Google

KRSI (BPF+LSM) Updates & Progress

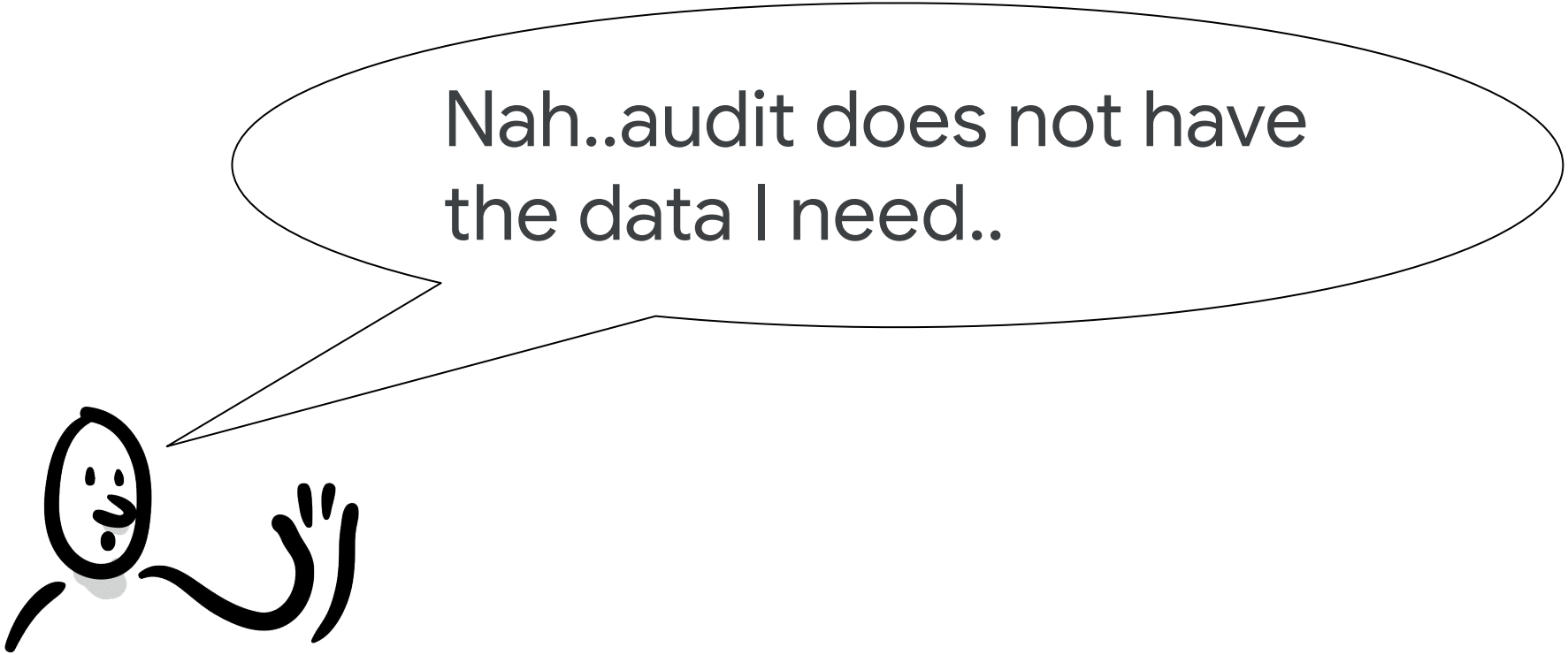
KP Singh

The Story..

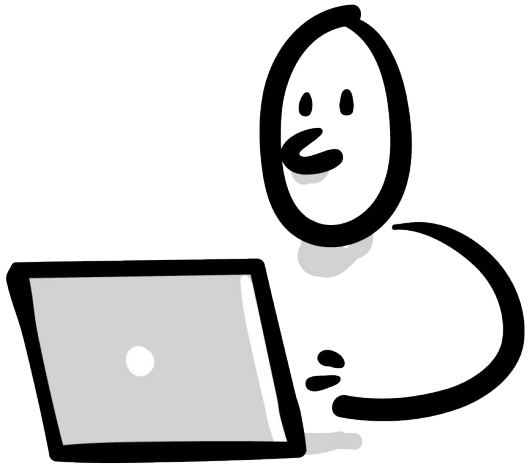
A long long time ago...

Actually, sometime back in 2019...

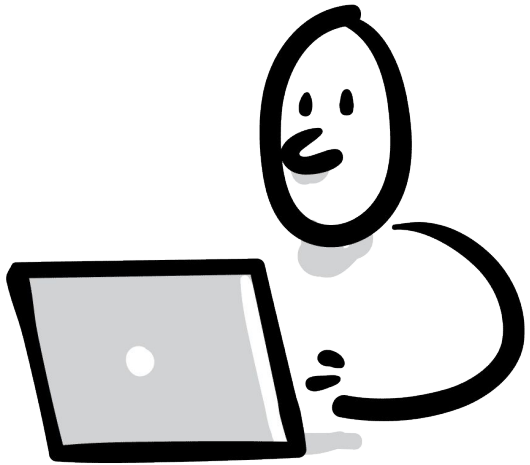

Hey! I need some audit logs..



Patch audit in the kernel..



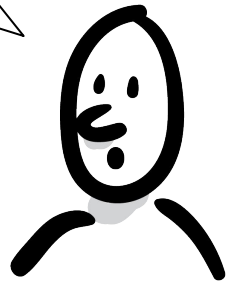
Update auditctl and userspace..

