



Adding KASAN support to eBPF

Alexis Lothoré

LPC 2026

© Copyright 2004-2026, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!





- ▶ Alexis Lothoré
- ▶ Living in Toulouse, France
- ▶ Linux engineer and trainer @ **Bootlin**
 - Engineering company specialized in **Embedded Linux** and **Zephyr**
 - 29 people, mostly in France
 - Very strong open-source focus
- ▶ Sponsored by the eBPF Foundation to work on multiple topics
 - selftests improvements
 - ARM64 support improvements
 - **KASAN support for eBPF programs**





Effort status

- ▶ Effort funded by the [Alpha-Omega](#) foundation, through the [eBPF Foundation](#)
- ▶ This talk is an update of the first talk presented at LSFMMBPF 2026 @ Zagreb, Croatia
 - The talk [slides](#) and a [LWN post](#) are available to catch up

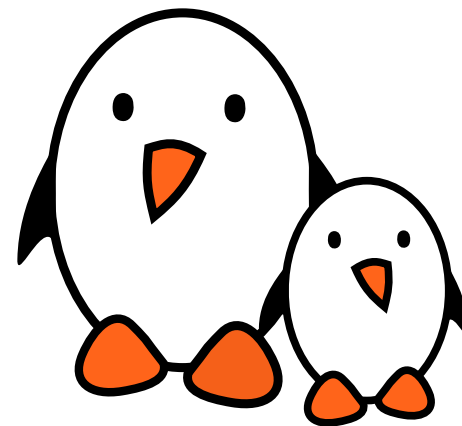


A very nice venue for the conference



KASAN basics

bootlin



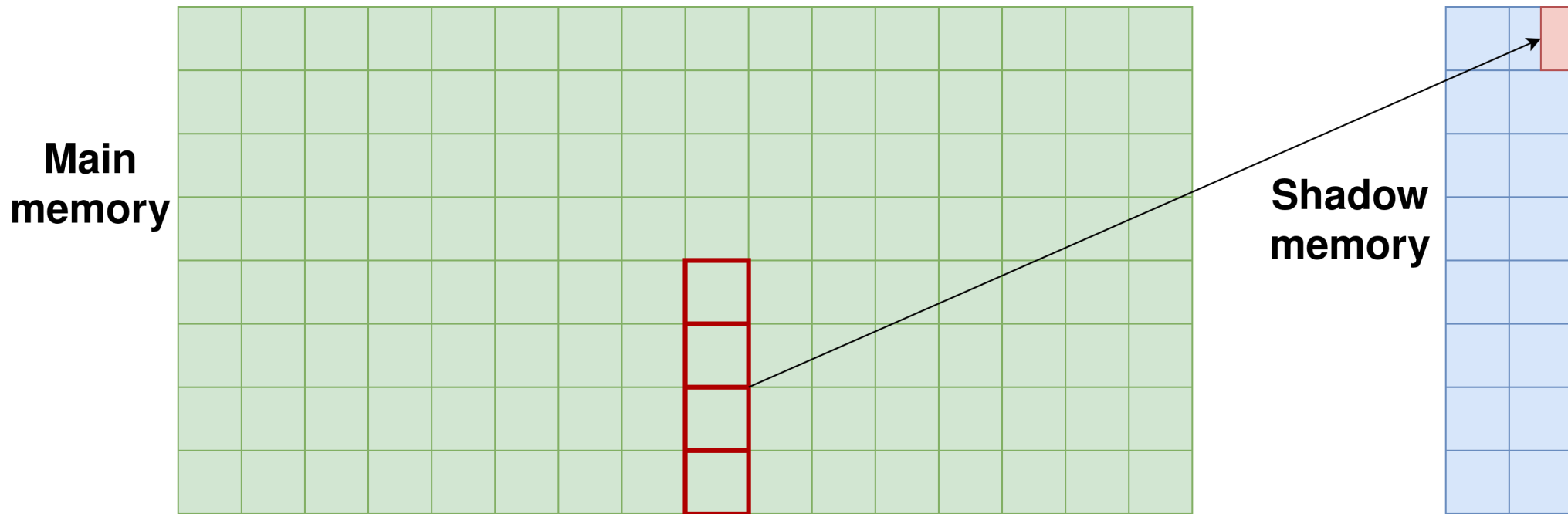


KASAN basics

- ▶ runtime memory access checker
- ▶ emits a report for any faulty access, with details about the access
- ▶ spots **U**se-**A**fter-**F**ree and **O**ut-**O**f-**B**ounds, on different kinds of memory (slab, stack, globals, vmalloc, ...)
- ▶ needs build-time instrumentation inserted by the compiler
 - non-negligible overhead both on image size and on performance
- ▶ needs runtime marking on memory
 - for generic KASAN, 1/8 of memory reserved for shadow memory
 - (lower for tag-based KASAN)



