

Debugging runtime-generated code in the Linux kernel

Alexis Lothoré

OSSEU 2026

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





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
- Working on various eBPF topics
  - selftests improvements
  - ARM64 support improvements
  - KASAN support for eBPF programs
- Teaching a Linux debugging training course





Why this talk

- sometimes you have to deal with tricky bugs for which ~~caveman debugging~~ print-debugging isn't enough, eg **runtime-generated code**, **at non-static location**
- knowing how to debug your system with a debugger then becomes a superpower

- This talk is a walkthrough guide about building a basic debugging setup, using the opportunity to share (*what I assume to be*) some cool tricks.
- This isn't bleeding edge tech, just standard tooling



Why this talk

- sometimes you have to deal with tricky bugs for which ~~caveman debugging~~ print-debugging isn't enough, eg **runtime-generated code**, **at non-static location**
- knowing how to debug your system with a debugger then becomes a superpower

- This talk is a walkthrough guide about building a basic debugging setup, using the opportunity to share (*what I assume to be*) some cool tricks.
- This isn't bleeding edge tech, just standard tooling

Bonus: you will be able to make your console look like it comes from the Matrix



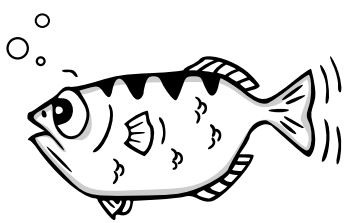
Linux Debugging basics





A GDB primer

- The official GNU debugger
- The standard interface is a CLI; a bit rough, but gives you superpowers:
  - controlling execution: `run`, `continue`, `next`, `step`, `stepi`...
  - break at precise places: `break`, `watch`, conditions...
  - memory *monitoring* and *alteration*: `print`, `x`...
- Can be used with a frontend, but knowing how the CLI works is highly beneficial
- Not the only debugger available, see LLDB



https://sourceware.org/gdb/



Warm-up: Userspace debugging

Local debugging

```
$ gdb my_prog
(gdb) break my_func
(gdb) run
# program runs until my_func
(gdb) print foo
```



Warm-up: Userspace debugging

Local debugging

```
$ gdb my_prog
(gdb) break my_func
(gdb) run
# program runs until my_func
(gdb) print foo
```

Remote debugging

```
$ gdbserver :3333 my_prog
```

```
$ gdb my_prog
(gdb) target remote 127.0.0.1:3333
(gdb) break my_func
(gdb) continue
# program runs until my_func
(gdb) print foo
```

Note: on Linux, GDB is based on the ptrace() system call



The real deal: kernel debugging

- How to *"debug the branch you are sitting on"* ?
- Multiple choices:



The real deal: kernel debugging

- How to *"debug the branch you are sitting on"* ?
- Multiple choices:
  - (*add printk*, *rebuild*, *re-deploy*, *re-boot*, *check logs*, *rinse*, *repeat*)



The real deal: kernel debugging

- How to *"debug the branch you are sitting on"* ?
- Multiple choices:
  - (*add printk, rebuild, re-deploy, re-boot, check logs, rinse, repeat*)
  - make your kernel act as the debug server (KGDB), but it comes with limitations:
    - breakpoints denylist
    - can't interrupt the kernel with Ctrl+C
    - **arch support**
    - ...



The real deal: kernel debugging

- How to *"debug the branch you are sitting on"* ?
- Multiple choices:
  - (*add printk, rebuild, re-deploy, re-boot, check logs, rinse, repeat*)
  - make your kernel act as the debug server (KGDB), but it comes with limitations:
    - breakpoints denylist
    - can't interrupt the kernel with Ctrl+C
    - **arch support**
    - ...
  - use a hardware probe (JTAG/SWD)
    - needs the corresponding interface to be exposed
    - can be expensive



The real deal: kernel debugging

- How to *"debug the branch you are sitting on"* ?
- Multiple choices:
  - (*add printk, rebuild, re-deploy, re-boot, check logs, rinse, repeat*)
  - make your kernel act as the debug server (KGDB), but it comes with limitations:
    - breakpoints denylist
    - can't interrupt the kernel with Ctrl+C
    - **arch support**
    - ...
  - use a hardware probe (JTAG/SWD)
    - needs the corresponding interface to be exposed
    - can be expensive
  - virtualize your environment
    - eg: QEMU in system mode
    - the emulator acts as a debug server
    - works very well if you don't depend on very specific hardware emulation



Virtualized kernel

- Let's assume we want to work on an x86_64 kernel:

```
$ qemu-system-x86 -m 4G -kernel <your_bzImage>
```

- and voila, you have a running kernel ready for debugging !



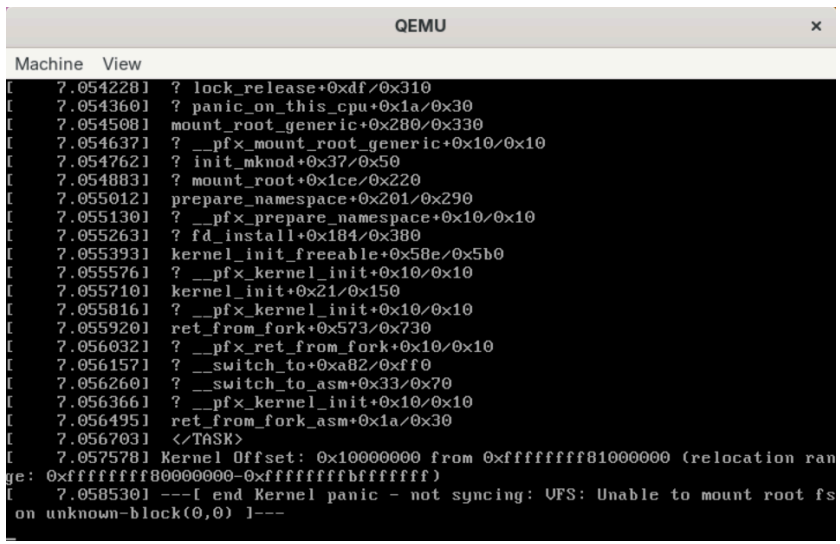
Virtualized kernel

- Let's assume we want to work on an x86_64 kernel:

```
$ qemu-system-x86 -m 4G -kernel <your_bzImage>
```

- ~~and voila, you have a running kernel ready for debugging !~~

Ok, not really, we are missing a ton of stuff...