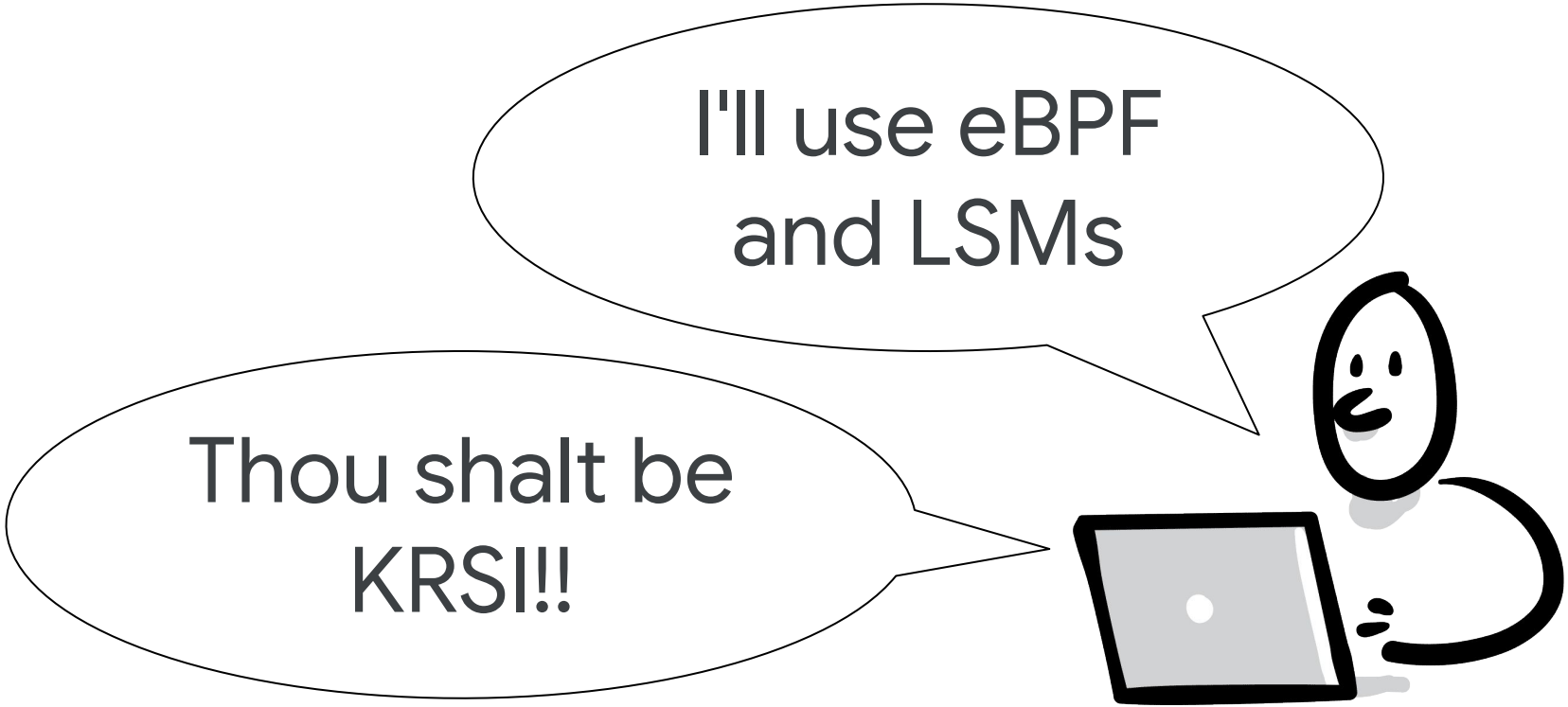


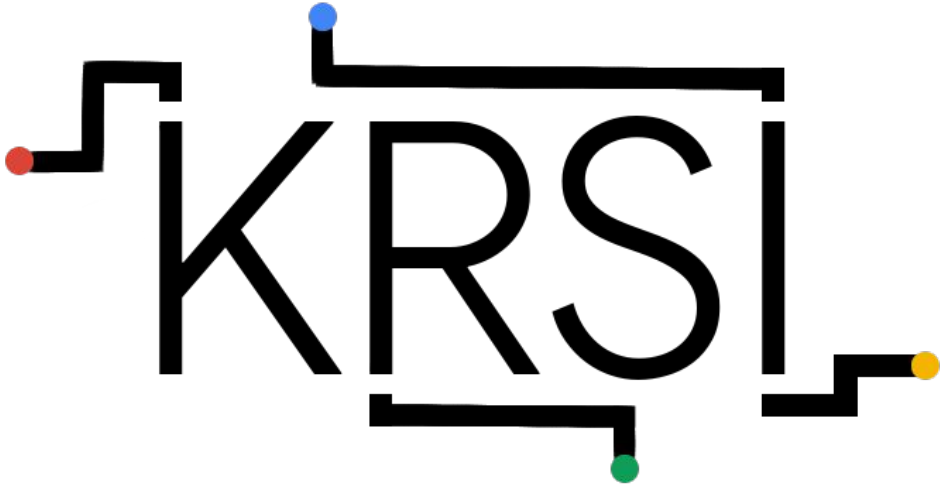

Erm...I want to use this data to prevent something..

Rinse and Repeat for LSMs...

We just need a new way to do Security in Linux..




I'll use eBPF
and LSMs
Thou shalt be
KRSI!!

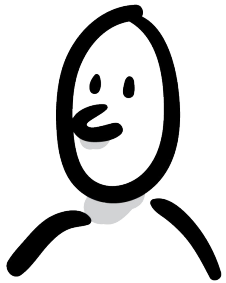


and

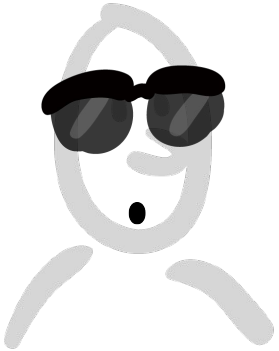
"Kernel Runtime Security Instrumentation"

was born..