KASAN shadow memory



- ▶ one shadow byte encodes state for 8 bytes of memory (a “granule”)
- ▶ translation:

```
shadow = (addr >> KASAN_SHADOW_SCALE_SHIFT) + KASAN_SHADOW_OFFSET
```



KASAN compile-time instrumentation

No KASAN

```
endbr64
call    <__fentry__>
push    %rbx
mov     %rdi,%rbx
call    <ktime_get_mono_fast_ns>
mov     0x8(%rbx),%rdx
test    %rdx,%rdx
je      <bpf_check_timed_may_goto+0x31>
sub     %rdx,%rax
cmp     $0xee6b27f,%rax
ja      <bpf_check_timed_may_goto+0x37>
mov     $0xffff,%eax
pop     %rbx
jmp     <__pi__x86_return_thunk>
mov     %rax,0x8(%rbx)
jmp     <bpf_check_timed_may_goto+0x26>
call    <bpf_prog_report_may_goto_violation>
xor     %eax,%eax
pop     %rbx
jmp     <__pi__x86_return_thunk>
```



KASAN compile-time instrumentation

No KASAN

```
endbr64
call    <__fentry__>
push    %rbx
mov     %rdi,%rbx
call    <ktime_get_mono_fast_ns>
mov     0x8(%rbx),%rdx

test    %rdx,%rdx
je      <bpf_check_timed_may_goto+0x31>
sub     %rdx,%rax
cmp     $0xee6b27f,%rax
ja      <bpf_check_timed_may_goto+0x37>
mov     $0xffff,%eax
pop     %rbx
jmp     <__pi__x86_return_thunk>
mov     %rax,0x8(%rbx)
jmp     <bpf_check_timed_may_goto+0x26>
call    <bpf_prog_report_may_goto_violation>
xor     %eax,%eax
pop     %rbx
jmp     <__pi__x86_return_thunk>
```

Generic Inline KASAN

```
endbr64
call    <__fentry__>
push    %rbx
mov     %rdi,%rbx
sub     $0x8,%rsp
call    <ktime_get_mono_fast_ns>

lea     0x8(%rbx),%rdi
movabs  $0xdffffc0000000000,%rdx
mov     %rdi,%rcx
shr     $0x3,%rcx
cmpl    $0x0,(%rcx,%rdx,1)
jne     <bpf_check_timed_may_goto+0x63>
mov     0x8(%rbx),%rdx

test    %rdx,%rdx
je      <bpf_check_timed_may_goto+0x54>
sub     %rdx,%rax
cmp     $0xee6b27f,%rax
ja      <bpf_check_timed_may_goto+0x5a>
mov     $0xffff,%eax
add     $0x8,%rsp
pop     %rbx
jmp     <__pi__x86_return_thunk>
mov     %rax,0x8(%rbx)
jmp     <bpf_check_timed_may_goto+0x45>
call    <bpf_prog_report_may_goto_violation>
xor     %eax,%eax
jmp     <bpf_check_timed_may_goto+0x4a>

mov     %rax,(%rsp)
call    <__asan_report_load8_noabort>
mov     (%rsp),%rax

jmp     <bpf_check_timed_may_goto+0x31>
```



KASAN compile-time instrumentation

No KASAN

```
endbr64
call <__fentry__>
push %rbx
mov %rdi,%rbx
call <ktime_get_mono_fast_ns>
mov 0x8(%rbx),%rdx
test %rdx,%rdx
je <bpfc_check_timed_may_goto+0x31>
sub %rdx,%rax
cmp $0xee6b27f,%rax
ja <bpfc_check_timed_may_goto+0x37>
mov $0xffff,%eax
pop %rbx
jmp <__pi__x86_return_thunk>
mov %rax,0x8(%rbx)
jmp <bpfc_check_timed_may_goto+0x26>
call <bpfc_prog_report_may_goto_violation>
xor %eax,%eax
pop %rbx
jmp <__pi__x86_return_thunk>
```

