Linux Socket Filtering aka Berkeley Packet Filter (BPF)

=======================================================

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

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

\*\* (Accelerated) VLAN w/ id 10:

ld vlan\_tci

jneq #10, 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 \*/

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

--------------------

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' or 'internal BPF' interchangeably. (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.)

It is designed to be JITed with one to one mapping, which can also open up

the possibility for GCC/LLVM compilers to generate optimized eBPF code through

an eBPF backend that performs almost as fast as natively compiled code.

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 macro

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.

Some core changes of the new internal format:

- Number of registers increase from 2 to 10:

The old format had two registers A and X, and a hidden frame pointer. The

new layout extends this to be 10 internal registers and a read-only frame

pointer. Since 64-bit CPUs are passing arguments to functions via registers

the number of args from eBPF program to in-kernel function is restricted

to 5 and one register is used to accept return value from an in-kernel

function. Natively, x86\_64 passes first 6 arguments in registers, aarch64/

sparcv9/mips64 have 7 - 8 registers for arguments; x86\_64 has 6 callee saved

registers, and aarch64/sparcv9/mips64 have 11 or more callee saved registers.

Therefore, eBPF calling convention is defined as:

\* R0 - return value from in-kernel function, and exit value for eBPF program

\* R1 - R5 - arguments from eBPF program to in-kernel function

\* R6 - R9 - callee saved registers that in-kernel function will preserve

\* R10 - read-only frame pointer to access stack

Thus, all eBPF registers map one to one to HW registers on x86\_64, aarch64,

etc, and eBPF calling convention maps directly to ABIs used by the kernel on

64-bit architectures.

On 32-bit architectures JIT may map programs that use only 32-bit arithmetic

and may let more complex programs to be interpreted.

R0 - R5 are scratch registers and eBPF program needs spill/fill them if

necessary across calls. Note that there is only one eBPF program (== one

eBPF main routine) and it cannot call other eBPF functions, it can only

call predefined in-kernel functions, though.

- Register width increases from 32-bit to 64-bit:

Still, the semantics of the original 32-bit ALU operations are preserved

via 32-bit subregisters. All eBPF registers are 64-bit with 32-bit lower

subregisters that zero-extend into 64-bit if they are being written to.

That behavior maps directly to x86\_64 and arm64 subregister definition, but

makes other JITs more difficult.

32-bit architectures run 64-bit internal BPF programs via interpreter.

Their JITs may convert BPF programs that only use 32-bit subregisters into

native instruction set and let the rest being interpreted.

Operation is 64-bit, because on 64-bit architectures, pointers are also

64-bit wide, and we want to pass 64-bit values in/out of kernel functions,

so 32-bit eBPF registers would otherwise require to define register-pair

ABI, thus, there won't be able to use a direct eBPF register to HW register

mapping and JIT would need to do combine/split/move operations for every

register in and out of the function, which is complex, bug prone and slow.

Another reason is the use of atomic 64-bit counters.

- Conditional jt/jf targets replaced with jt/fall-through:

While the original design has constructs such as "if (cond) jump\_true;

else jump\_false;", they are being replaced into alternative constructs like

"if (cond) jump\_true; /\* else fall-through \*/".

- Introduces bpf\_call insn and register passing convention for zero overhead

calls from/to other kernel functions:

Before an in-kernel function call, the internal BPF program needs to

place function arguments into R1 to R5 registers to satisfy calling

convention, then the interpreter will take them from registers and pass

to in-kernel function. If R1 - R5 registers are mapped to CPU registers

that are used for argument passing on given architecture, the JIT compiler

doesn't need to emit extra moves. Function arguments will be in the correct

registers and BPF\_CALL instruction will be JITed as single 'call' HW

instruction. This calling convention was picked to cover common call

situations without performance penalty.

After an in-kernel function call, R1 - R5 are reset to unreadable and R0 has

a return value of the function. Since R6 - R9 are callee saved, their state

is preserved across the call.

For example, consider three C functions:

u64 f1() { return (\*\_f2)(1); }

u64 f2(u64 a) { return f3(a + 1, a); }

u64 f3(u64 a, u64 b) { return a - b; }

GCC can compile f1, f3 into x86\_64:

f1:

movl $1, %edi

movq \_f2(%rip), %rax

jmp \*%rax

f3:

movq %rdi, %rax

subq %rsi, %rax

ret

Function f2 in eBPF may look like:

f2:

bpf\_mov R2, R1

bpf\_add R1, 1

bpf\_call f3

bpf\_exit

If f2 is JITed and the pointer stored to '\_f2'. The calls f1 -> f2 -> f3 and

returns will be seamless. Without JIT, \_\_bpf\_prog\_run() interpreter needs to

be used to call into f2.

For practical reasons all eBPF programs have only one argument 'ctx' which is

already placed into R1 (e.g. on \_\_bpf\_prog\_run() startup) and the programs

can call kernel functions with up to 5 arguments. Calls with 6 or more arguments

are currently not supported, but these restrictions can be lifted if necessary

in the future.

On 64-bit architectures all register map to HW registers one to one. For

example, x86\_64 JIT compiler can map them as ...

R0 - rax

R1 - rdi

R2 - rsi

R3 - rdx

R4 - rcx

R5 - r8

R6 - rbx

R7 - r13

R8 - r14

R9 - r15

R10 - rbp

... since x86\_64 ABI mandates rdi, rsi, rdx, rcx, r8, r9 for argument passing

and rbx, r12 - r15 are callee saved.