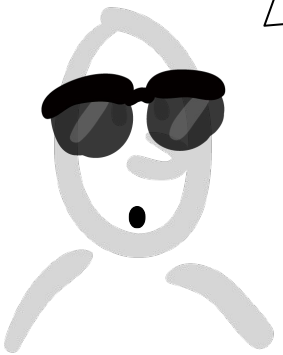
To LPC Portugal!



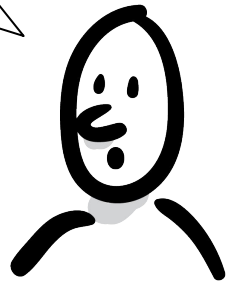
The only way bpf-based LSM can land is both landlock and KRSI developers work together on a design that solves all use cases.



Oh and KRSI is a "great" name!

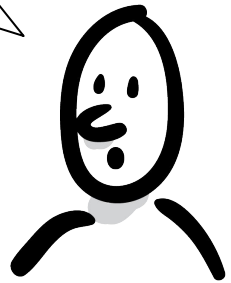


Hey Landlock, you want unprivileged Sandboxing?



Yeahh..

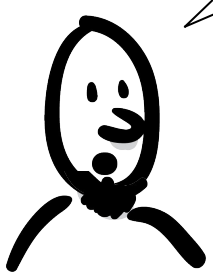
Unprivileged eBPF is still quite some-time away..



Okay, I won't use eBPF :)

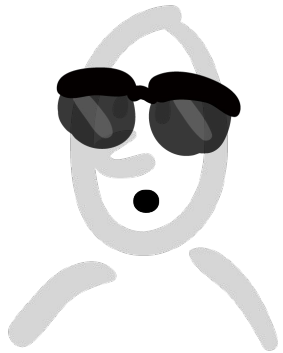
...and now we present security v/s eBPF...

The LSM mechanism
is not zero overhead.
It never has been.



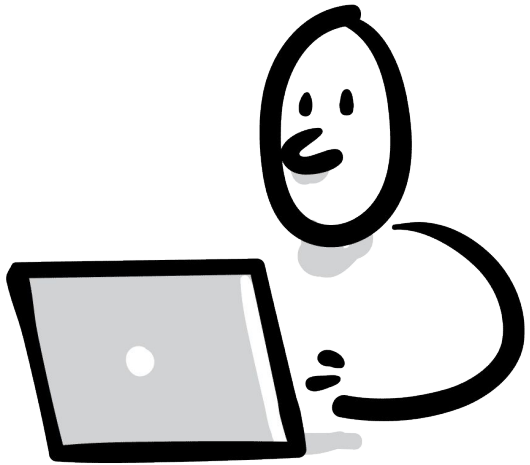
I don't give a flying fig!

I think the key mistake we made is that we classified KRSI as LSM



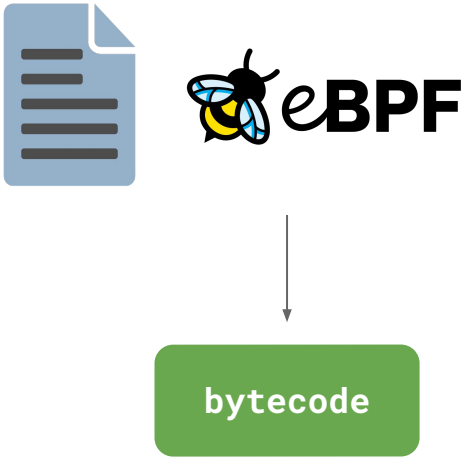
The Treaty of Impedance was signed...