Generic Inline KASAN

```
endbr64
call <__fentry__>
push %rbx
mov %rdi,%rbx
sub $0x8,%rsp
call <ktime_get_mono_fast_ns>
leaq 0x8(%rbx),%rdi
movabs $0xdffffc0000000000,%rdx
mov %rdi,%rcx
shrq $0x3,%rcx
cmpl $0x0,(%rcx,%rdx,1)
jne <bpfc_check_timed_may_goto+0x63>
mov 0x8(%rbx),%rdx
test %rdx,%rdx
je <bpfc_check_timed_may_goto+0x54>
sub %rdx,%rax
cmp $0xee6b27f,%rax
ja <bpfc_check_timed_may_goto+0x5a>
mov $0xffff,%eax
add $0x8,%rsp
pop %rbx
jmp <__pi__x86_return_thunk>
mov %rax,0x8(%rbx)
jmp <bpfc_check_timed_may_goto+0x45>
call <bpfc_prog_report_may_goto_violation>
xor %eax,%eax
jmp <bpfc_check_timed_may_goto+0x4a>
mov %rax,(%rsp)
call <__asan_report_load8_noabort>
mov (%rsp),%rax
jmp <bpfc_check_timed_may_goto+0x31>
```

Generic Outline KASAN

```
endbr64
call <__fentry__>
push %rbp
mov %rdi,%rbp
push %rbx
call <ktime_get_mono_fast_ns>
leaq 0x8(%rbp),%rdi
mov %rax,%rbx
call <__asan_load8>
mov 0x8(%rbp),%rax
test %rax,%rax
je <bpfc_check_timed_may_goto+0x40>
sub %rax,%rbx
cmp $0xee6b27f,%rbx
ja <bpfc_check_timed_may_goto+0x46>
mov $0xffff,%eax
pop %rbx
pop %rbp
jmp <__pi__x86_return_thunk>
mov %rbx,0x8(%rbp)
jmp <bpfc_check_timed_may_goto+0x34>
call <bpfc_prog_report_may_goto_violation>
xor %eax,%eax
jmp <bpfc_check_timed_may_goto+0x39>
```



KASAN at runtime

- ▶ on memory allocation: **unpoison** shadow bytes
- ▶ on memory deallocation: **poison** shadow bytes
- ▶ insert and maintain **red zones** *around* and *between* valid memory zones
- ▶ on access: **check** shadow bytes, if access is faulty, generate report



KASAN report

```
=====
BUG: KASAN: slab-out-of-bounds in kmalloc_oob_right+0xa8/0xbc [kasan_test]
Write of size 1 at addr ffff8801f44ec37b by task insmod/2760

CPU: 1 PID: 2760 Comm: insmod Not tainted 4.19.0-rc3+ #698
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.10.2-1 04/01/2014
Call Trace:
 dump_stack+0x94/0xd8
 print_address_description+0x73/0x280
 kasan_report+0x144/0x187
 __asan_report_store1_noabort+0x17/0x20
 kmalloc_oob_right+0xa8/0xbc [kasan_test]
 kmalloc_tests_init+0x16/0x700 [kasan_test]
 do_one_initcall+0xa5/0x3ae
 do_init_module+0x1b6/0x547
 load_module+0x75df/0x8070
 __do_sys_init_module+0x1c6/0x200
 __x64_sys_init_module+0x6e/0xb0
 do_syscall_64+0x9f/0x2c0
 entry_SYSCALL_64_after_hwframe+0x44/0xa9
RIP: 0033:0x7f96443109da
RSP: 002b:00007ffc0b51b08 EFLAGS: 00000202 ORIG_RAX: 00000000000000af
RAX: ffffffff00000000 RBX: 000055dc3ee521a0 RCX: 00007f96443109da
RDX: 00007f96445c0000 RSI: 0000000000000000 RDI: 00007f9644992000
RBP: 000055dc3ee510b0 R08: 0000000000000000 R09: 0000000000000000
R10: 00007f964430cd0a R11: 0000000000000202 R12: 00007f96445c0000
R13: 000055dc3ee51090 R14: 0000000000000000 R15: 0000000000000000

[...]
```

[...]

