Adding KASAN support to eBPF

Alexis Lothoré

LSFMMBPF 2026

whoami

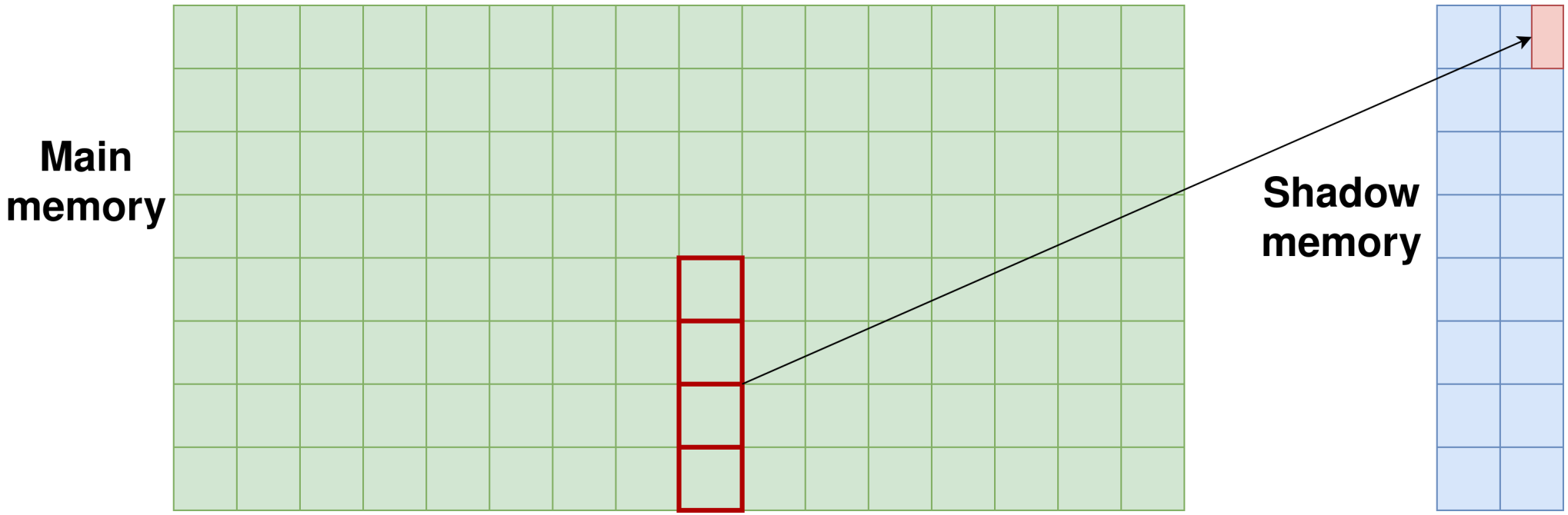
- Alexis Lothoré
- Living in Toulouse, France
- Linux engineer and trainer @ Bootlin
  - Engineering company specialized in **Embedded Linux** and **Zephyr**
  - 28 people, mostly in France
  - Very strong open-source focus
- Sponsored by the eBPF foundation to work on multiple topics
  - selftests improvements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

KASAN basics

KASAN basics

- runtime memory accesses 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-negligeable 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 build-time instrumentation (1/2)

kernel/bpf/core.c

```
u64 bpf_check_timed_may_goto(struct bpf_timed_may_goto *p)
{
        u64 time = ktime_get_mono_fast_ns();

        /* Populate the timestamp for this stack frame, and refresh count. */
        if (!p->timestamp) {
                p->timestamp = time;
                return BPF_MAX_TIMED_LOOPS;
        }
        /* Check if we've exhausted our time slice, and zero count. */
        if (unlikely(time - p->timestamp >= (NSEC_PER_SEC / 4))) {
                bpf_prog_report_may_goto_violation();
                return 0;
        }
        /* Refresh the count for the stack frame. */
        return BPF_MAX_TIMED_LOOPS;
}
```

include/linux/filter.h

```
struct bpf_timed_may_goto {
        u64 count;
        u64 timestamp;
};
```

KASAN compile-time instrumentation (2/2)

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 (2/2)

