KernelMemorySanitizer

a look under the hood

Alexander Potapenko <glider@google.com>

Agenda

- Intro
- Undefined behavior and uninitialized values
- MemorySanitizer
- KernelMemorySanitizer
- Automatic memory initialization
- Q&A

Intro

whoami

- doing memory analysis tools at Google for the last 13 years
- homegrown Valgrind distribution
- ASan for Linux and Mac OS userspace
- deploying ASan and TSan in Chrome
- memory initialization, KMSAN and KFENCE for Linux kernel
- @0xGlider

Dynamic Tools team at Google

- Started with custom Valgrind patches, but escalated quickly ;)
- ASan, TSan, MSan for userspace - check 'em out!
- KASAN, KCSAN, KMSAN for the kernel
- Probabilistic tools: GWP-ASan (userspace), KFENCE (kernel)
- Fuzzing: libfuzzer (userspace), syzkaller (kernel)
- CFI (control-flow integrity)
- Hardware-accelerated checks (ARM MTE)
- ... and way more

Most of our work is open source.

Fuzzing?

Generating random or semi-random inputs for your code to increase coverage

- because testing is hard
- and some people are too lazy to write tests

Without good test coverage, bug finding tools are useless - fuzzing can help!

NB: Fuzzing does not replace normal testing.

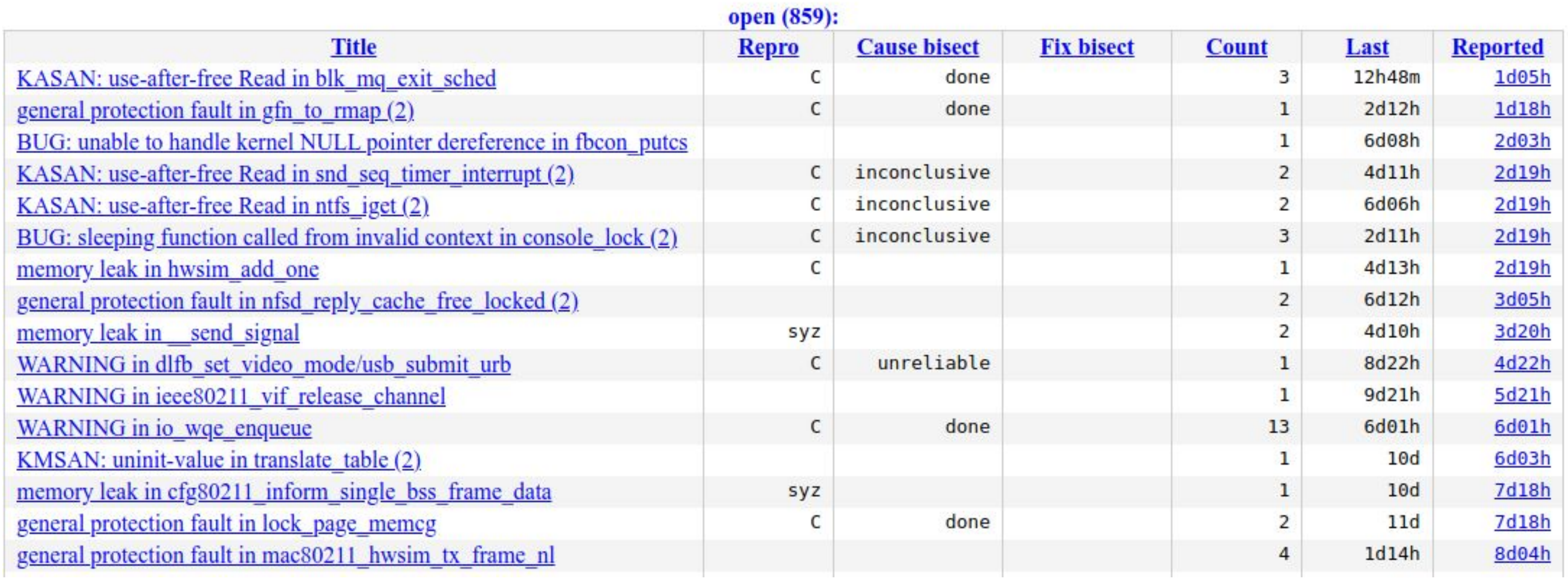
See also: "Fuzz or lose!" by Kostya Serebryany @ CppCon 2017

syzkaller

```
mmap(&(0x7f0000000000), (0x1000), 0x3, 0x32, -1, 0)
r0 = open(&(0x7f0000000000)="./file0", 0x3, 0x9)
read(r0, &(0x7f0000000000), 42)
close(r0)
```

See also: "syzkaller: Adventures in Continuous Coverage-guided Kernel Fuzzing"

syzbot



open (859):

Title	Repro	Cause bisect	Fix bisect	Count	Last	Reported
KASAN: use-after-free Read in blk_mq_exit_sched	C	done		3	12h48m	1d05h
general protection fault in gfn_to_rmap (2)	C	done		1	2d12h	1d18h
BUG: unable to handle kernel NULL pointer dereference in fbcon_putcs				1	6d08h	2d03h
KASAN: use-after-free Read in snd_seq_timer_interrupt (2)	C	inconclusive		2	4d11h	2d19h
KASAN: use-after-free Read in ntfs_iget (2)	C	inconclusive		2	6d06h	2d19h
BUG: sleeping function called from invalid context in console_lock (2)	C	inconclusive		3	2d11h	2d19h
memory leak in hwsim_add_one	C			1	4d13h	2d19h
general protection fault in nfsd_reply_cache_free_locked (2)				2	6d12h	3d05h
memory leak in __send_signal	syz			2	4d10h	3d20h
WARNING in dlfb_set_video_mode/usb_submit_urb	C	unreliable		1	8d22h	4d22h
WARNING in ieee80211_vif_release_channel				1	9d21h	5d21h
WARNING in io_wqe_enqueue	C	done		13	6d01h	6d01h
KMSAN: uninit-value in translate_table (2)				1	10d	6d03h
memory leak in cfg80211_inform_single_bss_frame_data	syz			1	10d	7d18h
general protection fault in lock_page_memcg	C	done		2	11d	7d18h
general protection fault in mac80211_hwsim_tx_frame_nl				4	1d14h	8d04h