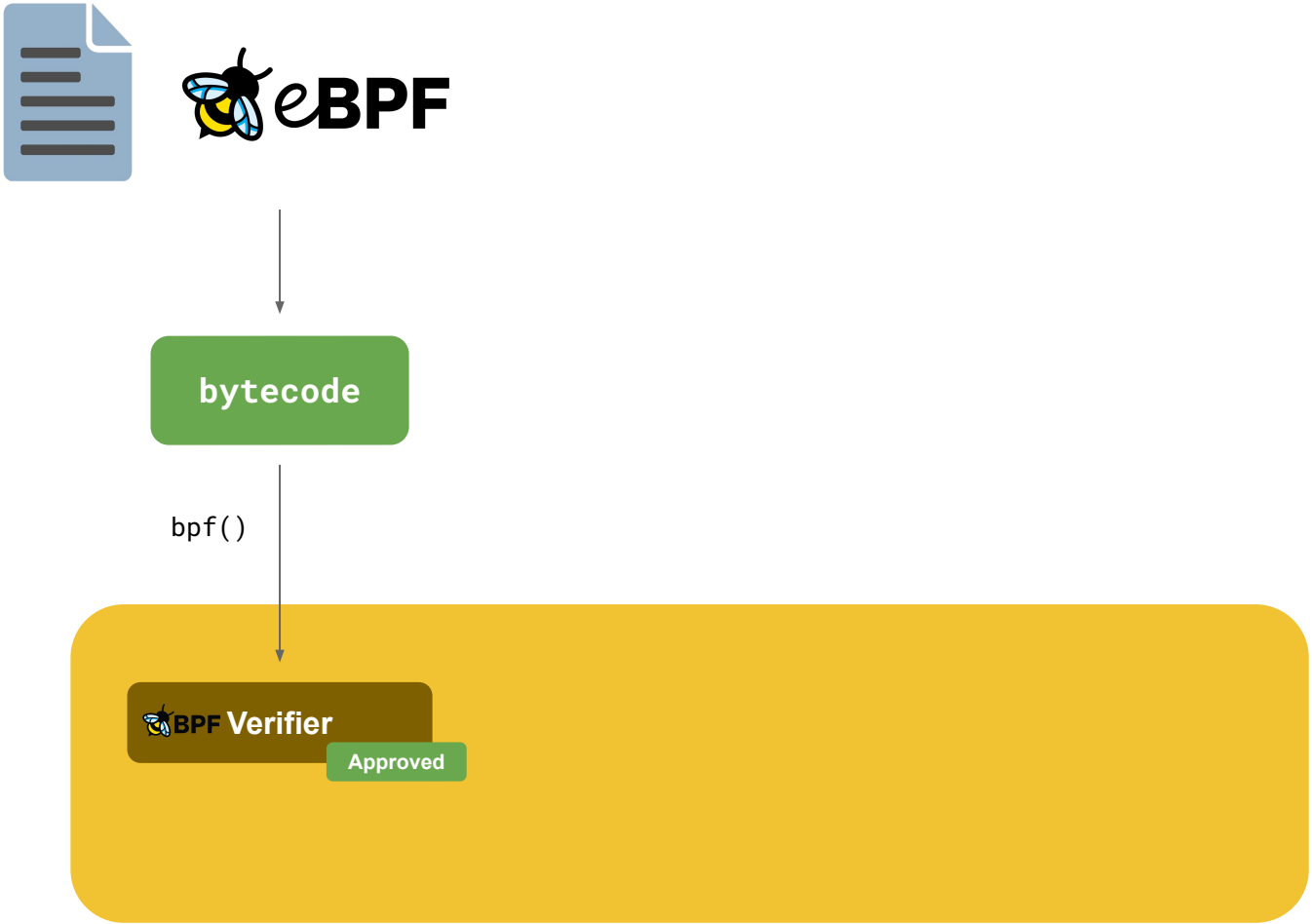
New Kinds of BPF Trampolines

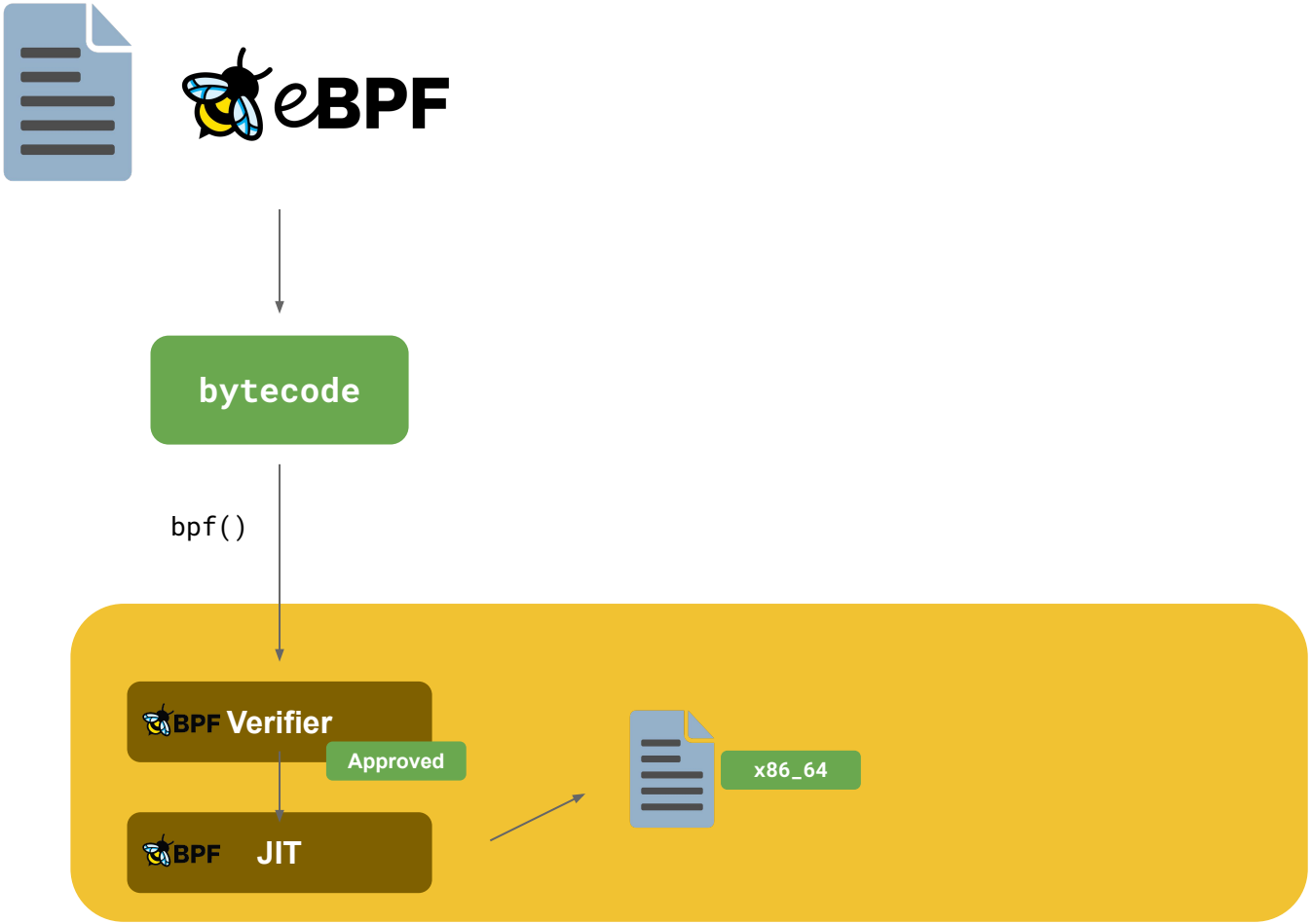


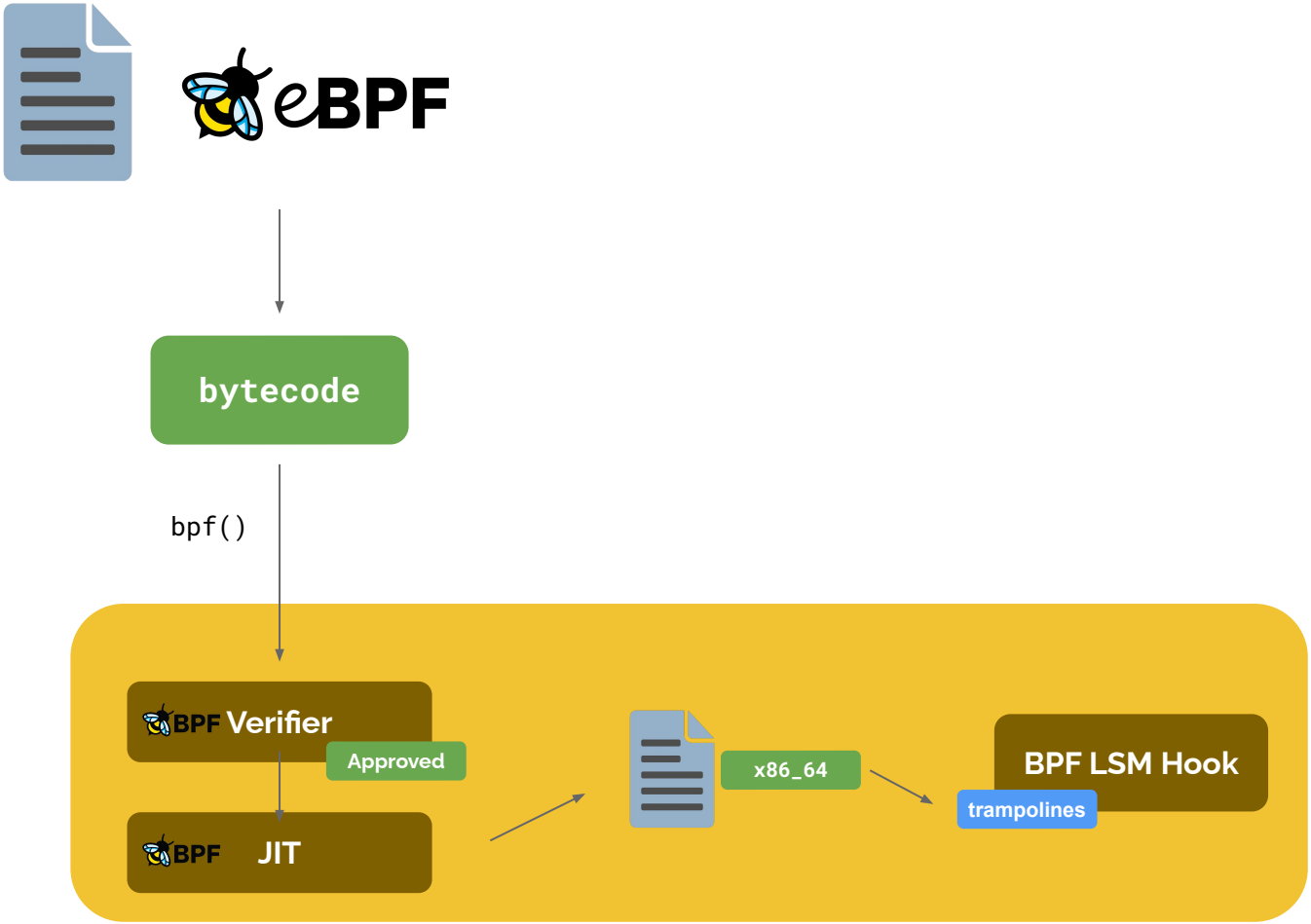
and BPF LSM a.k.a
KRSI was merged!

What?




eBPF
bytecode
bpf()
BPF Verifier
Approved




eBPF
bytecode
bpf()
BPF Verifier
Approved
BPF JIT
x86_64
BPF LSM Hook
trampolines