(Better) Virtualized kernel

```
$ qemu-system-x86 \
[...]
-cpu host \
-enable-kvm \
-nodefaults -display none -serial mon:stdio \
-drive file=<your_rootfs.ext4>,if=virtio \
-virtfs local,path=<host_path>,mount_tag=<a_random_tag>,security_model=mapped \
-device driver=virtio-net,netdev=net0 \
-netdev tap,id=net0,ifname=tap0,script=no,downscript=no \
-append "console=ttyS0 root=/dev/vda ip=10.0.0.2:::255.0.0.0::eth0" \
-gdb tcp:localhost:1234 \
[...]
```

Death by a Thousand Parameters



(Transparent and Easier) Virtualized kernel

- virtme-ng from Andrea Righi can do all the heavy lifting:
  - wraps QEMU and offers plenty of cool options

```
$ cd linux
# Tune your kernel configuration
$ vng -b
$ vng -r <your_kernel_img> --root=<your_rootfs_dir> --debug
```

- You can see the actual QEMU command line with `vng --dry-run [...]`



(Transparent and Easier) Virtualized kernel

- virtme-ng from Andrea Righi can do all the heavy lifting:
  - wraps QEMU and offers plenty of cool options

```
$ cd linux
# Tune your kernel configuration
$ vng -b
$ vng -r <your_kernel_img> --root=<your_rootfs_dir> --debug
```

- You can see the actual QEMU command line with `vng --dry-run [...]`
- It works for cross-compilation too ! (eg: running an *ARM64* kernel on *x86_64*)



Debugging our virtualized kernel

- When started with `--debug`, our emulator runs our development kernel and acts as a debug server:

```
$ gdb vmlinux
(gdb) target remote localhost:1234
(gdb) break do_execveat_common
(gdb) continue
```

- And that's it, we can start debugging our kernel: set breakpoints, control execution, print memory...



Debugging use cases





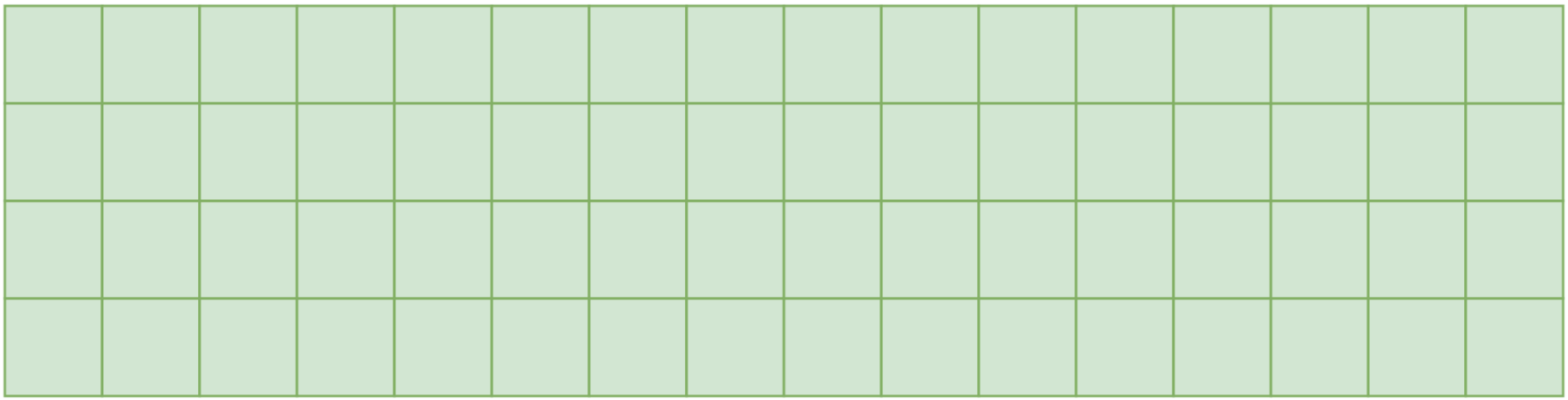
Next challenge: KASAN shadow memory

- KASAN is a kernel development/debug feature allowing to track memory programming mistakes (out-of-bounds and use-after-free)
- It is based on a **shadow memory** area that tracks memory validity
- The shadow memory is tested for each **access** (load or store) to the memory