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 Outline KASAN

```
endbr64
call   <__fentry__>
push   %rbp
mov    %rdi,%rbp
push   %rbx
call   <ktime_get_mono_fast_ns>
lea    0x8(%rbp),%rdi
mov    %rax,%rbx
call   <__asan_load8>
mov    0x8(%rbp),%rax
test   %rax,%rax
je     <bpf_check_timed_may_goto+0x40>
sub    %rax,%rbx
cmp    $0xee6b27f,%rbx
ja     <bpf_check_timed_may_goto+0x46>
mov    $0xffff,%eax
pop    %rbx
pop    %rbp
jmp    <__pi___x86_return_thunk>
mov    %rbx,0x8(%rbp)
jmp    <bpf_check_timed_may_goto+0x34>
call   <bpf_prog_report_may_goto_violation>
xor    %eax,%eax
jmp    <bpf_check_timed_may_goto+0x39>
```

KASAN compile-time instrumentation (2/2)

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 Outline KASAN

```
endbr64
call   <__fentry__>
push   %rbp
mov    %rdi,%rbp
push   %rbx
call   <ktime_get_mono_fast_ns>
lea    0x8(%rbp),%rdi
mov    %rax,%rbx
call   <__asan_load8>
mov    0x8(%rbp),%rax
test   %rax,%rax
je     <bpf_check_timed_may_goto+0x40>
sub    %rax,%rbx
cmp    $0xee6b27f,%rbx
ja     <bpf_check_timed_may_goto+0x46>
mov    $0xffff,%eax
pop    %rbx
pop    %rbp
jmp    <__pi___x86_return_thunk>
mov    %rbx,0x8(%rbp)
jmp    <bpf_check_timed_may_goto+0x34>
call   <bpf_prog_report_may_goto_violation>
xor    %eax,%eax
jmp    <bpf_check_timed_may_goto+0x39>
```

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
cmpb   $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 at runtime

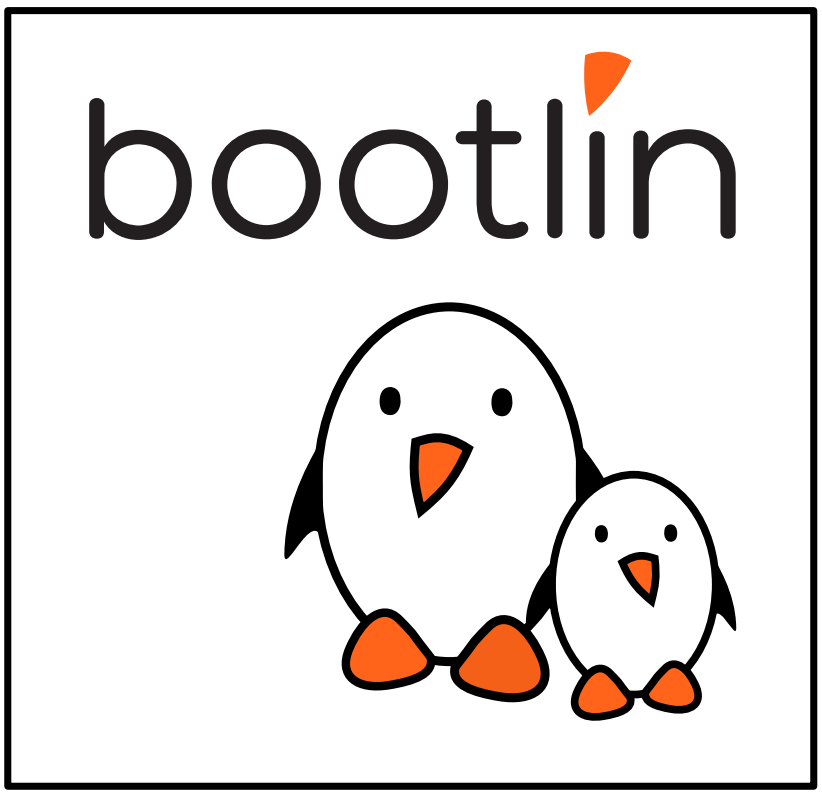
- on memory allocation: “unpoison” corresponding shadow bytes
- on memory deallocation: “poison” corresponding shadow bytes
- insert and maintain “red zones” around and between valid memory zones
- on access: check shadow bytes, if access is faulty, generate report