Then the following internal BPF pseudo-program:

bpf\_mov R6, R1 /\* save ctx \*/

bpf\_mov R2, 2

bpf\_mov R3, 3

bpf\_mov R4, 4

bpf\_mov R5, 5

bpf\_call foo

bpf\_mov R7, R0 /\* save foo() return value \*/

bpf\_mov R1, R6 /\* restore ctx for next call \*/

bpf\_mov R2, 6

bpf\_mov R3, 7

bpf\_mov R4, 8

bpf\_mov R5, 9

bpf\_call bar

bpf\_add R0, R7

bpf\_exit

After JIT to x86\_64 may look like:

push %rbp

mov %rsp,%rbp

sub $0x228,%rsp

mov %rbx,-0x228(%rbp)

mov %r13,-0x220(%rbp)

mov %rdi,%rbx

mov $0x2,%esi

mov $0x3,%edx

mov $0x4,%ecx

mov $0x5,%r8d

callq foo

mov %rax,%r13

mov %rbx,%rdi

mov $0x6,%esi

mov $0x7,%edx

mov $0x8,%ecx

mov $0x9,%r8d

callq bar

add %r13,%rax

mov -0x228(%rbp),%rbx

mov -0x220(%rbp),%r13

leaveq

retq

Which is in this example equivalent in C to:

u64 bpf\_filter(u64 ctx)

{

return foo(ctx, 2, 3, 4, 5) + bar(ctx, 6, 7, 8, 9);

}

In-kernel functions foo() and bar() with prototype: u64 (\*)(u64 arg1, u64

arg2, u64 arg3, u64 arg4, u64 arg5); will receive arguments in proper

registers and place their return value into '%rax' which is R0 in eBPF.

Prologue and epilogue are emitted by JIT and are implicit in the

interpreter. R0-R5 are scratch registers, so eBPF program needs to preserve

them across the calls as defined by calling convention.

For example the following program is invalid:

bpf\_mov R1, 1

bpf\_call foo

bpf\_mov R0, R1

bpf\_exit

After the call the registers R1-R5 contain junk values and cannot be read.

An in-kernel eBPF verifier is used to validate internal BPF programs.

Also in the new design, eBPF is limited to 4096 insns, which means that any

program will terminate quickly and will only call a fixed number of kernel

functions. Original BPF and the new format are two operand instructions,

which helps to do one-to-one mapping between eBPF insn and x86 insn during JIT.

The input context pointer for invoking the interpreter function is generic,

its content is defined by a specific use case. For seccomp register R1 points

to seccomp\_data, for converted BPF filters R1 points to a skb.

A program, that is translated internally consists of the following elements:

op:16, jt:8, jf:8, k:32 ==> op:8, dst\_reg:4, src\_reg:4, off:16, imm:32

So far 87 internal BPF instructions were implemented. 8-bit 'op' opcode field

has room for new instructions. Some of them may use 16/24/32 byte encoding. New

instructions must be multiple of 8 bytes to preserve backward compatibility.

Internal BPF is a general purpose RISC instruction set. Not every register and

every instruction are used during translation from original BPF to new format.

For example, socket filters are not using 'exclusive add' instruction, but

tracing filters may do to maintain counters of events, for example. Register R9

is not used by socket filters either, but more complex filters may be running

out of registers and would have to resort to spill/fill to stack.

Internal BPF can be used as a generic assembler for last step performance

optimizations, socket filters and seccomp are using it as assembler. Tracing

filters may use it as assembler to generate code from kernel. In kernel usage

may not be bounded by security considerations, since generated internal BPF code

may be optimizing internal code path and not being exposed to the user space.

Safety of internal BPF can come from a verifier (TBD). In such use cases as

described, it may be used as safe instruction set.

Just like the original BPF, the new format runs within a controlled environment,

is deterministic and the kernel can easily prove that. The safety of the program

can be determined in two steps: first step does depth-first-search to disallow

loops and other CFG validation; second step starts from the first insn and

descends all possible paths. It simulates execution of every insn and observes

the state change of registers and stack.

eBPF opcode encoding

--------------------

eBPF is reusing most of the opcode encoding from classic to simplify conversion

of classic BPF to eBPF. For arithmetic and jump instructions the 8-bit 'code'

field is divided into three parts:

+----------------+--------+--------------------+

| 4 bits | 1 bit | 3 bits |

| operation code | source | instruction class |

+----------------+--------+--------------------+

(MSB) (LSB)

Three LSB bits store instruction class which is one of:

Classic BPF classes: eBPF classes:

BPF\_LD 0x00 BPF\_LD 0x00

BPF\_LDX 0x01 BPF\_LDX 0x01

BPF\_ST 0x02 BPF\_ST 0x02

BPF\_STX 0x03 BPF\_STX 0x03

BPF\_ALU 0x04 BPF\_ALU 0x04

BPF\_JMP 0x05 BPF\_JMP 0x05

BPF\_RET 0x06 BPF\_JMP32 0x06

BPF\_MISC 0x07 BPF\_ALU64 0x07

When BPF\_CLASS(code) == BPF\_ALU or BPF\_JMP, 4th bit encodes source operand ...

BPF\_K 0x00

BPF\_X 0x08

\* in classic BPF, this means:

BPF\_SRC(code) == BPF\_X - use register X as source operand

BPF\_SRC(code) == BPF\_K - use 32-bit immediate as source operand

\* in eBPF, this means:

BPF\_SRC(code) == BPF\_X - use 'src\_reg' register as source operand

BPF\_SRC(code) == BPF\_K - use 32-bit immediate as source operand

... and four MSB bits store operation code.