BPF Trampolines: Types

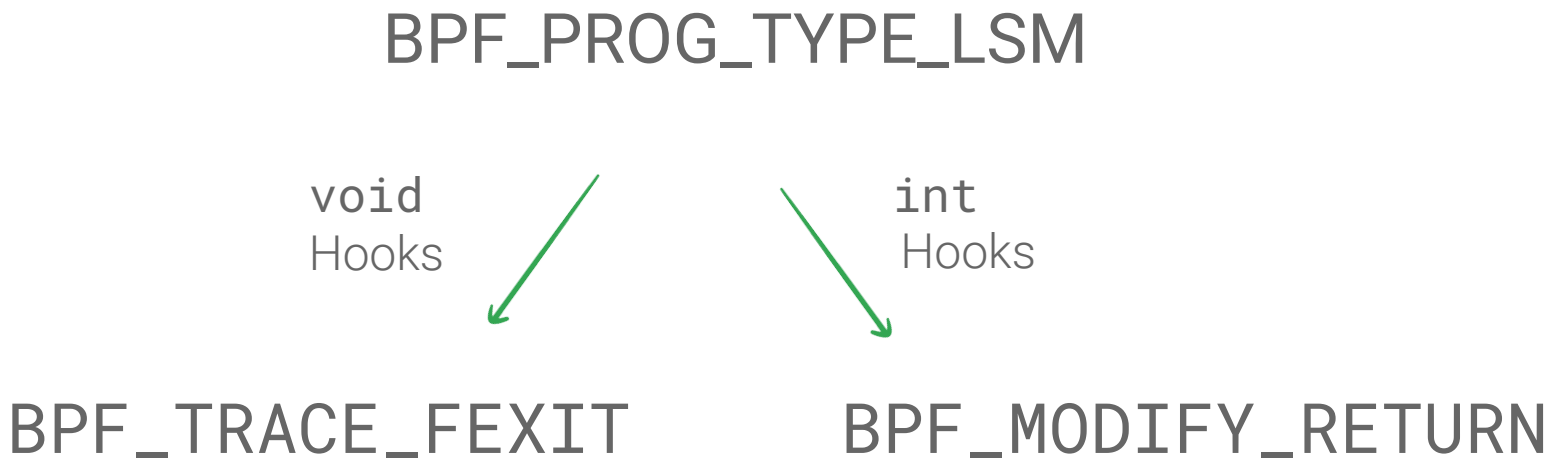
fentry

fmod_ret

fexit

```
int bpf_lsm_bprm_check_security
{
  <update ret by calling fmod_ret progs>

  if (ret != 0)
    goto fexit;

original_function:

  <side_effects_happen_here>

}
```

BPF Trampolines: Types



LSM Hooks

LSM_HOOK Macros (lsm_hook_defs.h)

```
LSM_HOOK(
    int,                                // Return Type
    0,                                  // Default Return Value
    bprm_check_security,                // Name
    struct linux_binprm *bprm           // Parameters
)
```

"Macro magic"

Default Callbacks

```
#define LSM_HOOK(RET, DEFAULT, NAME, ...)                \
                                                         \
  noinline RET bpf_lsm_##NAME(__VA_ARGS__)               \
  {                                                      \
    return DEFAULT;                                      \
  }


#include <linux/lsm_hook_defs.h>


#undef LSM_HOOK
```

Default Callbacks

```
[...]

noinline int
bpf_lsm_bprm_check_security(struct linux_binprm *bprm)
{
  return 0;
}


[...]
```

Initialize LSM Hooks

```
#define LSM_HOOK(RET, DEFAULT, NAME, ...) \
  LSM_HOOK_INIT(NAME, bpf_lsm_##NAME),
  #include linux/lsm_hook_defs.h>

#undef LSM_HOOK
```

Initialize BPF LSM Hooks

```
[...]

LSM_HOOK_INIT(bprm_check_security,
              bpf_lsm_bprm_check_security);

[...]
```

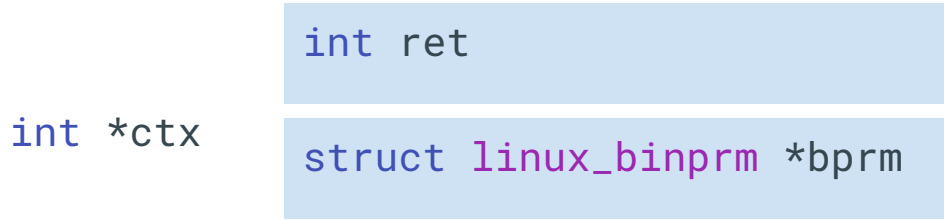
Implementing Hooks

Load a program for `bprm_check_security`

BPF Program

```
SEC("lsm/bprm_check_security")
int BPF_PROG(my_prog, struct linux_binprm *bprm, int ret)
{
    __u32 pid = bpf_get_current_pid_tgid() >> 32;

    if (monitored_pid == pid)
        bprm_count++;

    return 0;
}
```

Context Simplification




```
BPF_PROG(my_prog, struct linux_binprm *bprm, int ret)

my_prog(int *ctx) {
  __my_prog(ctx[0], ctx[1])
}
```

Verification

/sys/kernel/btf/vmlinux

Compact type information (BTF)

~125MB of DWARF

`btf_ctx_access`

Verifier

BPF LSM Hooks: Object File

```
$ objdump -Sr kernel/bpf/bpf_lsm.o

LSM_HOOK(int, 0, bprm_check_security, struct linux_binprm *bprm)

100:   e8 00 00 00 00          callq  105 <bpf_lsm_bprm_check_security+0x5>
101:                           R_X86_64_PLT32 __fentry__-0x4
105:   31 c0                   xor    %eax,%eax>:
107:   c3                      retq
108:   0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
```

BPF LSM Hooks: after `__init`

```
LSM_HOOK(int, 0, bprm_check_security, struct linux_binprm *bprm)

100:   0f 1f 44 00 00          nopl 0x00(%eax,%eax,1)

105:   31 c0                   xor    %eax,%eax>:
107:   c3                      retq
108:   0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
```

ftrace_nop_initialize

BPF Trampoline Update

```
LSM_HOOK(int, 0, bprm_check_security, struct linux_binprm *bprm)

100:   e8 00 00 00 00 64        callq  <trampoline_image>

105:   31 c0                    xor    %eax,%eax>:
107:   c3                       retq
108:   0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)

200: <trampoline_image>
```

LSM Trampolines: Creation

```
push   %rbp
mov    %rsp,%rbp
sub    $0x10,%rsp
push   %rbx
```

Create a frame for a stack size of 16 (`0x10`) bytes:

- 8 bytes for `struct linux_binprm *bprm`
- 8 bytes to save the return value

LSM Trampolines: Invocation

```
mov     %rdi,-0x10(%rbp)

xor     %eax,%eax
mov     %rax,-0x8(%rbp)

callq   __bpf_prog_enter
mov     %rax,%rbx

lea     -0x10(%rbp),%rdi

callq   addr_of_jited_lsm_prog
mov     %rax,-0x8(%rbp)

movabs $addr_struct_bpf_prog,%rdi
mov     %rbx,%rsi
callq   __bpf_prog_exit
```

