has been previously enabled by root:

```
echo 1 > /proc/sys/net/core/bpf_jit_enable
```

For JIT developers, doing audits etc, each compile run can output the generated opcode image into the kernel log via:

```
echo 2 > /proc/sys/net/core/bpf_jit_enable
```

Example output from dmesg:

```
[ 3389.935842] flen=6 proglen=70 pass=3 image=ffffffffa0069c8f
[ 3389.935847] JIT code: 00000000: 55 48 89 e5 48 83 ec 60 48 89 5d f8 44 8b 4f 68
[ 3389.935849] JIT code: 00000010: 44 2b 4f 6c 4c 8b 87 d8 00 00 00 be 0c 00 00 00
[ 3389.935850] JIT code: 00000020: e8 1d 94 ff e0 3d 00 08 00 00 75 16 be 17 00 00
[ 3389.935851] JIT code: 00000030: 00 e8 28 94 ff e0 83 f8 01 75 07 b8 ff ff 00 00
[ 3389.935852] JIT code: 00000040: eb 02 31 c0 c9 c3
```

When CONFIG_BPF_JIT_ALWAYS_ON is enabled, bpf_jit_enable is permanently set to 1 and setting any other value than that will return in failure. This is even the case for setting bpf_jit_enable to 2, since dumping the final JIT image into the kernel log is discouraged and introspection through bpftool (under tools/bpf/bpftool/) is the generally recommended approach instead.

In the kernel source tree under tools/bpf/, there's bpf_jit_disasm for generating disassembly out of the kernel log's hexdump:

```
# ./bpf_jit_disasm
70 bytes emitted from JIT compiler (pass:3, flen:6)
ffffffffa0069c8f + <x>:
0:       push   %rbp
1:       mov    %rsp,%rbp
4:       sub    $0x60,%rsp
8:       mov    %rbx,-0x8(%rbp)
c:       mov    0x68(%rdi),%r9d
10:      sub    0x6c(%rdi),%r9d
14:      mov    0xd8(%rdi),%r8
1b:      mov    $0xc,%esi
20:      callq  0xffffffffe0ff9442
25:      cmp    $0x800,%eax
2a:      jne    0x0000000000000042
2c:      mov    $0x17,%esi
31:      callq  0xffffffffe0ff945e
36:      cmp    $0x1,%eax
39:      jne    0x0000000000000042
3b:      mov    $0xffff,%eax
40:      jmp    0x0000000000000044
42:      xor    %eax,%eax
44:      leaveq
45:      retq

Issuing option `-o` will "annotate" opcodes to resulting assembler
instructions, which can be very useful for JIT developers:

# ./bpf_jit_disasm -o
70 bytes emitted from JIT compiler (pass:3, flen:6)
ffffffffa0069c8f + <x>:
0:       push   %rbp
         55
1:       mov    %rsp,%rbp
         48 89 e5
4:       sub    $0x60,%rsp
         48 83 ec 60
8:       mov    %rbx,-0x8(%rbp)
         48 89 5d f8
c:       mov    0x68(%rdi),%r9d
         44 8b 4f 68
10:      sub    0x6c(%rdi),%r9d
         44 2b 4f 6c
14:      mov    0xd8(%rdi),%r8
         4c 8b 87 d8 00 00 00
1b:      mov    $0xc,%esi
         be 0c 00 00 00
20:      callq  0xffffffffe0ff9442
         e8 1d 94 ff e0
25:      cmp    $0x800,%eax
         3d 00 08 00 00
2a:      jne    0x0000000000000042
         75 16
2c:      mov    $0x17,%esi
         be 17 00 00 00
31:      callq  0xffffffffe0ff945e
         e8 28 94 ff e0
36:      cmp    $0x1,%eax
         83 f8 01
39:      jne    0x0000000000000042
         75 07
3b:      mov    $0xffff,%eax
         b8 ff ff 00 00
40:      jmp    0x0000000000000044
         eb 02
42:      xor    %eax,%eax
         31 c0
44:      leaveq
         c9
45:      retq
         c3
```

For BPF JIT developers, bpf_jit_disasm, bpf_asm and bpf_dbg provides a useful toolchain for developing and testing the kernel's JIT compiler.

BPF kernel internals

BPF kernel internals

Internally, for the kernel interpreter, a different instruction set format with similar underlying principles from BPF described in previous paragraphs is being used. However, the instruction set format is modelled closer to the underlying architecture to mimic native instruction sets, so that a better performance can be achieved (more details later). This new ISA is called eBPF. See the ../bpf/index.rst for details. (Note: eBPF which originates from [e]xtended BPF is not the same as BPF extensions! While eBPF is an ISA, BPF extensions date back to classic BPF's 'overloading' of BPF_LD | BPF_{B,H,W} | BPF_ABS instruction.)

The new instruction set was originally designed with the possible goal in mind to write programs in "restricted C" and compile into eBPF with a optional GCC/LLVM backend, so that it can just-in-time map to modern 64-bit CPUs with minimal performance overhead over two steps, that is, C -> eBPF -> native code.

Currently, the new format is being used for running user BPF programs, which includes seccomp BPF, classic socket filters, cls_bpf traffic classifier, team driver's classifier for its load-balancing mode, netfilter's xt_bpf extension, PTP dissector/classifier, and much more. They are all internally converted by the kernel into the new instruction set representation and run in the eBPF interpreter. For in-kernel handlers, this all works transparently by using bpf_prog_create() for setting up the filter, resp. bpf_prog_destroy() for destroying it. The function bpf_prog_run(filter, ctx) transparently invokes eBPF interpreter or JITed code to run the filter. 'filter' is a pointer to struct bpf_prog that we got from bpf_prog_create(), and 'ctx' the given context (e.g. skb pointer). All constraints and restrictions from bpf_check_classic() apply before a conversion to the new layout is being done behind the scenes!

Currently, the classic BPF format is being used for JITing on most 32-bit architectures, whereas x86-64, aarch64, s390x, powerpc64, sparc64, arm32, riscv64, riscv32 perform JIT compilation from eBPF instruction set.

Testing

Next to the BPF toolchain, the kernel also ships a test module that contains various test cases for classic and eBPF that can be executed against the BPF interpreter and JIT compiler. It can be found in lib/test_bpf.c and enabled via Kconfig:

```
CONFIG_TEST_BPF=m
```

After the module has been built and installed, the test suite can be executed via insmod or modprobe against 'test_bpf' module. Results of the test cases including timings in nsec can be found in the kernel log (dmesg).

Misc

Also trinity, the Linux syscall fuzzer, has built-in support for BPF and SECCOMP-BPF kernel fuzzing.

Written by

The document was written in the hope that it is found useful and in order to give potential BPF hackers or security auditors a better overview of the underlying architecture.

- Jay Schulist <jschlst@samba.org>
- Daniel Borkmann <daniel@iogearbox.net>
- Alexei Starovoitov <ast@kernel.org>