ISOVALENT
(now part of Cisco)

BPF: Indirect Calls

LSF/MM/BPF 2025

Anton Protopopov

Indirect calls in BPF

- Actually supported in LLVM since forever (2017), see a thread
- Require some changes from the kernel/libbpf side

Indirect calls in BPF: toy selftest

```
SEC("syscall") int simple_test(struct simple_ctx *ctx)
{
        __u64 (*foo)(__u64);

        if (ctx->x % 2)
                foo = &foo_1;
        else
                foo = &foo_2;

        ret_user = foo(ctx->x);

        return 0;
}
```

Indirect calls in BPF: *objdump -D*

```
00000000000 <simple_test>:
  0:        79 11 00 00 00 00 00 00 r1 = *(u64 *)(r1 + 0x0)
  1:        bf 13 00 00 00 00 00 00 r3 = r1
  2:        57 03 00 00 01 00 00 00 r3 &= 0x1
  3:        18 02 00 00 18 00 00 00 00 00 00 00 00 00 00 00 00 r2 = 0x18 ll
  5:        15 03 02 00 00 00 00 00 if r3 == 0x0 goto +0x2 <simple_test+0x40>
  6:        18 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 r2 = 0x0 ll
  8:        8d 02 00 00 00 00 00 00 callx r2
  9:        18 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 r1 = 0x0 ll
 11:        7b 01 00 00 00 00 00 00 *(u64 *)(r1 + 0x0) = r0
 12:        b4 00 00 00 00 00 00 00 w0 = 0x0
 13:        95 00 00 00 00 00 00 00 exit
```

Indirect calls in BPF: *libbpf*

- In *`bpf_object__relocate()`* libbpf does:

```
if (relo->type == RELO_SUBPROG_ADDR)
        insn[0].src_reg = BPF_PSEUDO_FUNC;
```

Indirect calls in BPF: *bpftool p d x*

```
int simple_test(struct simple_ctx * ctx):
   0: (79) r1 = *(u64 *)(r1 +0)
   1: (bf) r3 = r1
   2: (57) r3 &= 1
   3: (18) r2 = subprog[+10]
   5: (15) if r3 == 0x0 goto pc+2
   6: (18) r2 = subprog[+10]
   8: (8d) callx r2
   9: (18) r1 = map[id:6][0]+0
  11: (7b) *(u64 *)(r1 +0) = r0
  12: (b4) w0 = 0
  13: (95) exit
```

Indirect calls in BPF: *bpftool p d x*

```
int simple_test(struct simple_ctx * ctx):
   0: (79) r1 = *(u64 *)(r1 +0)
   1: (bf) r3 = r1
   2: (57) r3 &= 1
   3: (18) r2 = subprog[+10]
   5: (15) if r3 == 0x0 goto pc+2
   6: (18) r2 = subprog[+10]
   8: (8d) callx r2
   9: (18) r1 = map[id:6][0]+0
  11: (7b) *(u64 *)(r1 +0) = r0
  12: (b4) w0 = 0
  13: (95) exit
```

Indirect calls in BPF: *bpftool p d x opcodes*

```
2: (57) r3 &= 1
    57 03 00 00 01 00 00 00
3: (18) r2 = subprog[+10]
    18 42 00 00 0a 00 00 00 00 00 00 00 01 00 00 00
5: (15) if r3 == 0x0 goto pc+2
    15 03 02 00 00 00 00 00
6: (18) r2 = subprog[+10]
    18 42 00 00 0a 00 00 00 00 00 00 00 02 00 00 00
```

Indirect calls in BPF: *bpftool p d x opcodes*

LDIMM64 BPF_PSEUDO_FUNC

so, after the load
R2->type = PTR_TO_FUNC;
R2->subprogno = subprogno;

```
2: (57) r3 &= 1
    57 03 00 00 01 00 00 00
3: (18) r2 = subprog[+10]
    18 42 00 00 0a 00 00 00 00 00 00 00 01 00 00 00
5: (15) if r3 == 0x0 goto pc+2
    15 03 02 00 00 00 00 00
6: (18) r2 = subprog[+10]
    18 42 00 00 0a 00 00 00 00 00 00 00 02 00 00 00
```

Indirect calls in BPF: *bpftool p d x opcodes*

The offset is relative. This load points to sub-function 1, the next one to sub-function 2

```
2: (57) r3 &= 1
    57 03 00 00 01 00 00 00
3: (18) r2 = subprog[+10]
    18 42 00 00 0a 00 00 00 00 00 00 00 01 00 00 00
5: (15) if r3 == 0x0 goto pc+2
    15 03 02 00 00 00 00 00
6: (18) r2 = subprog[+10]
    18 42 00 00 0a 00 00 00 00 00 00 00 02 00 00 00
```

indirect calls: a more realistic example

```
struct calculon_stack stack = {
        .top = -1,
};
int (*op)(struct calculon_stack *);

for (i = 0; i < 1024 && i < ctx->n; i++) {
        key = i;

        x = bpf_map_lookup_elem(&calculon_input, &key);
        if (!x)
                break;

        if (isdigit(*x)) {
                if (push(&stack, *x - '0'))
                        return -1;
                continue;
        } else if (*x == '*') {
                op = calculon_mul;
        } else if (*x == '+') {
                op = calculon_add;
        } else if (*x == '-') {
                op = calculon_sub;
        } else {
                return -1;
        }

        if (op(&stack))
                return -1;
}
```

indirect calls: a more realistic example

```
check_calculon(skel, "1234567++++++", 28);
check_calculon(skel, "2222222******", 128);
check_calculon(skel, "34+6522*321+*+*+*222**16+39-41+4321+*+*+*+-3+", 11);
```

