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 maps.rst) 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_
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.

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:

```
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
```