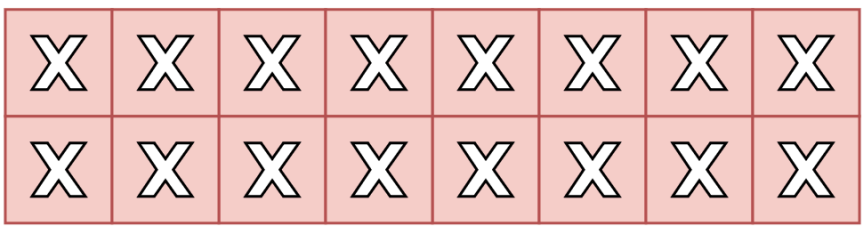


KASAN shadow memory at init/alloc/dealloc

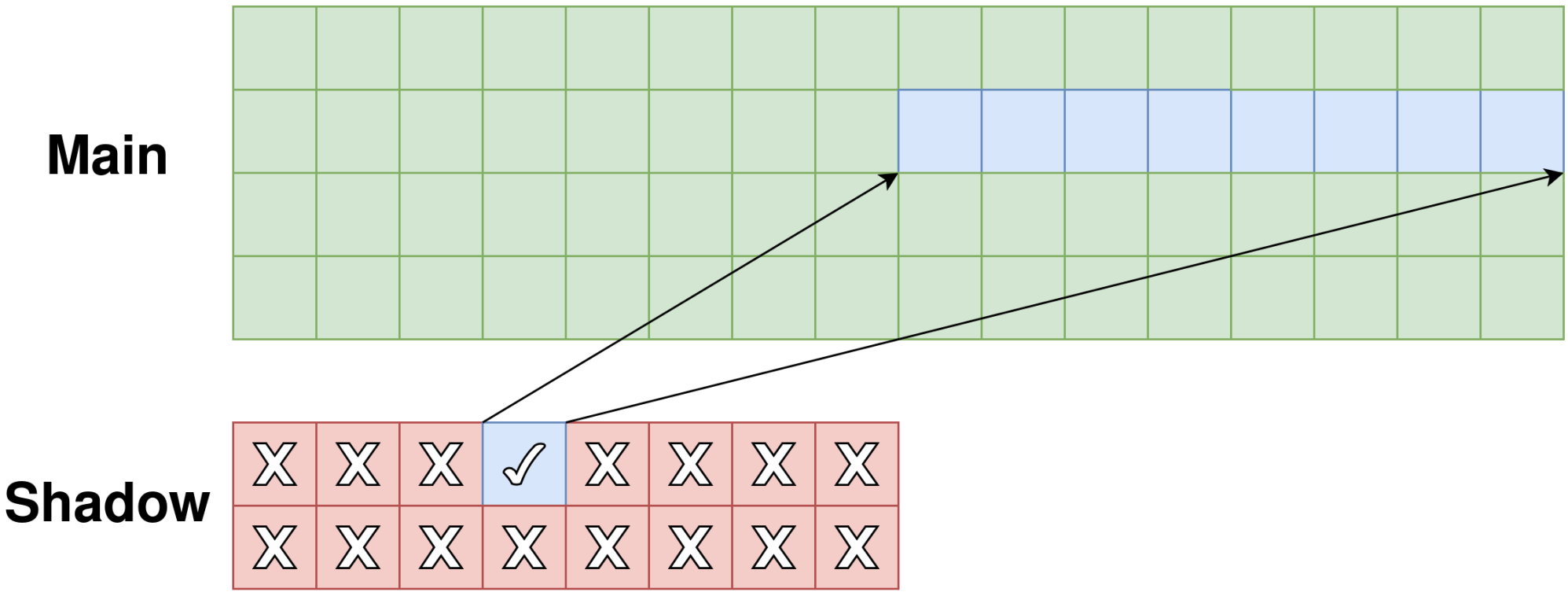
Main



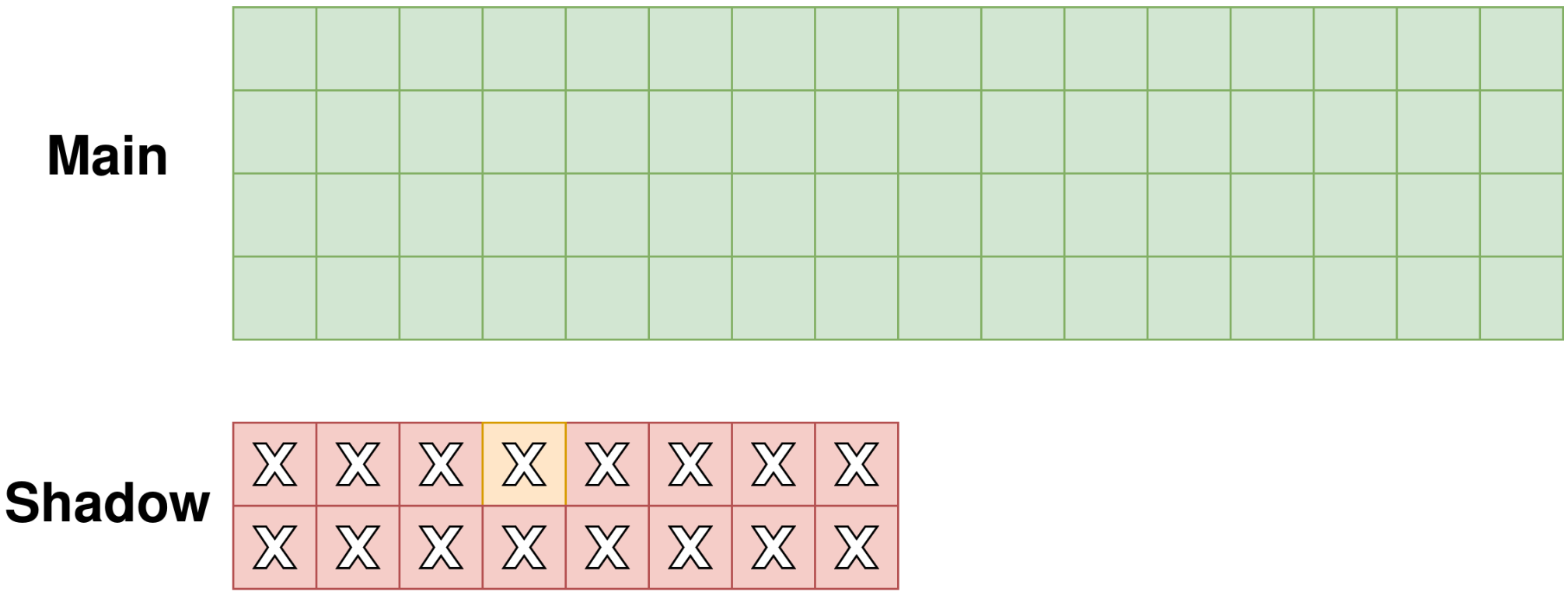
Shadow



KASAN shadow memory at init/alloc/dealloc



KASAN shadow memory at init/alloc/dealloc





KASAN shadow mapping

- There is a 1-to-1 mapping between a **memory granule** (generally, 8 bytes) and its corresponding shadow memory:

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

- When a memory granule can *not* be accessed, the corresponding shadow bytes encode the reason:
  - slab freed (`0xFB`)
  - slab red zone (`0xFC`)
  - invalid vmalloc area (`0xF8`)
  - etc
- How to monitor/debug those memory updates ?



GDB custom commands

- Let's define a custom command to dump some shadow memory !
  - takes an address and size
  - prints the corresponding shadow memory + a few bytes before + a few bytes after



GDB custom commands

- Let's define a custom command to dump some shadow memory !
  - takes an address and size
  - prints the corresponding shadow memory + a few bytes before + a few bytes after
- We just need a few standard GDB commands: `define`, `x`, `set`, `printf`, `eval`
- We will save it in a GDB script file to ease reuse
  - can be loaded manually in gdb, or even automatically if stored correctly (`.gdbinit`)



GDB custom commands

```
define dump_kasan_shadow
  if $argc != 2
    printf "Usage: dump_kasan_shadow <address> <len>\n"
  else
    set $_mem_addr = (unsigned long)($arg0) - (unsigned long)($arg0)%8
    set $_mem_len = (unsigned long)($arg1)
    set $_shadow_addr = ($_mem_addr >> 3) + 0xdffffc0000000000
    set $_shadow_len = $_mem_len >> 3
    if $_shadow_len % 8 != 0
        set $_extra_shadow = 8 -  $_shadow_len % 8
    else
        set $_extra_shadow = 8
    end
    printf "KASAN shadow for %d(+%d)B@%lx:\n", $_mem_len, $_extra_shadow*8, $_mem_addr
    x/8bx $_shadow_addr - 8
    printf "-\t-\t-\t-\t-\t-\t-\t-\t-\t-\t\n"
    eval "x/%dbx $_shadow_addr", $_shadow_len + $_extra_shadow
  end
end
```