If BPF\_CLASS(code) == BPF\_ALU or BPF\_ALU64 [ in eBPF ], BPF\_OP(code) is one of:

BPF\_ADD 0x00

BPF\_SUB 0x10

BPF\_MUL 0x20

BPF\_DIV 0x30

BPF\_OR 0x40

BPF\_AND 0x50

BPF\_LSH 0x60

BPF\_RSH 0x70

BPF\_NEG 0x80

BPF\_MOD 0x90

BPF\_XOR 0xa0

BPF\_MOV 0xb0 /\* eBPF only: mov reg to reg \*/

BPF\_ARSH 0xc0 /\* eBPF only: sign extending shift right \*/

BPF\_END 0xd0 /\* eBPF only: endianness conversion \*/

If BPF\_CLASS(code) == BPF\_JMP or BPF\_JMP32 [ in eBPF ], BPF\_OP(code) is one of:

BPF\_JA 0x00 /\* BPF\_JMP only \*/

BPF\_JEQ 0x10

BPF\_JGT 0x20

BPF\_JGE 0x30

BPF\_JSET 0x40

BPF\_JNE 0x50 /\* eBPF only: jump != \*/

BPF\_JSGT 0x60 /\* eBPF only: signed '>' \*/

BPF\_JSGE 0x70 /\* eBPF only: signed '>=' \*/

BPF\_CALL 0x80 /\* eBPF BPF\_JMP only: function call \*/

BPF\_EXIT 0x90 /\* eBPF BPF\_JMP only: function return \*/

BPF\_JLT 0xa0 /\* eBPF only: unsigned '<' \*/

BPF\_JLE 0xb0 /\* eBPF only: unsigned '<=' \*/

BPF\_JSLT 0xc0 /\* eBPF only: signed '<' \*/

BPF\_JSLE 0xd0 /\* eBPF only: signed '<=' \*/

So BPF\_ADD | BPF\_X | BPF\_ALU means 32-bit addition in both classic BPF

and eBPF. There are only two registers in classic BPF, so it means A += X.

In eBPF it means dst\_reg = (u32) dst\_reg + (u32) src\_reg; similarly,

BPF\_XOR | BPF\_K | BPF\_ALU means A ^= imm32 in classic BPF and analogous

src\_reg = (u32) src\_reg ^ (u32) imm32 in eBPF.

Classic BPF is using BPF\_MISC class to represent A = X and X = A moves.

eBPF is using BPF\_MOV | BPF\_X | BPF\_ALU code instead. Since there are no

BPF\_MISC operations in eBPF, the class 7 is used as BPF\_ALU64 to mean

exactly the same operations as BPF\_ALU, but with 64-bit wide operands

instead. So BPF\_ADD | BPF\_X | BPF\_ALU64 means 64-bit addition, i.e.:

dst\_reg = dst\_reg + src\_reg

Classic BPF wastes the whole BPF\_RET class to represent a single 'ret'

operation. Classic BPF\_RET | BPF\_K means copy imm32 into return register

and perform function exit. eBPF is modeled to match CPU, so BPF\_JMP | BPF\_EXIT

in eBPF means function exit only. The eBPF program needs to store return

value into register R0 before doing a BPF\_EXIT. Class 6 in eBPF is used as

BPF\_JMP32 to mean exactly the same operations as BPF\_JMP, but with 32-bit wide

operands for the comparisons instead.

For load and store instructions the 8-bit 'code' field is divided as:

+--------+--------+-------------------+

| 3 bits | 2 bits | 3 bits |

| mode | size | instruction class |

+--------+--------+-------------------+

(MSB) (LSB)

Size modifier is one of ...

BPF\_W 0x00 /\* word \*/

BPF\_H 0x08 /\* half word \*/

BPF\_B 0x10 /\* byte \*/

BPF\_DW 0x18 /\* eBPF only, double word \*/

... which encodes size of load/store operation:

B - 1 byte

H - 2 byte

W - 4 byte

DW - 8 byte (eBPF only)

Mode modifier is one of:

BPF\_IMM 0x00 /\* used for 32-bit mov in classic BPF and 64-bit in eBPF \*/

BPF\_ABS 0x20

BPF\_IND 0x40

BPF\_MEM 0x60

BPF\_LEN 0x80 /\* classic BPF only, reserved in eBPF \*/

BPF\_MSH 0xa0 /\* classic BPF only, reserved in eBPF \*/

BPF\_XADD 0xc0 /\* eBPF only, exclusive add \*/

eBPF has two non-generic instructions: (BPF\_ABS | <size> | BPF\_LD) and

(BPF\_IND | <size> | BPF\_LD) which are used to access packet data.

They had to be carried over from classic to have strong performance of

socket filters running in eBPF interpreter. These instructions can only

be used when interpreter context is a pointer to 'struct sk\_buff' and

have seven implicit operands. Register R6 is an implicit input that must

contain pointer to sk\_buff. Register R0 is an implicit output which contains

the data fetched from the packet. Registers R1-R5 are scratch registers

and must not be used to store the data across BPF\_ABS | BPF\_LD or

BPF\_IND | BPF\_LD instructions.

These instructions have implicit program exit condition as well. When

eBPF program is trying to access the data beyond the packet boundary,

the interpreter will abort the execution of the program. JIT compilers

therefore must preserve this property. src\_reg and imm32 fields are

explicit inputs to these instructions.