Save the first argument on the stack

Clear out the return value passed to first LSM program.

ctx (`int *` array) for the BPF program

Call the JITed program
Save the return value on the stack

LSM Trampolines: BPF_MODIFY_RETURN

```
  cmpq    $0x0,-0x8(%rbp)
  jne     <do_exit>

 [...]    // Repeat for more progs

  mov     -0x10(%rbp),%rdi
  callq   <bpf_lsm_bprm_check_security+0x5>
  mov     %rax,-0x8(%rbp)

  nopl    0x0(%rax,%rax,1)
  nopw    0x0(%rax,%rax,1)

do_exit:
```

Skip calling the original function and the rest of the programs upon a non-zero return value

nops to align jump target

BPF Trampolines - Exit

```
mov    -0x8(%rbp),%rax
```

} Update the return value from the stack

```
pop    %rbx
leaveq
add    $0x8,%rsp
retq
```

Improvements

Indirect Calls

```
hlist_for_each_entry(P, &security_hook_heads.FUNC, list)    \
{                                                           \
    RC = P->hook.FUNC(__VA_ARGS__);                         \
    if (RC != 0)                                            \
        break;                                              \
}
```

Indirect calls worsened by retpolines

and default callbacks are added everywhere!!

So how bad is it?

```c
int main(void) {
    int fd = eventfd(0, 0);
    int c = 10000;

    while (c--)
        eventfd_write(fd, 1);

    return 0;
}
```

```
int main(void) {
    int fd = eventfd(0, 0);
    int c = 10000;

    while (c--)
        eventfd_write(fd, 1);

    return 0;
}
```

```
int main(void) {
    int fd = eventfd(0, 0);
    int c = 10000;

    while (c--)
        eventfd_write(fd, 1);

    return 0;
}
```

+ ~4%

We know the addresses of LSM Hooks at `__init..`

Use DEFINE_STATIC_CALL...

Turn this...

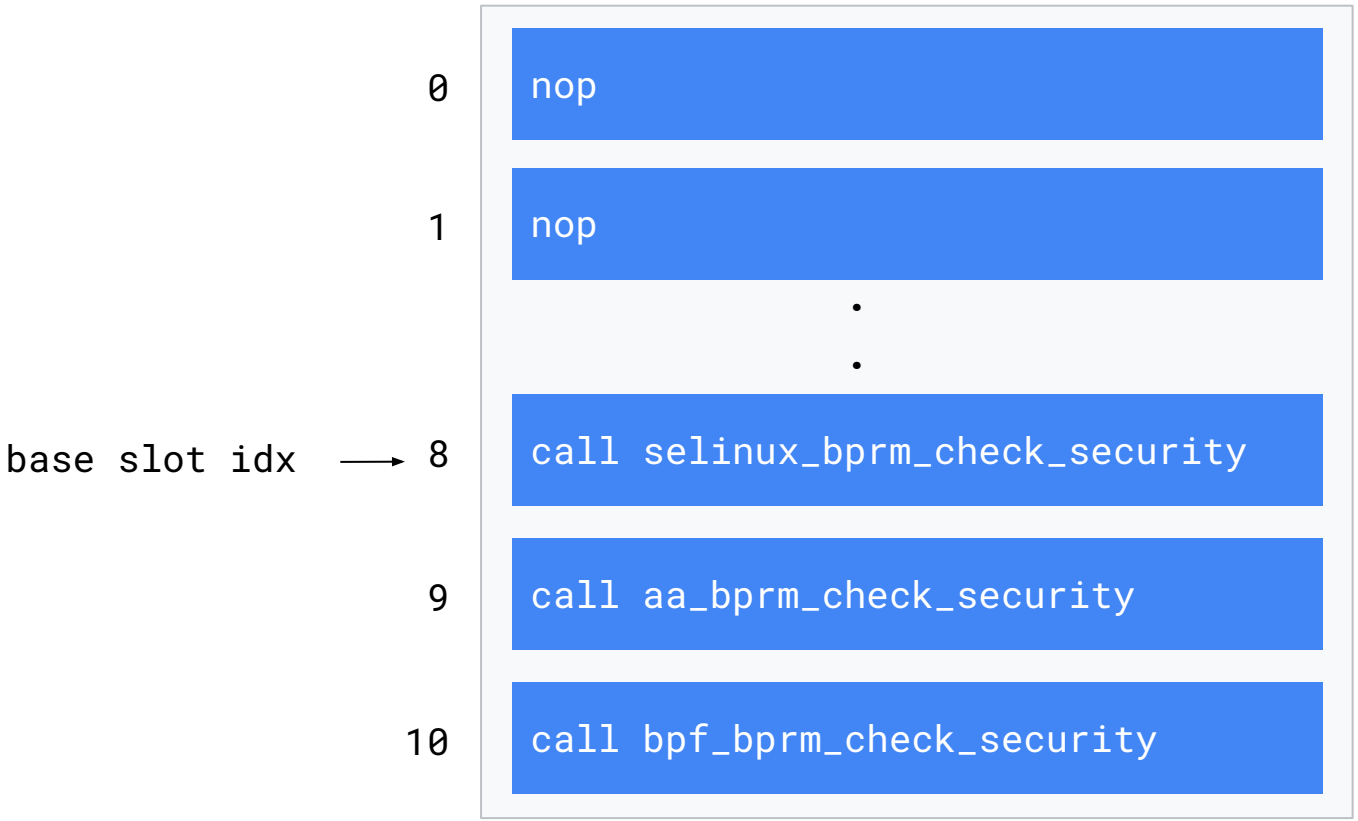
```
hlist_for_each_entry(P, &security_hook_heads.FUNC, list)    \
{                                                           \
    RC = P->hook.FUNC(__VA_ARGS__);                         \
    if (RC != 0)                                            \
        break;                                              \
}
```

Into this!