GDB custom commands

```
(gdb) dump_kasan_shadow name 11
KASAN shadow for 11(+56)B@ffff888008a4ebc0
0xffffed1001149d70:     0xfc    0xfc    0xfc    0xfc    0xfc    0xfc    0xfc    0xfc
-       -       -       -       -       -       -       -       -       -
0xffffed1001149d78:     0x00    0x03    0xfc    0xfc    0x00    0x00    0xfc    0xfc
```



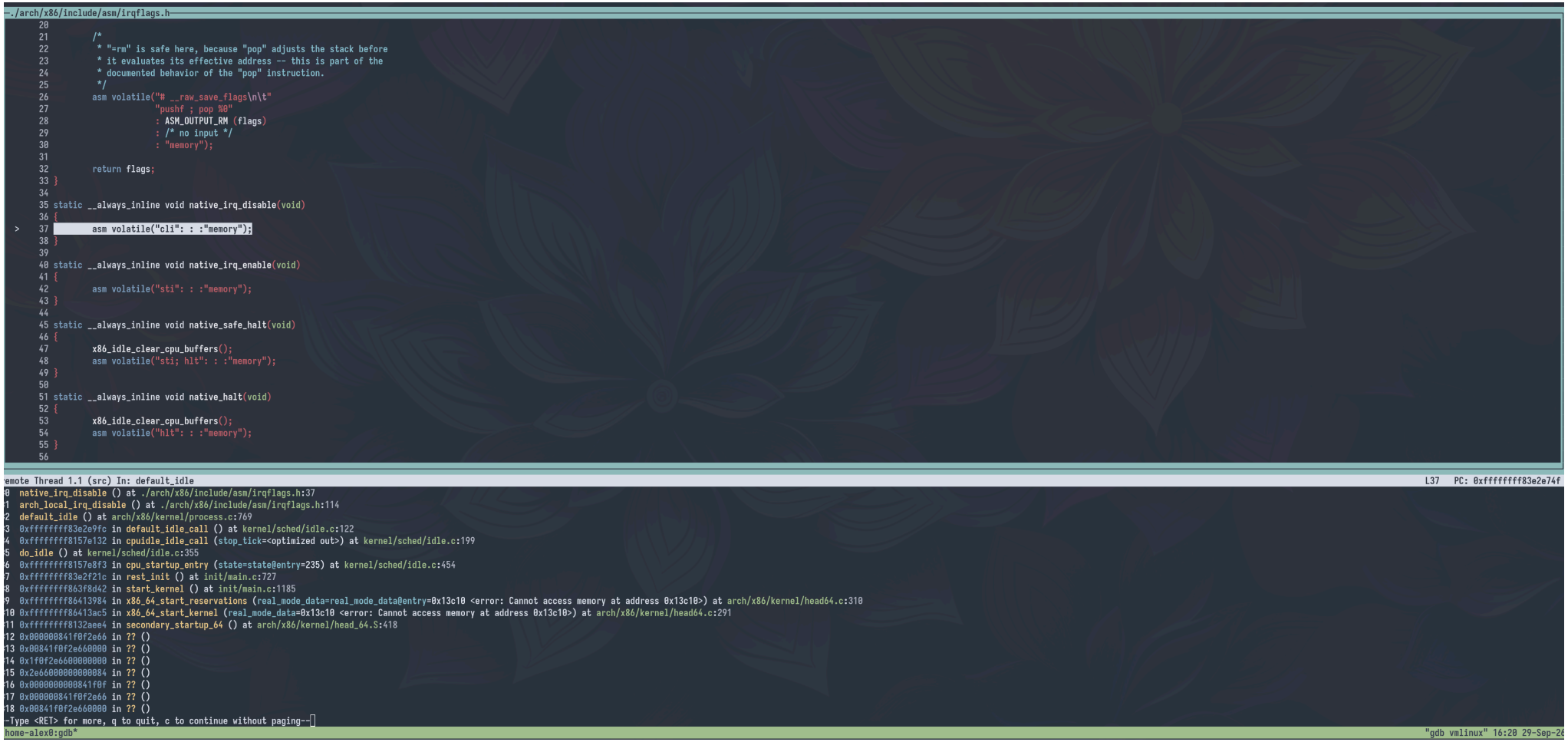
Improving the workflow: TUI

- `gdb` is a pure command-line tool, not very convenient when needing to perform advanced manipulations
- we could use some GDB frontends, but they can be a hassle to configure for our kernel-specific, remote setup
- in-between, we have the TUI !

```
(gdb) tui enable
```

(or C-x C-a to toggle on/off the TUI)

The TUI


-./arch/x86/include/asm/irqflags.h
20
21 /*
22 * "=rm" is safe here, because "pop" adjusts the stack before
23 * it evaluates its effective address -- this is part of the
24 * documented behavior of the "pop" instruction.
25 */
26 asm volatile("# __raw_save_flags\n\t"
27 "pushf ; pop %0"
28 : ASM_OUTPUT_RM (flags)
29 : /* no input */
30 : "memory");
31
32 return flags;
33 }
34
35 static __always_inline void native_irq_disable(void)
36 {
> 37 asm volatile("cli": : :"memory");
38 }
39
40 static __always_inline void native_irq_enable(void)
41 {
42 asm volatile("sti": : :"memory");
43 }
44
45 static __always_inline void native_safe_halt(void)
46 {
47 x86_idle_clear_cpu_buffers();
48 asm volatile("sti; hlt": : :"memory");
49 }
50
51 static __always_inline void native_halt(void)
52 {
53 x86_idle_clear_cpu_buffers();
54 asm volatile("hlt": : :"memory");
55 }
56
emote Thread 1.1 (src) In: default_idle L37 PC: 0xffffffff83e2e74f
0 native_irq_disable () at ./arch/x86/include/asm/irqflags.h:37
1 arch_local_irq_disable () at ./arch/x86/include/asm/irqflags.h:114
2 default_idle () at arch/x86/kernel/process.c:769
3 0xffffffff83e2e9fc in default_idle_call () at kernel/sched/idle.c:122
4 0xffffffff8157e132 in cpuidle_idle_call (stop_tick=<optimized out>) at kernel/sched/idle.c:199
5 do_idle () at kernel/sched/idle.c:355
6 0xffffffff8157e8f3 in cpu_startup_entry (state=state@entry=235) at kernel/sched/idle.c:454
7 0xffffffff83e2f21c in rest_init () at init/main.c:727
8 0xffffffff863f8d42 in start_kernel () at init/main.c:1185
9 0xffffffff86413984 in x86_64_start_reservations (real_mode_data=real_mode_data@entry=0x13c10 <error: Cannot access memory at address 0x13c10>) at arch/x86/kernel/head64.c:310
10 0xffffffff86413ac5 in x86_64_start_kernel (real_mode_data=0x13c10 <error: Cannot access memory at address 0x13c10>) at arch/x86/kernel/head64.c:291
11 0xffffffff8132aee4 in secondary_startup_64 () at arch/x86/kernel/head_64.S:418
12 0x000008841f0f2e66 in ?? ()
13 0x00841f0f2e660000 in ?? ()
14 0x1f0f2e6600000000 in ?? ()
15 0x2e6600000000084 in ?? ()
16 0x0000000000841f0f in ?? ()
17 0x000008841f0f2e66 in ?? ()
18 0x00841f0f2e660000 in ?? ()
-Type <RET> for more, q to quit, c to continue without paging--
home-alex0:gdb* "gdb vmlinux" 16:20 29-Sep-2