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:00007ffcf0b51b08 EFLAGS: 00000202 ORIG_RAX: 00000000000000af
RAX: ffffffffffffffda RBX: 000055dc3ee521a0 RCX: 00007f96443109da
RDX: 00007f96445cff88 RSI: 0000000000057a50 RDI: 00007f9644992000
RBP: 000055dc3ee510b0 R08: 0000000000000003 R09: 0000000000000000
R10: 00007f964430cd0a R11: 0000000000000202 R12: 00007f96445cff88
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 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



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 (eg 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

Shadow memory for eBPF programs

Most of the memory we want to monitor is already tracked by some shadow memory:

- (most) maps are kmalloc or vmalloc: ✅
  - arenas are handled differently, see Emil Tsalapatis' work and presentation tomorrow morning)
- global variables are exposed through array maps: ✅
- stack memory: generally thread stack memory: ≈ ✅
  - the risk is rather about OOB
  - we already have CONFIG_VMAP_STACK which protect the *overall* stack
  - a finer KASAN tracking could be set (meaning: per variable on stack), but it then needs red zones around each variable: complexify *a lot* JIT compilers work

Initial RFC

bpf.vger.kernel.org archive mirror

search help / color / mirror / Atom feed

From: "Alexis Lothoré" <alexis.lothore@bootlin.com>
To: "bpf" <bpf@vger.kernel.org>
Cc: "Alexei Starovoitov" <ast@kernel.org>,
"Thomas Petazzoni" <thomas.petazzoni@bootlin.com>,
"Bastien Curutchet (eBPF Foundation)"
<bastien.curutchet@bootlin.com>,
"Emil Tsalapatis" <emil@etsalapatis.com>,
"Daniel Borkmann" <daniel@iogearbox.net>
Subject: [RFC] adding KASAN support to JIT compiler(s)
Date: Fri, 06 Feb 2026 12:31:34 +0100 [thread overview]
Message-ID: <DG7UG112AVBC.JKYISDTAM30T@bootlin.com> (raw)

Hi everyone,

I am starting to work on adding KASAN support for loaded programs in the kernel (in the context of the sponsorship granted to the eBPF Foundation by the Alpha-Omega project, see [0]). I have done a bit of research/experiments, but before actually writing and sending things upstream, I'm sharing here what I understood on the requirement overall, and how I am currently considering to implement it. Please feel free to correct or enrich any part below!

Brief

Once a program has passed the verifier (and have been JIted into native code), we have strong guarantees that its _behavior_ will not trigger bugs like infinite loops, invalid access on input context, NULL pointer dereference, and so on. However, eBPF programs can still generate invalid memory accesses in some scenarios:
- programs validated behavior relies on the hypothesis that the tools (helpers, kfuncs) and data (eg kernel data) provided to eBPF programs are sane: eg if a pointer returned by the kernel to a program is somehow bogus (used after free, triggering out of bounds accesses, etc), a bpf program that has been validated by the verifier will happily use this pointer.
- there could be a bug in the verifier and/or the JIT compiler, letting invalid accesses slip through

https://lore.kernel.org/bpf/DG7UG112AVBC.JKYISDTAM30T@bootlin.com/

- Focus on x86_64
- Focus on most common load/store instructions (LDX, STX)
- Discards stack monitoring

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 needed kasan func: `__asan_load1`, `__asan_store2`, etc
    - defined in mm/kasan/kasan.h
  - save registers
  - realign stack
  - emit call to KASAN checker
  - restore registers

arch/x86/net/bpf_jit_comp.c

```
[...]
case BPF_STX | BPF_MEM | BPF_B:
case BPF_STX | BPF_MEM | BPF_H:
case BPF_STX | BPF_MEM | BPF_W:
case BPF_STX | BPF_MEM | BPF_DW:
        err = emit_kasan_check(&prog, dst_reg, insn,
                               ip, accesses_stack_only);
        if (err)
                return err;
        emit_stx(&prog, BPF_SIZE(insn->code), dst_reg,
                 src_reg, insn->off);
        break;
[...]
```

Skipping stack accesses monitoring

- accesses on stack must be ignored, but JIT compiler does not know about stack accesses, only about basic instruction info
- verifier does know about it though
- recent work from Xu Kuohai allows passing the verifier env to JIT compilers
  - see [PATCH bpf v15 0/5] emit ENDBR/BTI instructions for indirect
- for KASAN, we can then:
  - flag instructions accessing the stack in the verifier (through insn_aux_data)
  - check this flag in JIT compiler
  - emit the KASAN check only if relevant