For example:

BPF\_IND | BPF\_W | BPF\_LD means:

R0 = ntohl(\*(u32 \*) (((struct sk\_buff \*) R6)->data + src\_reg + imm32))

and R1 - R5 were scratched.

Unlike classic BPF instruction set, eBPF has generic load/store operations:

BPF\_MEM | <size> | BPF\_STX: \*(size \*) (dst\_reg + off) = src\_reg

BPF\_MEM | <size> | BPF\_ST: \*(size \*) (dst\_reg + off) = imm32

BPF\_MEM | <size> | BPF\_LDX: dst\_reg = \*(size \*) (src\_reg + off)

BPF\_XADD | BPF\_W | BPF\_STX: lock xadd \*(u32 \*)(dst\_reg + off16) += src\_reg

BPF\_XADD | BPF\_DW | BPF\_STX: lock xadd \*(u64 \*)(dst\_reg + off16) += src\_reg

Where size is one of: BPF\_B or BPF\_H or BPF\_W or BPF\_DW. Note that 1 and

2 byte atomic increments are not supported.

eBPF has one 16-byte instruction: BPF\_LD | BPF\_DW | BPF\_IMM which consists

of two consecutive 'struct bpf\_insn' 8-byte blocks and interpreted as single

instruction that loads 64-bit immediate value into a dst\_reg.

Classic BPF has similar instruction: BPF\_LD | BPF\_W | BPF\_IMM which loads

32-bit immediate value into a register.

eBPF verifier

-------------

The safety of the eBPF program is determined in two steps.

First step does DAG check to disallow loops and other CFG validation.

In particular it will detect programs that have unreachable instructions.

(though classic BPF checker allows them)

Second step starts from the first insn and descends all possible paths.

It simulates execution of every insn and observes the state change of

registers and stack.

At the start of the program the register R1 contains a pointer to context

and has type PTR\_TO\_CTX.

If verifier sees an insn that does R2=R1, then R2 has now type

PTR\_TO\_CTX as well and can be used on the right hand side of expression.

If R1=PTR\_TO\_CTX and insn is R2=R1+R1, then R2=SCALAR\_VALUE,

since addition of two valid pointers makes invalid pointer.

(In 'secure' mode verifier will reject any type of pointer arithmetic to make

sure that kernel addresses don't leak to unprivileged users)

If register was never written to, it's not readable:

bpf\_mov R0 = R2

bpf\_exit

will be rejected, since R2 is unreadable at the start of the program.

After kernel function call, R1-R5 are reset to unreadable and

R0 has a return type of the function.

Since R6-R9 are callee saved, their state is preserved across the call.

bpf\_mov R6 = 1

bpf\_call foo

bpf\_mov R0 = R6

bpf\_exit

is a correct program. If there was R1 instead of R6, it would have

been rejected.

load/store instructions are allowed only with registers of valid types, which

are PTR\_TO\_CTX, PTR\_TO\_MAP, PTR\_TO\_STACK. They are bounds and alignment checked.

For example:

bpf\_mov R1 = 1

bpf\_mov R2 = 2

bpf\_xadd \*(u32 \*)(R1 + 3) += R2

bpf\_exit

will be rejected, since R1 doesn't have a valid pointer type at the time of

execution of instruction bpf\_xadd.

At the start R1 type is PTR\_TO\_CTX (a pointer to generic 'struct bpf\_context')

A callback is used to customize verifier to restrict eBPF program access to only

certain fields within ctx structure with specified size and alignment.

For example, the following insn:

bpf\_ld R0 = \*(u32 \*)(R6 + 8)

intends to load a word from address R6 + 8 and store it into R0

If R6=PTR\_TO\_CTX, via is\_valid\_access() callback the verifier will know

that offset 8 of size 4 bytes can be accessed for reading, otherwise

the verifier will reject the program.

If R6=PTR\_TO\_STACK, then access should be aligned and be within

stack bounds, which are [-MAX\_BPF\_STACK, 0). In this example offset is 8,

so it will fail verification, since it's out of bounds.

The verifier will allow eBPF program to read data from stack only after

it wrote into it.

Classic BPF verifier does similar check with M[0-15] memory slots.

For example:

bpf\_ld R0 = \*(u32 \*)(R10 - 4)

bpf\_exit

is invalid program.

Though R10 is correct read-only register and has type PTR\_TO\_STACK

and R10 - 4 is within stack bounds, there were no stores into that location.

Pointer register spill/fill is tracked as well, since four (R6-R9)

callee saved registers may not be enough for some programs.

Allowed function calls are customized with bpf\_verifier\_ops->get\_func\_proto()

The eBPF verifier will check that registers match argument constraints.

After the call register R0 will be set to return type of the function.

Function calls is a main mechanism to extend functionality of eBPF programs.

Socket filters may let programs to call one set of functions, whereas tracing

filters may allow completely different set.

If a function made accessible to eBPF program, it needs to be thought through

from safety point of view. The verifier will guarantee that the function is

called with valid arguments.

seccomp vs socket filters have different security restrictions for classic BPF.

Seccomp solves this by two stage verifier: classic BPF verifier is followed

by seccomp verifier. In case of eBPF one configurable verifier is shared for

all use cases.

See details of eBPF verifier in kernel/bpf/verifier.c

Register value tracking

-----------------------

In order to determine the safety of an eBPF program, the verifier must track

the range of possible values in each register and also in each stack slot.