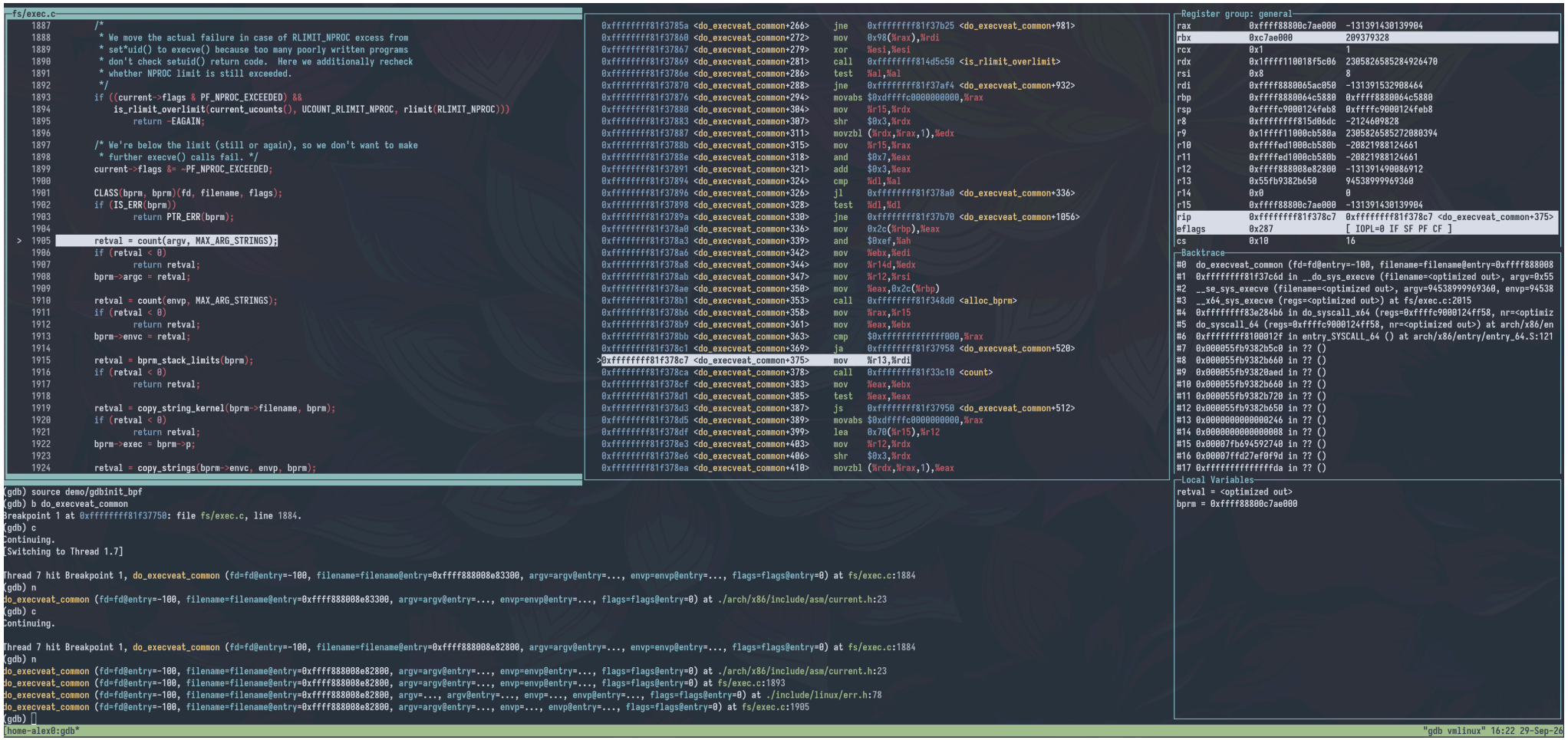
Improving the workflow: custom TUI

- the TUI can display different views: `command`, `source`, `assembly` and `registers`
- you can create your own layout with those views

```
(gdb) tui new-layout my_layout {-horizontal {{-horizontal src 1 asm 1} 2 cmd 1} 3 regs 1} 1
(gdb) layout my_layout
```

- you can even implement your own TUI panes with the Python API

Improving the workflow: custom TUI





TUI: Single-Key mode

- typing full commands is tedious and slows you down
- the TUI comes with a **Single-Key Mode**, can be entered with `C-x s`
- you now get an experience similar to an integrated IDE
  - `c`: continue
  - `n`: next
  - `i`: step into
  - `s`: step
  - `v`: info locals
  - ...
- If the keystroke does not match a shortcut, you get a GDB command prompt