The buggy address belongs to the object at ffff8801f44ec300
which belongs to the cache kmalloc-128 of size 128
The buggy address is located 123 bytes inside of
128-byte region [ffff8801f44ec300, ffff8801f44ec380)
The buggy address belongs to the page:
page:ffffea0007d13b00 count:1 mapcount:0 mapping:ffff8801f7001640 index:0x0
flags: 0x200000000000100(slab)
raw: 0200000000000100 ffffea0007d11dc0 0000001a0000001a ffff8801f7001640
raw: 0000000000000000 0000000080150015 00000001ffffffff 0000000000000000
page dumped because: kasan: bad access detected

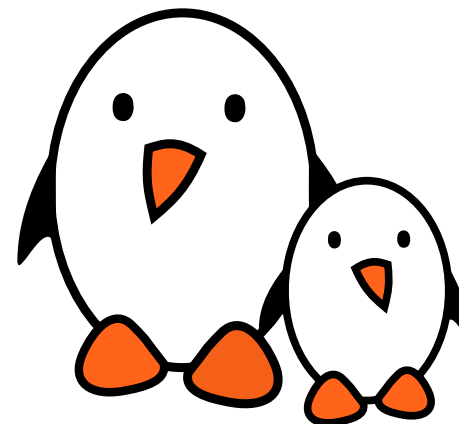
Memory state around the buggy address:

```
ffff8801f44ec200: fc fc fc fc fc fc fc fc fb fb fb fb fb fb fb
ffff8801f44ec280: fb fb fb fb fb fb fb fb fc fc fc fc fc fc fc fc
>ffff8801f44ec300: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03
                                                         ^
ffff8801f44ec380: fc fc fc fc fc fc fc fc fb fb fb fb fb fb fb fb
ffff8801f44ec400: fb fb fb fb fb fb fb fb fc fc fc fc fc fc fc fc
=====
```



KASAN and eBPF

bootlin





KASAN and eBPF

- ▶ Not all code is instrumented when KASAN is enabled
 - when loading eBPF programs with JIT enabled, the **kernel generates new native instructions**
 - those are **not** prefixed with KASAN instrumentation generated by GCC/Clang
- ▶ memory accesses performed by those new instructions are not monitored!
- ▶ this prevents potential bugs from being identified/caught early:
 - verifier can let a faulty access slip in
 - JIT compilers can make mistakes while translating instructions
 - programs can call into faulty kernel functions (e.g. BPF helpers, kfuncs)



Adding KASAN support for eBPF

Goal: replicate **at runtime** generic outline KASAN instrumentation

- ▶ JIT compiler goes over each eBPF instruction to translate (see `do_jit()` in `arch/x86/net/bpf_jit_comp.c`)
- ▶ if instruction is a load or store, emit an outline KASAN check just before it
- ▶ if a BPF program emits a KASAN splat, **this is a verifier/JIT compiler bug**
- ▶ the memory BPF programs are using is already tracked by some shadow memory!



Basic JIT algorithm

- ▶ in `do_jit()`, before each LDX/STX, add an `emit_kasan_check()` call:
 - identify access size (1, 2, 4, or 8 bytes)
 - identify read or write
 - derive the needed KASAN func:
`__asan_load1`, `__asan_store2`, etc
 - defined in `mm/kasan/generic.c`
 - save registers
 - emit call to KASAN checker
 - restore registers

[arch/x86/net/bpf_jit_comp.c](#)

```
[...]
case BPF_LOAD_ACQ:
    if (!accesses_stack_only) {
        err = emit_kasan_check(env, pprog, src_reg, insn->off,
                               BPF_SIZE(insn->code), ip, false);

        if (err)
            return err;
    }
    /* dst_reg = smp_load_acquire(src_reg + off16) */
    emit_ldx(pprog, BPF_SIZE(insn->code), dst_reg, src_reg,
             insn->off);
    break;
[...]
```



Changes since RFC

- ▶ all the points introduced above were part of the [original RFC](#)
- ▶ A few points have changed since then:
 - cover `BPF_ST` instructions
 - cover `BPF_ATOMIC` instructions
 - handle call depth accounting (retbleed mitigation)
 - handle JIT hardening (e.g. constant blinding)
 - refine stack accessing instructions tracking (quite a big topic, see next slide)