This is done with 'struct bpf\_reg\_state', defined in include/linux/

bpf\_verifier.h, which unifies tracking of scalar and pointer values. Each

register state has a type, which is either NOT\_INIT (the register has not been

written to), SCALAR\_VALUE (some value which is not usable as a pointer), or a

pointer type. The types of pointers describe their base, as follows:

PTR\_TO\_CTX Pointer to bpf\_context.

CONST\_PTR\_TO\_MAP Pointer to struct bpf\_map. "Const" because arithmetic

on these pointers is forbidden.

PTR\_TO\_MAP\_VALUE Pointer to the value stored in a map element.

PTR\_TO\_MAP\_VALUE\_OR\_NULL

Either a pointer to a map value, or NULL; map accesses

(see section 'eBPF maps', below) return this type,

which becomes a PTR\_TO\_MAP\_VALUE when checked != NULL.

Arithmetic on these pointers is forbidden.

PTR\_TO\_STACK Frame pointer.

PTR\_TO\_PACKET skb->data.

PTR\_TO\_PACKET\_END skb->data + headlen; arithmetic forbidden.

PTR\_TO\_SOCKET Pointer to struct bpf\_sock\_ops, implicitly refcounted.

PTR\_TO\_SOCKET\_OR\_NULL

Either a pointer to a socket, or NULL; socket lookup

returns this type, which becomes a PTR\_TO\_SOCKET when

checked != NULL. PTR\_TO\_SOCKET is reference-counted,

so programs must release the reference through the

socket release function before the end of the program.

Arithmetic on these pointers is forbidden.

However, a pointer may be offset from this base (as a result of pointer

arithmetic), and this is tracked in two parts: the 'fixed offset' and 'variable

offset'. The former is used when an exactly-known value (e.g. an immediate

operand) is added to a pointer, while the latter is used for values which are

not exactly known. The variable offset is also used in SCALAR\_VALUEs, to track

the range of possible values in the register.

The verifier's knowledge about the variable offset consists of:

\* minimum and maximum values as unsigned

\* minimum and maximum values as signed

\* knowledge of the values of individual bits, in the form of a 'tnum': a u64

'mask' and a u64 'value'. 1s in the mask represent bits whose value is unknown;

1s in the value represent bits known to be 1. Bits known to be 0 have 0 in both

mask and value; no bit should ever be 1 in both. For example, if a byte is read

into a register from memory, the register's top 56 bits are known zero, while

the low 8 are unknown - which is represented as the tnum (0x0; 0xff). If we

then OR this with 0x40, we get (0x40; 0xbf), then if we add 1 we get (0x0;

0x1ff), because of potential carries.

Besides arithmetic, the register state can also be updated by conditional

branches. For instance, if a SCALAR\_VALUE is compared > 8, in the 'true' branch

it will have a umin\_value (unsigned minimum value) of 9, whereas in the 'false'

branch it will have a umax\_value of 8. A signed compare (with BPF\_JSGT or

BPF\_JSGE) would instead update the signed minimum/maximum values. Information

from the signed and unsigned bounds can be combined; for instance if a value is

first tested < 8 and then tested s> 4, the verifier will conclude that the value

is also > 4 and s< 8, since the bounds prevent crossing the sign boundary.

PTR\_TO\_PACKETs with a variable offset part have an 'id', which is common to all

pointers sharing that same variable offset. This is important for packet range

checks: after adding a variable to a packet pointer register A, if you then copy

it to another register B and then add a constant 4 to A, both registers will

share the same 'id' but the A will have a fixed offset of +4. Then if A is

bounds-checked and found to be less than a PTR\_TO\_PACKET\_END, the register B is

now known to have a safe range of at least 4 bytes. See 'Direct packet access',

below, for more on PTR\_TO\_PACKET ranges.

The 'id' field is also used on PTR\_TO\_MAP\_VALUE\_OR\_NULL, common to all copies of

the pointer returned from a map lookup. This means that when one copy is

checked and found to be non-NULL, all copies can become PTR\_TO\_MAP\_VALUEs.

As well as range-checking, the tracked information is also used for enforcing

alignment of pointer accesses. For instance, on most systems the packet pointer

is 2 bytes after a 4-byte alignment. If a program adds 14 bytes to that to jump

over the Ethernet header, then reads IHL and addes (IHL \* 4), the resulting

pointer will have a variable offset known to be 4n+2 for some n, so adding the 2

bytes (NET\_IP\_ALIGN) gives a 4-byte alignment and so word-sized accesses through

that pointer are safe.

The 'id' field is also used on PTR\_TO\_SOCKET and PTR\_TO\_SOCKET\_OR\_NULL, common

to all copies of the pointer returned from a socket lookup. This has similar

behaviour to the handling for PTR\_TO\_MAP\_VALUE\_OR\_NULL->PTR\_TO\_MAP\_VALUE, but

it also handles reference tracking for the pointer. PTR\_TO\_SOCKET implicitly

represents a reference to the corresponding 'struct sock'. To ensure that the

reference is not leaked, it is imperative to NULL-check the reference and in

the non-NULL case, and pass the valid reference to the socket release function.

Direct packet access

--------------------

In cls\_bpf and act\_bpf programs the verifier allows direct access to the packet

data via skb->data and skb->data\_end pointers.