Next challenge: runtime-generated instructions

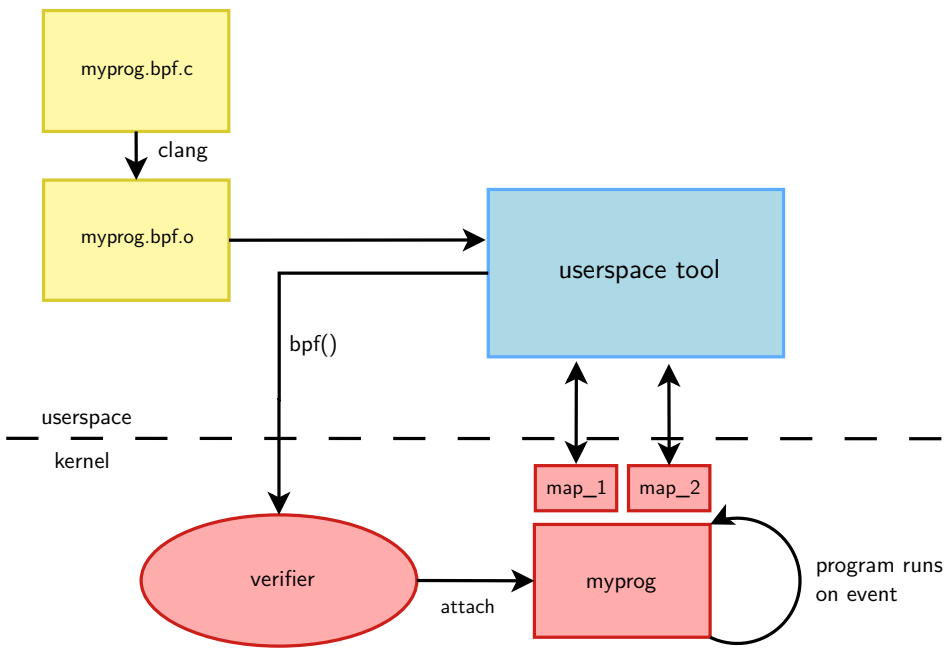
- our current setup works pretty well with kernel **static** code
- but the kernel can generate machine code at runtime !
  - ftrace trampolines
  - **BPF programs**
  - kprobes
  - ...
- gdb can't know in advance about the location of the code we want to debug !

Let's explore this issue with the eBPF example



An eBPF primer

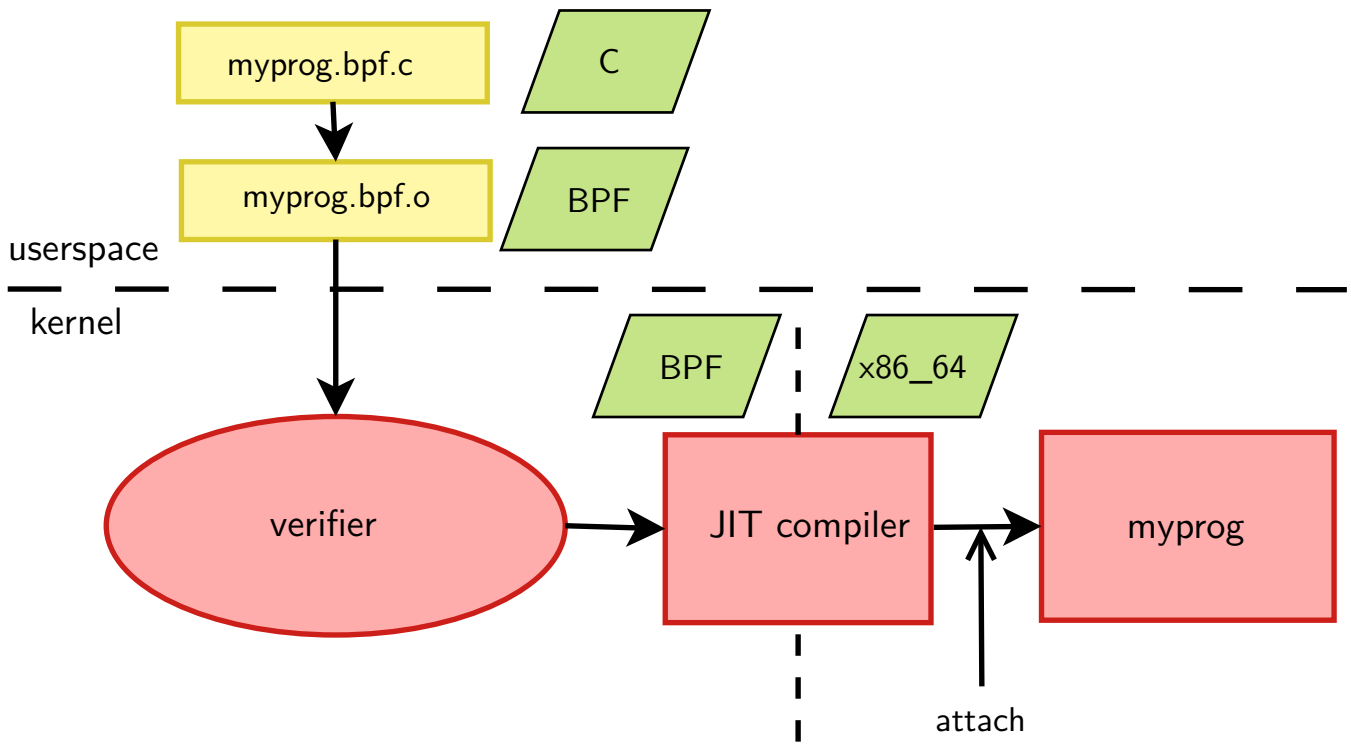
- A powerful technology allowing to inject user programs in the kernel at runtime
- Used for networking, security, monitoring, debugging...
- Allows to virtually hook anywhere in the kernel
- Event-driven



https://bootlin.com/doc/training/debugging/debugging-slides.pdf



Our problem


myprog.bpf.c
C
myprog.bpf.o
BPF
userspace
kernel
BPF
x86_64
verifier
JIT compiler
myprog
attach



BPF/x86 machine code

```c
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

SEC("tcx/ingress")
int sample(struct __sk_buff *skb)
{
  char msg[] = "Hello!";

  bpf_trace_printk(msg,
  sizeof(msg));

  return 0;
}
char LICENSE[] SEC("license") = "GPL";
```

C code

```
0: (b4) w1 = 33
1: (6b) *(u16 *)(r10 -4) = r1
2: (b4) w1 = 543452274
3: (63) *(u32 *)(r10 -8) = r1
4: (18) r1 = 0x6f57206f6c6c6548
6: (7b) *(u64 *)(r10 -16) = r1
7: (bf) r1 = r10
8: (07) r1 += -16
9: (b4) w2 = 14
10: (85) call
bpf_trace_printk#-355520
11: (b4) w0 = 0
12: (95) exit
```

BPF code

```
0:   endbr64
4:   nopl    (%rax,%rax)
9:   nopl    (%rax)
c:   pushq   %rbp
d:   movq    %rsp, %rbp
10:  endbr64
14:  subq    $16, %rsp
1b:  movl    $33, %edi
20:  movw    %di, -4(%rbp)
24:  movl    $543452274, %edi
29:  movl    %edi, -8(%rbp)
2c:  movabsq $8022916924116329800, %rdi
36:  movq    %rdi, -16(%rbp)
3a:  movq    %rbp, %rdi
3d:  addq    $-16, %rdi
41:  movl    $14, %esi
46:  sarq    $5, %gs:-962900675(%rip)
4f:  callq   0xffffffff818e48d0
54:  xorl    %eax, %eax
56:  leave
57:  jmp     0xffffffff83e20f90
```

x86 code



BPF/x86 machine code

```c
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

SEC("tcx/ingress")
int sample(struct __sk_buff *skb)
{
  char msg[] = "Hello!";

  bpf_trace_printk(msg,
  sizeof(msg));

  return 0;
}
char LICENSE[] SEC("license") = "GPL";
```