Challenge: handling correctly stack accessing instructions

- ▶ accesses on the stack must be ignored, but the JIT compiler has limited knowledge about the targeted memory
- ▶ the verifier has this knowledge, but after verification, it can rewrite plenty of instructions into new instructions (see `kernel/bpf/fixups.c`)
 - a one-to-one rewrite may or may not target the same kind of memory
 - for one-to-many rewrites:
 - the original instruction could be the first one in the new block
 - the original instruction could be the last one in the new block
 - the original instruction could be somewhere in the middle of the new block
 - the concept of “original instruction” may not even make sense in the new block!



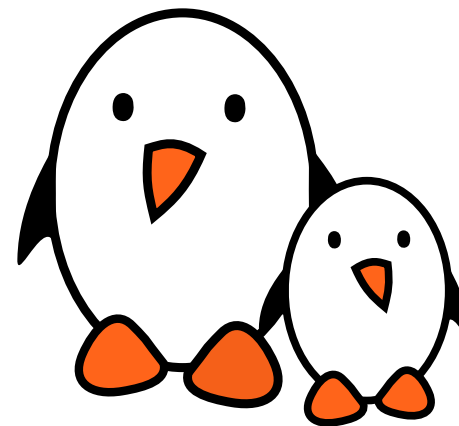
Challenge: handling correctly stack accessing instructions

- ▶ accesses on the stack must be ignored, but the JIT compiler has limited knowledge about the targeted memory
- ▶ the verifier has this knowledge, but after verification, it can rewrite plenty of instructions into new instructions (see `kernel/bpf/fixups.c`)
 - a one-to-one rewrite may or may not target the same kind of memory
 - for one-to-many rewrites:
 - the original instruction could be the first one in the new block
 - the original instruction could be the last one in the new block
 - the original instruction could be somewhere in the middle of the new block
 - the concept of “original instruction” may not even make sense in the new block!
- ▶ it is then pretty hard to reliably track whether an instruction targets the stack, so the implementation is **conservative: when not sure, add KASAN instrumentation**
 - the additional cost is a systematically passing KASAN check



Integrated feature

bootlin





Instrumented code

eBPF code

```
call -0x1
w1 = *(u8*)(r0 + 0x0)
w0 = 0x1
if w1 != 0x0 goto +0x1 <bpf_kasan_uaf+0x28>
w0 = 0x0
exit
```



Instrumented code

eBPF code

```
call -0x1
w1 = *(u8 *) (r0 + 0x0)
w0 = 0x1
if w1 != 0x0 goto +0x1 <bpf_kasan_uaf+0x28>
w0 = 0x0
exit
```

JITed instructions

```
[...]
endbr64
call 0xffffffffc04065d0
movzbq 0x0(%rax),%rdi
mov $0x1,%eax
test %edi,%edi
jne 0xffffffffc0000ad7
xor %eax,%eax
leave
ret
```



Instrumented code

eBPF code

```
call -0x1
w1 = *(u8 *) (r0 + 0x0)
w0 = 0x1
if w1 != 0x0 goto +0x1 <bpf_kasan_uaf+0x28>
w0 = 0x0
exit
```

JITed instructions

```
[...]
endbr64
call 0xffffffffc04065d0
movzbq 0x0(%rax),%rdi
mov $0x1,%eax
test %edi,%edi
jne 0xffffffffc0000ad7
xor %eax,%eax
leave
ret
```

JITed instructions with KASAN

```
[...]
endbr64
call 0xffffffffc04065d0
push %rax
push %rcx
push %rdx
push %rdi
push %rsi
push %r8
push %r9
push %r10
push %r11
sub $0x8,%rsp
mov %rax,%rdi
call 0xffffffff81da80a0 <__asan_load1>
add $0x8,%rsp
pop %r11
pop %r10
pop %r9
pop %r8
pop %rsi
pop %rdi
pop %rdx
pop %rcx
pop %rax
movzbq 0x0(%rax),%rdi
mov $0x1,%eax
test %edi,%edi
jne 0xffffffffc0000ad7
xor %eax,%eax
leave
ret
```