Proof Of Concept

bpf.vger.kernel.org archive mirror

search help / color / mirror / Atom feed

From: "Alexis Lothoré (eBPF Foundation)" <alexis.lothore@bootlin.com>
To: Alexei Starovoitov <ast@kernel.org>,
Daniel Borkmann <daniel@iogearbox.net>,
Andrii Nakryiko <andrii@kernel.org>,
[...]
Cc: ebpf@linuxfoundation.org,
"Bastien Curutchet" <bastien.curutchet@bootlin.com>,
"Thomas Petazzoni" <thomas.petazzoni@bootlin.com>,
"Xu Kuohai" <xukuohai@huawei.com>,
bpf@vger.kernel.org, linux-kernel@vger.kernel.org,
netdev@vger.kernel.org, linux-kselftest@vger.kernel.org,
linux-stm32@st-md-mailman.stormreply.com,
linux-arm-kernel@lists.infradead.org, kasan-dev@googlegroups.com,
linux-mm@kvack.org,
"Alexis Lothoré (eBPF Foundation)" <alexis.lothore@bootlin.com>
Subject: [PATCH RFC bpf-next 0/8] bpf: add support for KASAN checks in JITed programs
Date: Mon, 13 Apr 2026 20:28:40 +0200 [thread overview]
Message-ID: <20260413-kasan-v1-0-1a5831230821@bootlin.com> (raw)

Hello,
this series aims to bring basic support for KASAN checks to BPF JITed
programs. This follows the first RFC posted in [1].

KASAN allows to spot memory management mistakes by reserving a fraction
of memory as "shadow memory" that will map to the rest of the memory and
allow its monitoring. Each memory-accessing instruction is then
instrumented at build time to call some ASAN check function, that will
analyze the corresponding bits in shadow memory, and if it detects the
access as invalid, trigger a detailed report. The goal of this series is
to replicate this mechanism for BPF programs when they are being JITed
into native instructions: that's then the (runtime) JIT compiler who is
in charge of inserting calls to the corresponding kasan checks, when a
program is being loaded into the kernel. This task involves:
- identifying at program load time the instructions performing memory
accesses
- identifying those accesses properties (size ? read or write ?) to
define the relevant kasan check function to call
- just before the identified instructions:
- perform the basic context saving (ie: saving registers)
- inserting a call to the relevant kasan check function
- restore context
- whenever the instrumented program executes, if it performs an invalid
access, it triggers a kasan report identical to those instrumented on
kernel side at build time.

https://lore.kernel.org/bpf/20260413-kasan-v1-0-1a5831230821@bootlin.com/

- RFC patch series posted a few weeks ago
- Implements a few tests showing JITed KASAN in action
- Plenty of small (and not-so-small) issues to fix

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
nopl   0x0(%rax,%rax,1)
nopl   (%rax)
push   %rbp
mov    %rsp,%rbp
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
nopl   0x0(%rax,%rax,1)
nopl   (%rax)
push   %rbp
mov    %rsp,%rbp
endbr64
call   0xffffffffc04065d0
movzbq 0x0(%rax),%rdi
mov    $0x1,%eax
test   %edi,%edi
jne    0xffffffffc0000ad7
xor    %eax,%eax
leave
ret
```

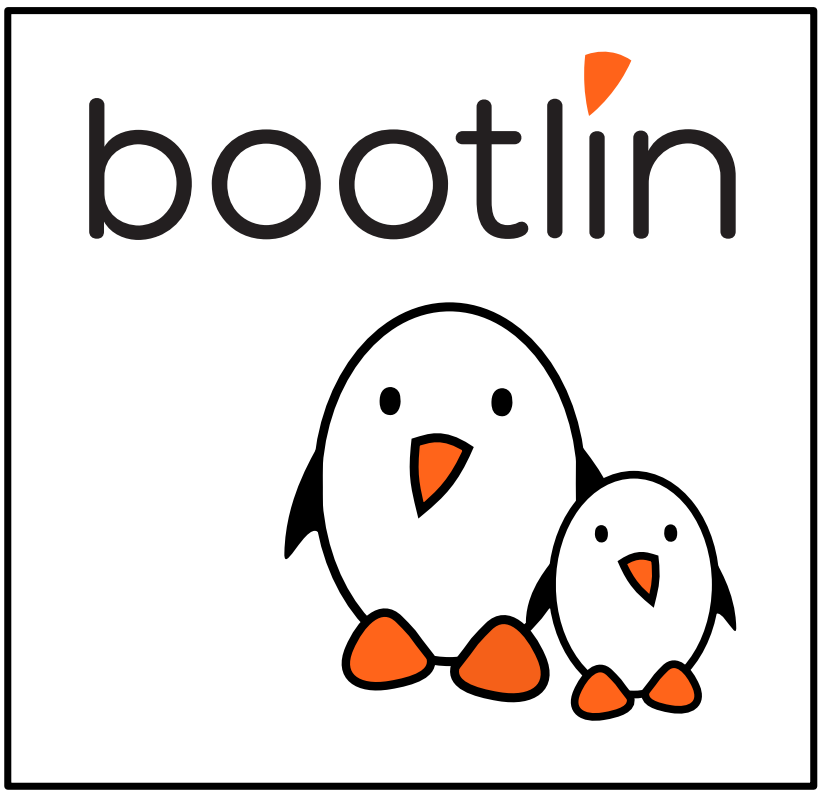
JITed instructions with KASAN