C code

```
0: (b4) w1 = 33
1: (6b) *(u16 *)(r10 -4) = r1
2: (b4) w1 = 543452274
3: (63) *(u32 *)(r10 -8) = r1
4: (18) r1 = 0x6f57206f6c6c6548
6: (7b) *(u64 *)(r10 -16) = r1
7: (bf) r1 = r10
8: (07) r1 += -16
9: (b4) w2 = 14
10: (85) call
bpf_trace_printk#-355520
11: (b4) w0 = 0
12: (95) exit
```

BPF code

```
0:   endbr64
4:   nopl    (%rax,%rax)
9:   nopl    (%rax)
c:   pushq   %rbp
d:   movq    %rsp, %rbp
10:  endbr64
14:  subq    $16, %rsp
1b:  movl    $33, %edi
20:  movw    %di, -4(%rbp)
24:  movl    $543452274, %edi
29:  movl    %edi, -8(%rbp)
2c:  movabsq $8022916924116329800, %rdi
36:  movq    %rdi, -16(%rbp)
3a:  movq    %rbp, %rdi
3d:  addq    $-16, %rdi
41:  movl    $14, %esi
46:  sarq    $5, %gs:-962900675(%rip)
4f:  callq   0xffffffff818e48d0
54:  xorl    %eax, %eax
56:  leave
57:  jmp     0xffffffff83e20f90
```

x86 code

How to *easily* debug the running program instructions ?



General algorithm

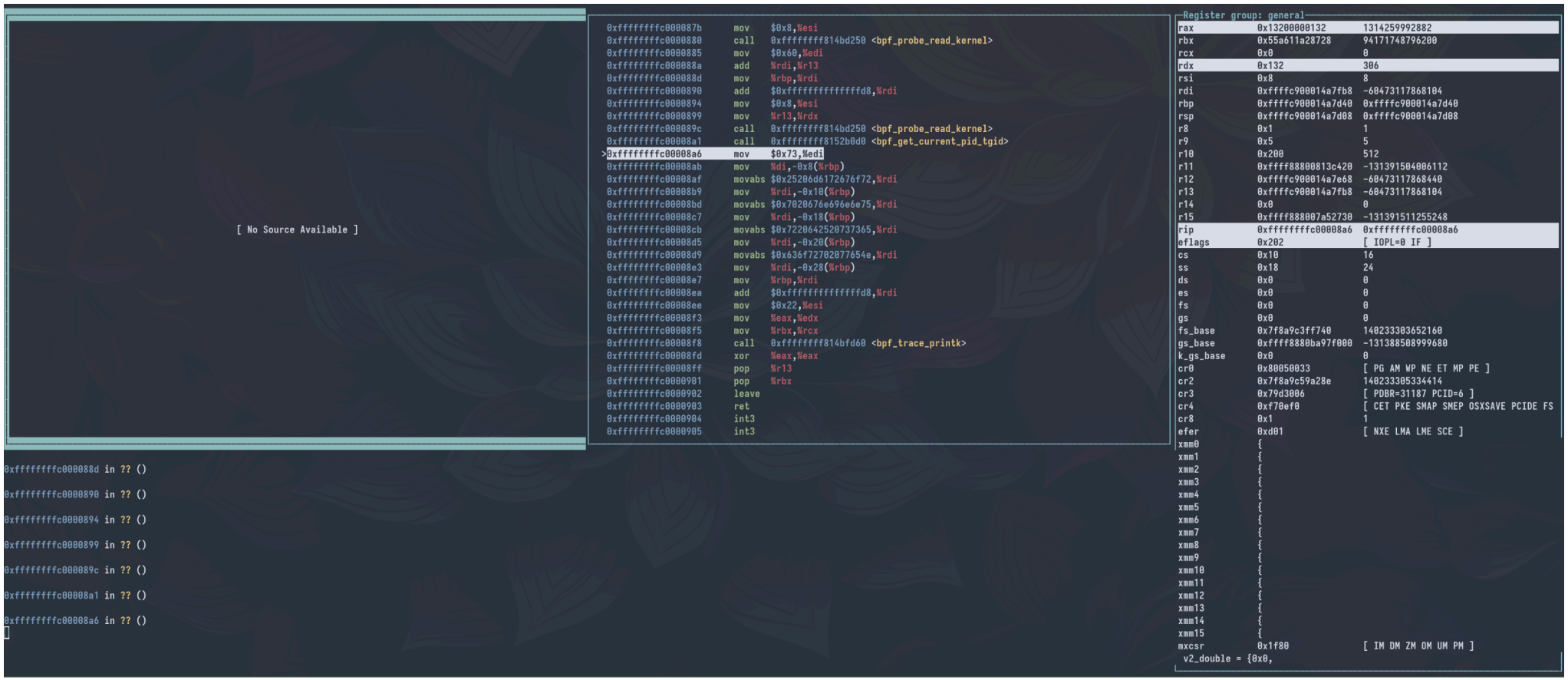
- identify a specific program to debug by its name
- at load time, break at a place where both the name and the address are known
  - right after the JIT step: `bpf_prog_kallsyms_add`
  - the (truncated) name is in `prog->aux->name`
  - the JITed program address is in `prog->bpf_func`
- add a breakpoint on `prog->bpf_func` *if* `prog->aux->name` matches the target program name
- resume execution, wait for the breakpoint to trigger when the program is executed



Breaking in JITed BPF program

```
define bpf-break
  if $argc != 1
    printf "Usage: bpf-break <program_name>\n"
  else
    set $bpf_target_name_full = "$arg0"
    eval "set $bpf_target_name = \"%.15s\"", $bpf_target_name_full
    b bpf_prog_kallsyms_add
    commands
      silent
      set $__prog = (struct bpf_prog *)$rdi
      set $__name = (char *)$__prog->aux->name
      if $_streq($__name, $bpf_target_name)
        printf "Caught program %s loading\n", $bpf_target_name_full
        del $_hit_bpnum
        break *$__prog->bpf_func
      end
      c
    end
  end
  c
end
```