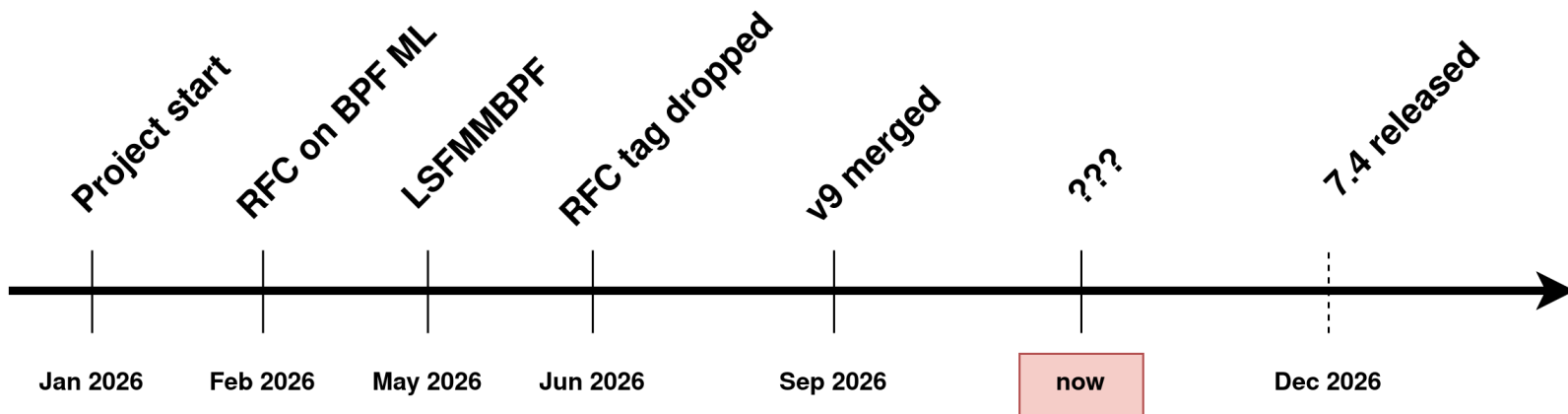
eBPF KASAN reports

```
[ 2217.398190] =====
[ 2217.398606] BUG: KASAN: slab-use-after-free in bpf_prog_dac77fbc04532208_bpf_kasan_uaf+0x2e/0x4d
[ 2217.398622] Read of size 1 at addr ffff888111b8f500 by task test_progs/199
[ 2217.398622]
[ 2217.398622] CPU: 3 UID: 0 PID: 199 Comm: test_progs Tainted: G    B      OE      7.0.0-kasan+ #202 PREEMPT(full)
[ 2217.398622] Tainted: [B]=BAD_PAGE, [O]=OOT_MODULE, [E]=UNSIGNED_MODULE
[ 2217.398622] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS Arch Linux 1.17.0-2-2 04/01/2014
[ 2217.398622] Call Trace:
[ 2217.398622] <TASK>
[ 2217.398622] dump_stack_lvl+0x5d/0x80
[ 2217.398622] print_report+0x153/0x4c6
[ 2217.398622] ? bpf_prog_dac77fbc04532208_bpf_kasan_uaf+0x2e/0x4d
[ 2217.398622] ? __virt_addr_valid+0x225/0x4c0
[ 2217.398622] ? bpf_prog_dac77fbc04532208_bpf_kasan_uaf+0x2e/0x4d
[ 2217.398622] kasan_report+0xe4/0x1b0
[ 2217.398622] ? bpf_prog_dac77fbc04532208_bpf_kasan_uaf+0x2e/0x4d
[ 2217.398622] bpf_prog_dac77fbc04532208_bpf_kasan_uaf+0x2e/0x4d
[ 2217.398622] ? trace_hardirqs_on+0x53/0x1a0
[ 2217.398622] ? kasan_quarantine_put+0xdd/0x220
[ 2217.398622] bpf_test_run+0x376/0x910
[... ]
[ 2217.398622] Allocated by task 199:
[ 2217.398622] kasan_save_stack+0x30/0x50
[ 2217.398622] kasan_save_track+0x14/0x30
[ 2217.398622] __kasan_kmalloc+0x7f/0x90
[ 2217.398622] bpf_kfunc_kasan_uaf_1+0x3b/0x80 [bpf_testmod]
[ 2217.398622] bpf_prog_dac77fbc04532208_bpf_kasan_uaf+0x19/0x4d
[ 2217.398622] bpf_test_run+0x376/0x910
[ 2217.398622] bpf_prog_test_run_skb+0x1247/0x3060
[ 2217.398622] __sys_bpf+0xf3e/0x5100
[... ]
```