Ex:

1: r4 = \*(u32 \*)(r1 +80) /\* load skb->data\_end \*/

2: r3 = \*(u32 \*)(r1 +76) /\* load skb->data \*/

3: r5 = r3

4: r5 += 14

5: if r5 > r4 goto pc+16

R1=ctx R3=pkt(id=0,off=0,r=14) R4=pkt\_end R5=pkt(id=0,off=14,r=14) R10=fp

6: r0 = \*(u16 \*)(r3 +12) /\* access 12 and 13 bytes of the packet \*/

this 2byte load from the packet is safe to do, since the program author

did check 'if (skb->data + 14 > skb->data\_end) goto err' at insn #5 which

means that in the fall-through case the register R3 (which points to skb->data)

has at least 14 directly accessible bytes. The verifier marks it

as R3=pkt(id=0,off=0,r=14).

id=0 means that no additional variables were added to the register.

off=0 means that no additional constants were added.

r=14 is the range of safe access which means that bytes [R3, R3 + 14) are ok.

Note that R5 is marked as R5=pkt(id=0,off=14,r=14). It also points

to the packet data, but constant 14 was added to the register, so

it now points to 'skb->data + 14' and accessible range is [R5, R5 + 14 - 14)

which is zero bytes.

More complex packet access may look like:

R0=inv1 R1=ctx R3=pkt(id=0,off=0,r=14) R4=pkt\_end R5=pkt(id=0,off=14,r=14) R10=fp

6: r0 = \*(u8 \*)(r3 +7) /\* load 7th byte from the packet \*/

7: r4 = \*(u8 \*)(r3 +12)

8: r4 \*= 14

9: r3 = \*(u32 \*)(r1 +76) /\* load skb->data \*/

10: r3 += r4

11: r2 = r1

12: r2 <<= 48

13: r2 >>= 48

14: r3 += r2

15: r2 = r3

16: r2 += 8

17: r1 = \*(u32 \*)(r1 +80) /\* load skb->data\_end \*/

18: if r2 > r1 goto pc+2

R0=inv(id=0,umax\_value=255,var\_off=(0x0; 0xff)) R1=pkt\_end R2=pkt(id=2,off=8,r=8) R3=pkt(id=2,off=0,r=8) R4=inv(id=0,umax\_value=3570,var\_off=(0x0; 0xfffe)) R5=pkt(id=0,off=14,r=14) R10=fp

19: r1 = \*(u8 \*)(r3 +4)

The state of the register R3 is R3=pkt(id=2,off=0,r=8)

id=2 means that two 'r3 += rX' instructions were seen, so r3 points to some

offset within a packet and since the program author did

'if (r3 + 8 > r1) goto err' at insn #18, the safe range is [R3, R3 + 8).

The verifier only allows 'add'/'sub' operations on packet registers. Any other

operation will set the register state to 'SCALAR\_VALUE' and it won't be

available for direct packet access.

Operation 'r3 += rX' may overflow and become less than original skb->data,

therefore the verifier has to prevent that. So when it sees 'r3 += rX'

instruction and rX is more than 16-bit value, any subsequent bounds-check of r3

against skb->data\_end will not give us 'range' information, so attempts to read

through the pointer will give "invalid access to packet" error.

Ex. after insn 'r4 = \*(u8 \*)(r3 +12)' (insn #7 above) the state of r4 is

R4=inv(id=0,umax\_value=255,var\_off=(0x0; 0xff)) which means that upper 56 bits

of the register are guaranteed to be zero, and nothing is known about the lower

8 bits. After insn 'r4 \*= 14' the state becomes

R4=inv(id=0,umax\_value=3570,var\_off=(0x0; 0xfffe)), since multiplying an 8-bit

value by constant 14 will keep upper 52 bits as zero, also the least significant

bit will be zero as 14 is even. Similarly 'r2 >>= 48' will make

R2=inv(id=0,umax\_value=65535,var\_off=(0x0; 0xffff)), since the shift is not sign

extending. This logic is implemented in adjust\_reg\_min\_max\_vals() function,

which calls adjust\_ptr\_min\_max\_vals() for adding pointer to scalar (or vice

versa) and adjust\_scalar\_min\_max\_vals() for operations on two scalars.

The end result is that bpf program author can access packet directly

using normal C code as:

void \*data = (void \*)(long)skb->data;

void \*data\_end = (void \*)(long)skb->data\_end;

struct eth\_hdr \*eth = data;

struct iphdr \*iph = data + sizeof(\*eth);

struct udphdr \*udp = data + sizeof(\*eth) + sizeof(\*iph);

if (data + sizeof(\*eth) + sizeof(\*iph) + sizeof(\*udp) > data\_end)

return 0;

if (eth->h\_proto != htons(ETH\_P\_IP))

return 0;

if (iph->protocol != IPPROTO\_UDP || iph->ihl != 5)

return 0;

if (udp->dest == 53 || udp->source == 9)

...;

which makes such programs easier to write comparing to LD\_ABS insn

and significantly faster.

eBPF maps

---------

'maps' is a generic storage of different types for sharing data between kernel

and userspace.