How to verify indirect calls

- Given that LDIMM64, src=PSEUDO_FUNC creates proper pointer, the only change required was*

```
+   } else if (BPF_SRC(insn->code) == BPF_X) {
+       struct bpf_reg_state *reg = &regs[insn->dst_reg];
+
+       if (reg->type != PTR_TO_FUNC)
+               return -EINVAL;
+
+       err = __check_func_call(env, insn, &insn_idx, reg->subprogno);
+   }
```

* the actual change, of course, is a bit bigger, and the piece of the patch above is edited to fit on the screen

What’s next

```
SEC("syscall") int table_test(struct table_ctx *ctx)
{
        __u64 (*foo[2])(__u64) = { &foo_1, &foo_2 };
        __u64 i = ctx->i, x = ctx->x;

        if (i >= ARRAY_SIZE(foo))
                return -22;

        ret_user = foo[i](x);

        return 0;
}
```

What’s next

```
SEC("syscall") int table_test(struct table_ctx *ctx)
{
        __u64 (*foo[2])(__u64) = { &foo_1, &foo_2 };
        __u64 i = ctx->i, x = ctx->x;

        if (i >= ARRAY_SIZE(foo))
                return -22;

        ret_user = foo[i](x);

        return 0;
}
```

What’s next

```
SEC("syscall") int table_test(struct table_ctx *ctx)
{
        __u64 (*foo[2])(__u64) = { &foo_1, &foo_2 };
        __u64 i = ctx->i, x = ctx->x;

        if (i >= ARRAY_SIZE(foo))
                return -22;

        ret_user = foo[i](x);

        return 0;
}
```

libbpf: relocation against STT_SECTION in non-exec section is not supported!

What’s next

```
SEC("syscall") int table_test(struct table_ctx *ctx)
{
        __u64 (*foo[2])(__u64);
        __u64 i = ctx->i, x = ctx->x;

        foo[0] = &foo_1;
        foo[1] = &foo_2;

        if (i >= ARRAY_SIZE(foo))
                return -22;

        ret_user = foo[i](x);

        return 0;
}
```

What's next

```
SEC("syscall") int table_test(struct table_ctx *ctx)
{
        __u64 (*foo[2])(__u64);
        __u64 i = ctx->i, x = ctx->x;

        foo[0] = &foo_1;
        foo[1] = &foo_2;

        if (i >= ARRAY_SIZE(foo))
                return -22;

        ret_user = foo[i](x);

        return 0;
}
```

Problem: on load from stack (or .bss, if foo declared globally), the register aux information is lost: which subfunction to verify?

What’s next

```
__u64 (*foo_table[2])(__u64) SEC("callx") = { &foo_1, &foo_2 };

SEC("syscall") int table_test(struct table_ctx *ctx)
{
        __u64 i = ctx->i, x = ctx->x;
        __u64 (*foo)(__u64);

        foo = foo_table[i];
        if (!foo)
                return -22;

        ret_user = foo(x);

        return 0;
}
```

What’s next

```
__u64 (*foo_table[2])(__u64) SEC("callx") = { &foo_1, &foo_2 };
```

What’s next

```
__u64 (*foo_table[2])(__u64) SEC("callx") = { &foo_1, &foo_2 };
```

Libbpf creates a map, and populates with [indexes] of sub-functions

```
struct {
        __uint(type, BPF_MAP_TYPE_INSN_SET);
        __uint(max_entries, 2);
        __type(key, u32);
        __type(value, u32);
        __ulong(map_extra, BPF_F_CALL_TABLE);
} foo_table SEC(".maps");
```

What's next

```
__u64 (*foo_table[2])(__u64) SEC("callx") = { &foo_1, &foo_2 };
```

Libbpf creates a map, and populates with [indexes] of sub-functions

```
struct {
        __uint(type, BPF_MAP_TYPE_INSN_SET);
        __uint(max_entries, 2);
        __type(key, u32);
        __type(value, u32);
        __ulong(map_extra, BPF_F_CALL_TABLE);
} foo_table SEC(".maps");
```

This flag tells kernel that this type of INSN_SET map contains only pointers to functions, and lookups should be dereferenced to addresses of functions

What’s next

```
SEC("syscall") int table_test(struct table_ctx *ctx)
{
        __u64 i = ctx->i, x = ctx->x;
        __u64 (*foo)(__u64);

        foo = bpf_map_lookup_elem(&foo_table, &i);
        if (!foo)
                return -22;

        ret_user = foo(x);

        return 0;
}
```

What's next

```
SEC("syscall") int table_test(struct table_ctx *ctx)
{
        __u64 i = ctx->i, x = ctx->x;
        __u64 (*foo)(__u64);

        foo = bpf_map_lookup_elem(&foo_table, &i);
        if (!foo)
                return -22;

        ret_user = foo(x);

        return 0;
}
```

Here foo is known to be a PTR_TO_FUNC and `reg(foo)->aux` keeps a ref to `foo_table`. Therefore, the verifier can validate all possible calls.

Indirect calls: questions

- Is this even ok to rely on **`LDIMM64[PSEUDO_FUNC]`**? Are there any potential problems with it? “Pseudo” part of the name looks suspicious. (Originally, it was added in 69c087ba62 (“bpf: Add bpf_for_each_map_elem() helper”.)
- Why does **`LDIMM64[PSEUDO_FUNC]`** only allow static functions (not global)?
- Need more use cases, are there real use cases?

Thanks!