Project timeline

- ▶ the feature is now merged, and should be part of the 7.4 release cycle
- ▶ it is already running in BPF CI (and the [splat_detector](#) introduced by [Mykyta Yatsenko](#) should catch any KASAN error)





KASAN splats in CI

Summary

All jobs

set-matrix

x86_64 gcc-15

build kernel and selftests

build-release

test_progs on x86_64 with gcc-15

test_progs_parallel on x86_64 with gcc-15

test_progs_no_alu32 on x86_64 with gcc-15

test_progs_no_alu32_parallel on x86_64 with gcc-15

test_verifier on x86_64 with gcc-15

test_maps on x86_64 with gcc-15

test_progs_cpu_v4 on x86_64 with gcc-15

build test_progs with ASAN

test_progs ASAN

veristat-kernel

veristat-meta

build sched-ext/scx

veristat-scx

veristat-cilium

GCC BPF build

GCC BPF

x86_64 llvm-21

aarch64 gcc-15

s390x gcc-15

x86_64 gcc-15 / test (test_progs, false, 360) / test_progs on x86_64 with gcc-15

succeeded 28 minutes ago in 9m 17s

Search logs

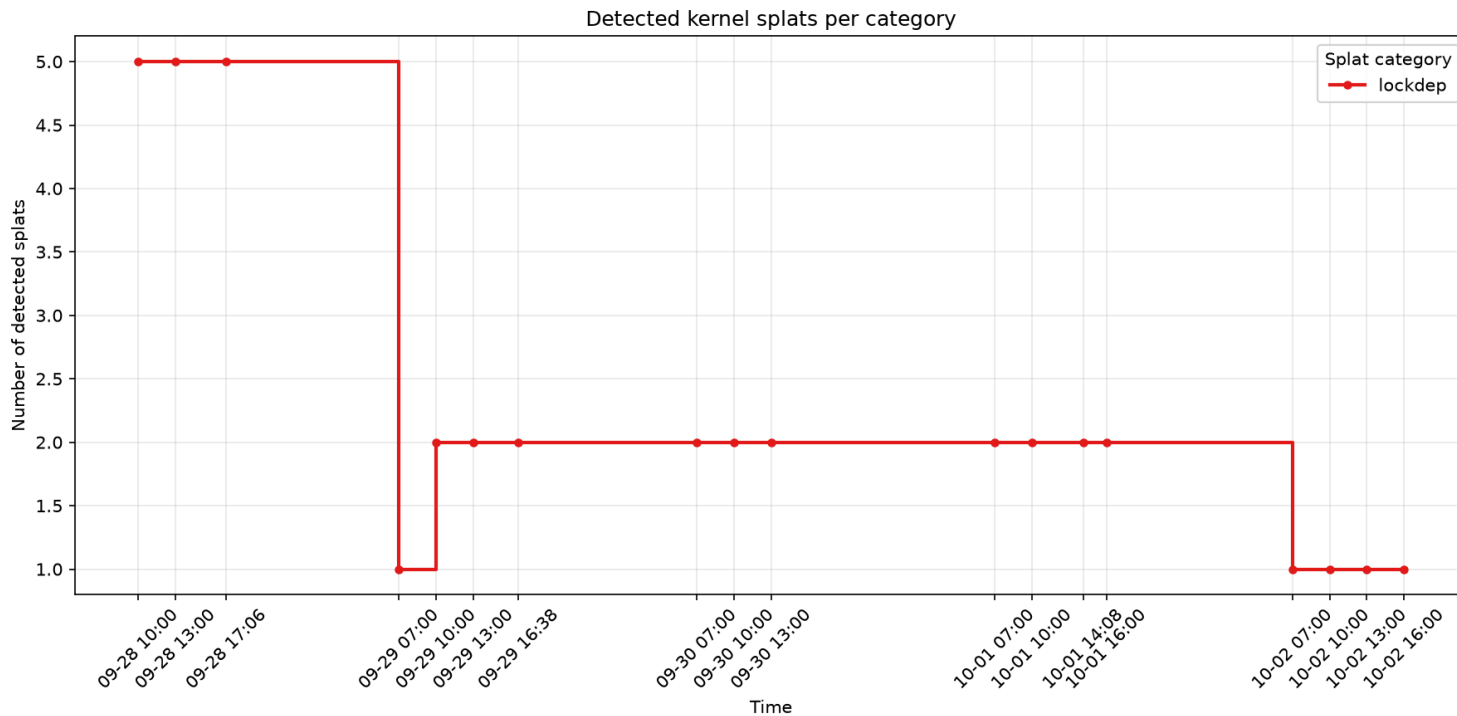
Run selftests 8m 32s

```
#185/1 jeq_infer_not_null/jeq_infer_not_null_ptr_to_btfd:OK
#185 jeq_infer_not_null:OK
#186 jit_probe_mem:OK
[ 91.580394] =====
[ 91.580399] BUG: KASAN: slab-use-after-free in bpf_prog_319a599a10235764_stx_not_on_stack+0x1d5/0x253
[ 91.580405] Write of size 1 at addr ffff88810d118a00 by task test_progs/128
[ 91.580408]
[ 91.580412] CPU: 1 UID: 0 PID: 128 Comm: test_progs Tainted: G          OE      7.3.0-rc4-gef85b4e8c9c9-dirty #1 PREEMPT(full)
[ 91.580417] Tainted: [O]=OOT_MODULE, [E]=UNSIGNED_MODULE
[ 91.580418] Hardware name: QEMU Ubuntu 24.04 PC v2 (i440FX + PIIX, arch_caps fix, 1996), BIOS 1.16.3-debian-1.16.3-2 04/01/2014
[ 91.580421] Call Trace:
[ 91.580422] <TASK>