```
endbr64
nopl   0x0(%rax,%rax,1)
nopl   (%rax)
push   %rbp
mov    %rsp,%rbp
endbr64
call   0xffffffffc04065d0
push   %rax
push   %rcx
push   %rdx
push   %rdi
push   %rsi
push   %r8
push   %r9
sub    $0x8,%rsp
mov    %rax,%rdi
call   0xffffffff81da80a0 <__asan_load1>
add    $0x8,%rsp
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
[...]
```

Next steps and issues to solve



Covered instructions

- implementation currently only covers basic LDX/STX
- there are other cases like `BPF_PROBE_MEM`, `BPF_PROBE_ATOMIC` and `BPF_ATOMIC`
  - `BPF_PROBE_MEM` are likely not to be covered: used addresses can fault (and so need specific exception handling) and so they would make KASAN checks fault as well
  - same for `BPF_PROBE_ATOMIC`
  - `BPF_ATOMIC` are currently not covered but they need to
    - a single BPF instruction can lead to a complex set of translated instruction
    - there is no 1-to-1 mapping (eg a `BPF_STX|BPF_ATOMIC` can be a Read-Modify-Write operation, thus writing then reading memory)
- special care must be taken for instrumentation accuracy: some other JIT compiler features can slightly offset generated instructions, making KASAN completely wrong.

Stack access tracking

- smarter stack accessing instructions tracking: current solution can break in many ways:
  - verifier fixups and patches break offsets in `insn_aux_data`
  - pointer types can change between prog and subprogs (eg: a `PTR_TO_STACK` becomes a `PTR_TO_MEM` in a global subprog)
  - different verifier states can lead to different pointer types for the same instruction
- can lead to many instructions being wrongly instrumented or not instrumented when they should

Testing

- current testing is more of a showcase than actual thorough testing:
  - makes ebpf programs trigger faulty read and writes on OoB and UaF thanks to some buggy testing kfuncs
  - read tests really are OOB and UAF, write tests just “hack” shadow memory to trigger the KASAN reports
  - tests validate presence of the expected KASAN report in kernel logs
- Needed work:
  - extend coverage, especially for corner cases (eg stack accessing)
  - potentially two distinct testing scales:
    1. validate emitted instrumentation
    2. validate capability to trigger KASAN reports
  - how far should it go ?
    - if 1, `test_loader` style, checking any generated instruction ? (sounds fragile)
    - is faking poisoned shadow memory enough ?

Performance

- The current implementation exposes a **Dumb** **Generic** **Outline** KASAN instrumentation
  - *Dumb*: plenty of registers are preserved (could benefit from register usage tracking)
  - *Generic*: pure software KASAN (on ARM64, we could use SW or HW tags based KASAN)
  - *Outline*: we emit function calls
- no measure yet of the impact of this (work-in-progress) instrumentation
- Some performance metrics could be interesting:
  - what's the overall runtime overhead, eg when running selftests ?
  - what about some inline instrumentation ?

Let's discuss it !

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