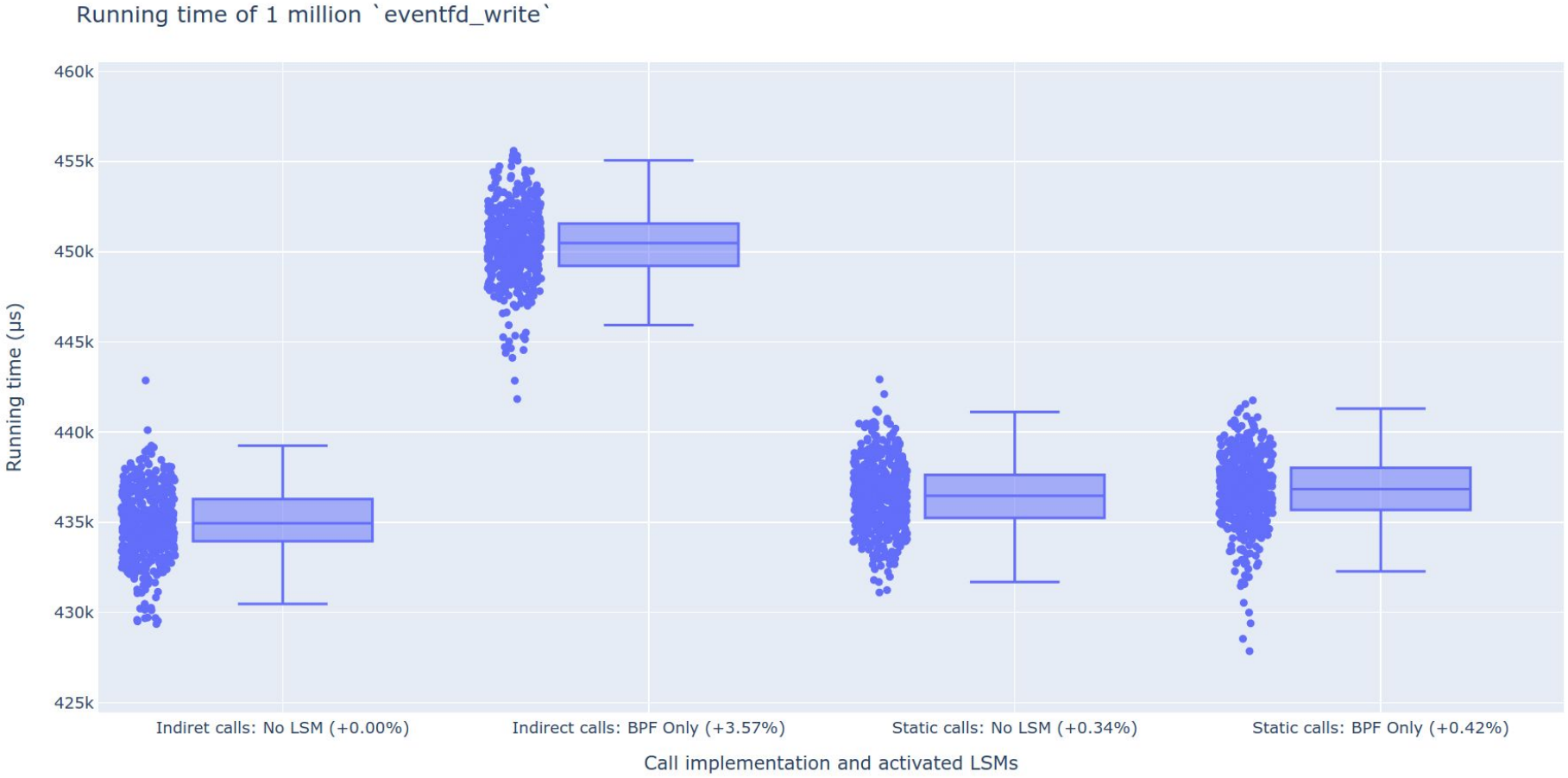
```
do                                                          \
{                                                           \
    RC = callback_A(__VA_ARGS__);                           \
    if (RC != 0)                                            \
        break;                                              \
    RC = callback_B(__VA_ARGS__);                           \
    if (RC != 0)                                            \
        break;                                              \
   ...
} while(0);
```

Slots for call instructions at compile time.

`bprm_check_security` call slots patched at `__init`, starting from the bottom



```
switch (base_slot_idx) {
case 0:
    NOP
    if (RC != 0) break;
case 1:
    NOP
    if (RC != 0) break;
    ...
case 8:
    RC = selinux_bprm_check_security(__VA_ARGS__);
    if (RC != 0) break;
case 9:
    RC = aa_bprm_check_security(__VA_ARGS__);
    if (RC != 0) break;
case 10:
    RC = bpf_bprm_check_security(__VA_ARGS__);
    if (RC != 0) break;
}
```


Running time of 1 million `eventfd_write`
460k
455k
450k
445k
440k
435k
430k
425k
Running time (µs)
Indiret calls: No LSM (+0.00%)
Indirect calls: BPF Only (+3.57%)
Static calls: No LSM (+0.34%)
Static calls: BPF Only (+0.42%)
Call implementation and activated LSMs

Progress on `DEFINE_STATIC_CALL` is slow...

Upcoming..

BPF Ring Buffer

Merged!

`bpf_d_path` helper

Merged!

Storage blobs a.k.a `bpf_local_storage`

almost there..

Sleepable BPF

getting close...

Advanced string helpers
(argv, file paths..)

Not started
(Custom Patches)

Load BPF programs during boot..

Not started...

Thank You!

Google