The maps are accessed from user space via BPF syscall, which has commands:

- create a map with given type and attributes

map\_fd = bpf(BPF\_MAP\_CREATE, union bpf\_attr \*attr, u32 size)

using attr->map\_type, attr->key\_size, attr->value\_size, attr->max\_entries

returns process-local file descriptor or negative error

- lookup key in a given map

err = bpf(BPF\_MAP\_LOOKUP\_ELEM, union bpf\_attr \*attr, u32 size)

using attr->map\_fd, attr->key, attr->value

returns zero and stores found elem into value or negative error

- create or update key/value pair in a given map

err = bpf(BPF\_MAP\_UPDATE\_ELEM, union bpf\_attr \*attr, u32 size)

using attr->map\_fd, attr->key, attr->value

returns zero or negative error

- find and delete element by key in a given map

err = bpf(BPF\_MAP\_DELETE\_ELEM, union bpf\_attr \*attr, u32 size)

using attr->map\_fd, attr->key

- to delete map: close(fd)

Exiting process will delete maps automatically

userspace programs use this syscall to create/access maps that eBPF programs

are concurrently updating.

maps can have different types: hash, array, bloom filter, radix-tree, etc.

The map is defined by:

. type

. max number of elements

. key size in bytes

. value size in bytes

Pruning

-------

The verifier does not actually walk all possible paths through the program. For

each new branch to analyse, the verifier looks at all the states it's previously

been in when at this instruction. If any of them contain the current state as a

subset, the branch is 'pruned' - that is, the fact that the previous state was

accepted implies the current state would be as well. For instance, if in the

previous state, r1 held a packet-pointer, and in the current state, r1 holds a

packet-pointer with a range as long or longer and at least as strict an

alignment, then r1 is safe. Similarly, if r2 was NOT\_INIT before then it can't

have been used by any path from that point, so any value in r2 (including

another NOT\_INIT) is safe. The implementation is in the function regsafe().

Pruning considers not only the registers but also the stack (and any spilled

registers it may hold). They must all be safe for the branch to be pruned.

This is implemented in states\_equal().

Understanding eBPF verifier messages

------------------------------------

The following are few examples of invalid eBPF programs and verifier error

messages as seen in the log:

Program with unreachable instructions:

static struct bpf\_insn prog[] = {

BPF\_EXIT\_INSN(),

BPF\_EXIT\_INSN(),

};

Error:

unreachable insn 1

Program that reads uninitialized register:

BPF\_MOV64\_REG(BPF\_REG\_0, BPF\_REG\_2),

BPF\_EXIT\_INSN(),

Error:

0: (bf) r0 = r2

R2 !read\_ok

Program that doesn't initialize R0 before exiting:

BPF\_MOV64\_REG(BPF\_REG\_2, BPF\_REG\_1),

BPF\_EXIT\_INSN(),

Error:

0: (bf) r2 = r1

1: (95) exit

R0 !read\_ok

Program that accesses stack out of bounds:

BPF\_ST\_MEM(BPF\_DW, BPF\_REG\_10, 8, 0),

BPF\_EXIT\_INSN(),

Error:

0: (7a) \*(u64 \*)(r10 +8) = 0

invalid stack off=8 size=8

Program that doesn't initialize stack before passing its address into function:

BPF\_MOV64\_REG(BPF\_REG\_2, BPF\_REG\_10),

BPF\_ALU64\_IMM(BPF\_ADD, BPF\_REG\_2, -8),

BPF\_LD\_MAP\_FD(BPF\_REG\_1, 0),

BPF\_RAW\_INSN(BPF\_JMP | BPF\_CALL, 0, 0, 0, BPF\_FUNC\_map\_lookup\_elem),

BPF\_EXIT\_INSN(),

Error:

0: (bf) r2 = r10

1: (07) r2 += -8

2: (b7) r1 = 0x0

3: (85) call 1

invalid indirect read from stack off -8+0 size 8

Program that uses invalid map\_fd=0 while calling to map\_lookup\_elem() function:

BPF\_ST\_MEM(BPF\_DW, BPF\_REG\_10, -8, 0),

BPF\_MOV64\_REG(BPF\_REG\_2, BPF\_REG\_10),

BPF\_ALU64\_IMM(BPF\_ADD, BPF\_REG\_2, -8),

BPF\_LD\_MAP\_FD(BPF\_REG\_1, 0),

BPF\_RAW\_INSN(BPF\_JMP | BPF\_CALL, 0, 0, 0, BPF\_FUNC\_map\_lookup\_elem),

BPF\_EXIT\_INSN(),

Error:

0: (7a) \*(u64 \*)(r10 -8) = 0

1: (bf) r2 = r10

2: (07) r2 += -8

3: (b7) r1 = 0x0

4: (85) call 1

fd 0 is not pointing to valid bpf\_map

Program that doesn't check return value of map\_lookup\_elem() before accessing

map element:

BPF\_ST\_MEM(BPF\_DW, BPF\_REG\_10, -8, 0),

BPF\_MOV64\_REG(BPF\_REG\_2, BPF\_REG\_10),

BPF\_ALU64\_IMM(BPF\_ADD, BPF\_REG\_2, -8),

BPF\_LD\_MAP\_FD(BPF\_REG\_1, 0),

BPF\_RAW\_INSN(BPF\_JMP | BPF\_CALL, 0, 0, 0, BPF\_FUNC\_map\_lookup\_elem),

BPF\_ST\_MEM(BPF\_DW, BPF\_REG\_0, 0, 0),

BPF\_EXIT\_INSN(),

Error:

0: (7a) \*(u64 \*)(r10 -8) = 0

1: (bf) r2 = r10

2: (07) r2 += -8

3: (b7) r1 = 0x0

4: (85) call 1

5: (7a) \*(u64 \*)(r0 +0) = 0

R0 invalid mem access 'map\_value\_or\_null'

Program that correctly checks map\_lookup\_elem() returned value for NULL, but

accesses the memory with incorrect alignment:

BPF\_ST\_MEM(BPF\_DW, BPF\_REG\_10, -8, 0),

BPF\_MOV64\_REG(BPF\_REG\_2, BPF\_REG\_10),

BPF\_ALU64\_IMM(BPF\_ADD, BPF\_REG\_2, -8),

BPF\_LD\_MAP\_FD(BPF\_REG\_1, 0),

BPF\_RAW\_INSN(BPF\_JMP | BPF\_CALL, 0, 0, 0, BPF\_FUNC\_map\_lookup\_elem),

BPF\_JMP\_IMM(BPF\_JEQ, BPF\_REG\_0, 0, 1),

BPF\_ST\_MEM(BPF\_DW, BPF\_REG\_0, 4, 0),

BPF\_EXIT\_INSN(),

Error:

0: (7a) \*(u64 \*)(r10 -8) = 0

1: (bf) r2 = r10

2: (07) r2 += -8

3: (b7) r1 = 1

4: (85) call 1

5: (15) if r0 == 0x0 goto pc+1

R0=map\_ptr R10=fp

6: (7a) \*(u64 \*)(r0 +4) = 0

misaligned access off 4 size 8

Program that correctly checks map\_lookup\_elem() returned value for NULL and

accesses memory with correct alignment in one side of 'if' branch, but fails

to do so in the other side of 'if' branch:

BPF\_ST\_MEM(BPF\_DW, BPF\_REG\_10, -8, 0),

BPF\_MOV64\_REG(BPF\_REG\_2, BPF\_REG\_10),

BPF\_ALU64\_IMM(BPF\_ADD, BPF\_REG\_2, -8),

BPF\_LD\_MAP\_FD(BPF\_REG\_1, 0),

BPF\_RAW\_INSN(BPF\_JMP | BPF\_CALL, 0, 0, 0, BPF\_FUNC\_map\_lookup\_elem),

BPF\_JMP\_IMM(BPF\_JEQ, BPF\_REG\_0, 0, 2),

BPF\_ST\_MEM(BPF\_DW, BPF\_REG\_0, 0, 0),

BPF\_EXIT\_INSN(),

BPF\_ST\_MEM(BPF\_DW, BPF\_REG\_0, 0, 1),

BPF\_EXIT\_INSN(),

Error:

0: (7a) \*(u64 \*)(r10 -8) = 0

1: (bf) r2 = r10

2: (07) r2 += -8

3: (b7) r1 = 1

4: (85) call 1

5: (15) if r0 == 0x0 goto pc+2

R0=map\_ptr R10=fp

6: (7a) \*(u64 \*)(r0 +0) = 0

7: (95) exit

from 5 to 8: R0=imm0 R10=fp

8: (7a) \*(u64 \*)(r0 +0) = 1

R0 invalid mem access 'imm'

Program that performs a socket lookup then sets the pointer to NULL without

checking it:

value:

BPF\_MOV64\_IMM(BPF\_REG\_2, 0),

BPF\_STX\_MEM(BPF\_W, BPF\_REG\_10, BPF\_REG\_2, -8),

BPF\_MOV64\_REG(BPF\_REG\_2, BPF\_REG\_10),

BPF\_ALU64\_IMM(BPF\_ADD, BPF\_REG\_2, -8),

BPF\_MOV64\_IMM(BPF\_REG\_3, 4),

BPF\_MOV64\_IMM(BPF\_REG\_4, 0),

BPF\_MOV64\_IMM(BPF\_REG\_5, 0),

BPF\_EMIT\_CALL(BPF\_FUNC\_sk\_lookup\_tcp),

BPF\_MOV64\_IMM(BPF\_REG\_0, 0),

BPF\_EXIT\_INSN(),

Error:

0: (b7) r2 = 0

1: (63) \*(u32 \*)(r10 -8) = r2

2: (bf) r2 = r10

3: (07) r2 += -8

4: (b7) r3 = 4

5: (b7) r4 = 0

6: (b7) r5 = 0

7: (85) call bpf\_sk\_lookup\_tcp#65

8: (b7) r0 = 0

9: (95) exit

Unreleased reference id=1, alloc\_insn=7

Program that performs a socket lookup but does not NULL-check the returned

value:

BPF\_MOV64\_IMM(BPF\_REG\_2, 0),

BPF\_STX\_MEM(BPF\_W, BPF\_REG\_10, BPF\_REG\_2, -8),

BPF\_MOV64\_REG(BPF\_REG\_2, BPF\_REG\_10),

BPF\_ALU64\_IMM(BPF\_ADD, BPF\_REG\_2, -8),

BPF\_MOV64\_IMM(BPF\_REG\_3, 4),

BPF\_MOV64\_IMM(BPF\_REG\_4, 0),

BPF\_MOV64\_IMM(BPF\_REG\_5, 0),

BPF\_EMIT\_CALL(BPF\_FUNC\_sk\_lookup\_tcp),

BPF\_EXIT\_INSN(),

Error:

0: (b7) r2 = 0

1: (63) \*(u32 \*)(r10 -8) = r2

2: (bf) r2 = r10

3: (07) r2 += -8

4: (b7) r3 = 4

5: (b7) r4 = 0

6: (b7) r5 = 0

7: (85) call bpf\_sk\_lookup\_tcp#65

8: (95) exit

Unreleased reference id=1, alloc\_insn=7

Testing

-------

Next to the BPF toolchain, the kernel also ships a test module that contains

various test cases for classic and internal BPF 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>