Breaking in JITed BPF program

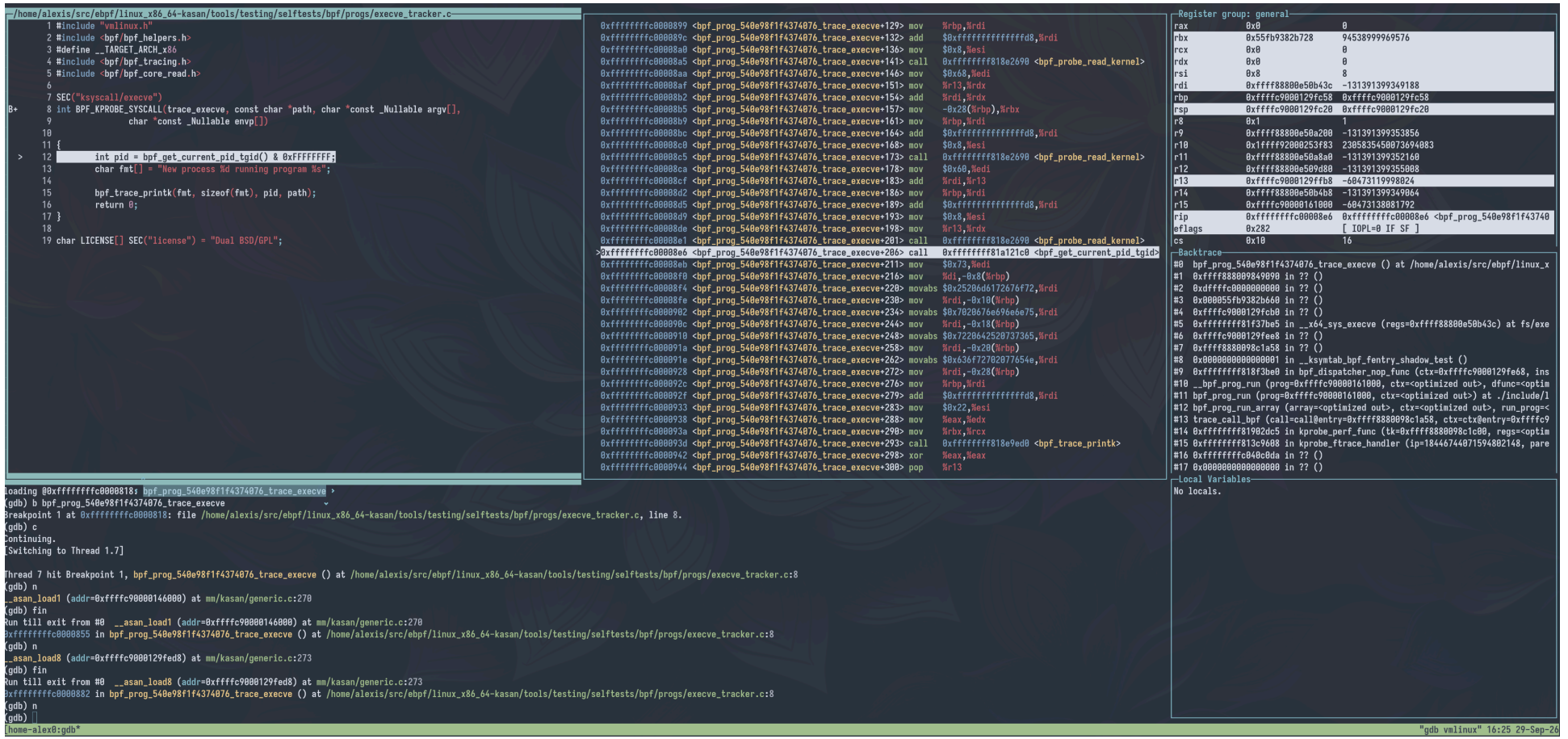




Learning more tips and tricks

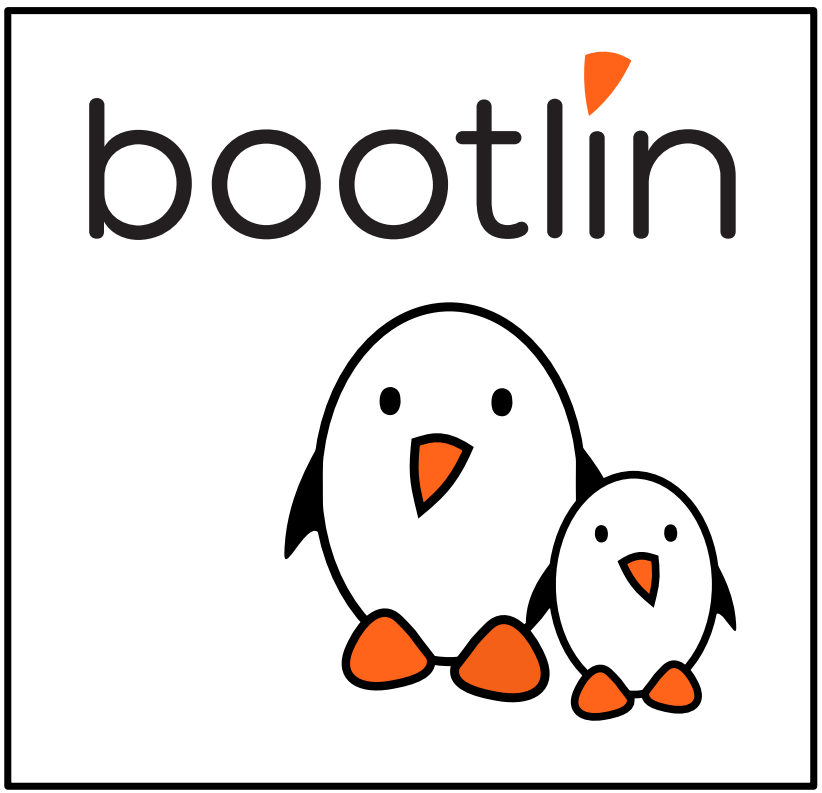
- We have actually started reimplementing an existing kernel tool
- see `scripts/gdb/bpf.py`
  - allows to break into a BPF program based on its *loaded* name (you need to know the program **tag**, eg: `bpf_prog_4e612a6a881a086b_trace_execve`)
  - builds and injects on-the-fly debug info to be able to debug **C code** from the BPF source !

bpf.py



We now see the BPF source code !

Wrapping up





Conclusion

- Knowing how to use GDB in Linux gives debugging superpowers
- Don't be afraid to dive into the documentation
- Learn the shortcuts, increase your workflow speed !
- Automate the tedious debugging tasks

Thank you ! Any questions ?

Slides under CC-BY-SA 3.0

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