[ 91.580424] dump_stack_lvl+0x5d/0x80
[ 91.580430] print_report+0x153/0x4c6
[ 91.580435] ? bpf_prog_319a599a10235764_stx_not_on_stack+0x1d5/0x253
[ 91.580436] ? __virt_addr_valid+0x226/0x4e0
[ 91.580441] ? bpf_prog_319a599a10235764_stx_not_on_stack+0x1d5/0x253
[ 91.580442] kasan_report+0xe4/0x1a0
[ 91.580447] ? bpf_prog_319a599a10235764_stx_not_on_stack+0x1d5/0x253
[ 91.580450] bpf_prog_319a599a10235764_stx_not_on_stack+0x1d5/0x253
#187/1 kasan/st_1_not_on_stack:SKIP
#187/2 kasan/st_1_on_stack:SKIP
#187/3 kasan/st_2_not_on_stack:SKIP
#187/4 kasan/st_2_on_stack:SKIP
#187/5 kasan/st_4_not_on_stack:SKIP
#187/6 kasan/st_4_on_stack:SKIP
#187/7 kasan/st_8_not_on_stack:SKIP
#187/8 kasan/st_8_on_stack:SKIP
#187/9 kasan/stx_1_not_on_stack:OK
#187/10 kasan/stx_1_on_stack:OK
#187/11 kasan/stx_2_not_on_stack:OK
#187/12 kasan/stx_2_on_stack:OK
#187/13 kasan/stx_4_not_on_stack:OK
#187/14 kasan/stx_4_on_stack:OK
#187/15 kasan/stx_8_not_on_stack:OK
#187/16 kasan/stx_8_on_stack:OK
#187/17 kasan/ldx_1_not_on_stack:OK
#187/18 kasan/ldx_1_on_stack:OK
#187/19 kasan/ldx_2_not_on_stack:OK
#187/20 kasan/ldx_2_on_stack:OK
```



Initial results

Good (or bad?) news: no KASAN splat emitted by any BPF program in CI so far



kernel splats sorted by type on a few days, on any pending series



Next steps

- ▶ Did maintainers catch some KASAN splats elsewhere?
- ▶ Does the feature cause a non-negligeable performance issue in CI? If so, there are potential optimizations to consider:
 - inline instrumentation ?
 - track register liveness to save/restore only used registers ?
- ▶ Any undesired side effect observed in the wild?

Let's discuss it!

Slides under CC-BY-SA 3.0

<https://bootlin.com/pub/conferences/>

alexis.lothore@bootlin.com