Undefined behavior and uninitialized values

Standards everywhere

C and C++ implementations are regulated by standards, which:

- describe the behavior of conforming programs
- list the cases in which the behavior is implementation-defined, unspecified or undefined.

Implementation-defined behavior - compiler-dependent, must be documented.

Unspecified behavior - compiler-dependent, may be undocumented.

Undefined behavior - a certain construct must not occur in well-formed programs

- otherwise the compiler may do whatever it wants ***with the whole program***.

Why have undefined behavior?

Introducing undefined behavior lets the compilers perform many powerful optimizations.

If they don't today, they still may want that tomorrow.

Developers may ignore the standards, but the compilers don't.

See also the "What Every C Programmer Should Know About Undefined Behavior" series.

Uninitialized memory

```
1) int i;
   if (i) { ... }

2) int *p = kmalloc(size, ...);
   copy_to_user(uptr, p, size);

3) kfree(p);
   array[*p] = q;
```

```
4) struct pair {
     char a;
     int b;
   }

   pair c = {1, 2};
   if (((char *)&c)[2]) {
     ...
   }
```

Using uninitialized values is also undefined behavior

All bets are off: the compiler may do whatever it wants with your code.

```
int crypto_random() {
    int r;  // get a random value from the stack!
    return r;
}
```

EXPECTATION

```
crypto_random():
        movl    -4(%rsp), %eax
        ret
```

REALITY 1

```
crypto_random():
        xorl    %eax, %eax
        ret
```

REALITY 2

Information leaks

```c
#include <stdio.h>
#include <string.h>

void foo() {
  char buf[16];
  strcpy(buf, "Hello!");
}

void bar() {
  char buf[16];  // uninitialized
  printf("%s\n", buf);
}

int main() {
  foo();
  bar();
  return 0;
}
```

Security impact

Attackers have a lot of options to control uninitialized memory

- hundreds of articles on how to exploit uninitialized memory.

Possible outcome:

- crashes and denials of service;
- information leaks (from pointers to passwords and other sensitive data);
- privilege escalations and remote code execution.

MemorySanitizer

MemorySanitizer (MSan)

Userspace LLVM tool that detects uses of uninitialized values

- around since 2013, Linux-only
- 1:1 shadow memory to track every bit of app memory (~2x memory overhead)
- compiler instrumentation to update shadow (~3x performance overhead)
- optional origin tracking (extra 1.5x memory, 1.5x performance)

See also: "MemorySanitizer: fast detector of uninitialized memory use in C++"

Shadow memory

A concept borrowed from Valgrind:

- every bit corresponds to a bit of application memory
- 0 means initialized, 1 - uninitialized ("**poisoned**")

MSan maintains a huge memory mapping at `0x200000000000` for shadow

- compile-time constants and globals are initialized
- malloc()ed memory is uninitialized
- local variables are uninitialized

See also: "Using Valgrind to detect undefined value errors with bit-precision".

Bugs and non-bugs

```
int a, b;

b = a;
```

```
int a, b;

if (a) ...
```

Not a bug!

BUG!

Uses of uninitialized values

- Conditions: `if (uninit) ...`
- Pointer dereferencing: `buffer[uninit]` or `uninit[index]`
- library calls: `strlen(uninit)`
- system calls: `write(uninit, ...)`

Shadow propagation: bitwise AND

```
C = A & B

C' = (A' & B') | (B & C') | (B' & C)
```

A	0	?	?	1	1	0	1	0
B	0	0	0	0	0	0	1	1
C	0	0	0	0	0	0	1	0

Values

A'	0	1	1	0	0	0	0	0
B'	0	0	0	0	0	0	0	0
C'	0	0	0	0	0	0	0	0

Shadow

MSan instrumentation

- shadow load/store for each memory load/store
- shadow checks for conditions and pointer dereferences
- shadow propagation for operations
- TLS variables to track function parameters and return values
    - no changes to function prototypes
- instrumentation actually done at LLVM middle-end level
    - a lot of redundant checks are deleted

MSan instrumentation

```
void foo(char *pa) {
  char a = *pa;




  if (a) bar();
}
```

MSan instrumentation

```
void foo(char *pa) {
  char a = *pa;
  char a_shadow = *(pa - 0x400000000000);
  if (a_shadow)
      __msan_warning();
  if (a) bar();
}
```

MSan instrumentation

```
void foo(char *pa) {
  char a = *pa, b = 3;


  char c = a & b;





  if (c) bar();
}
```

MSan instrumentation

```
void foo(char *pa) {
  char a = *pa, b = 3;
  char a_shadow = *(pa - 0x400000000000);
  char b_shadow = 0x0;
  char c = a & b;
  char c_shadow = (a_shadow & b_shadow) |
                  (a & b_shadow) | (b & a_shadow);
  if (c_shadow)
      __msan_warning();
  if (c) bar();
}
```

```
 entry:
    %0 = load i64, i64* getelementptr inbounds ([100 x i64], [100 x i64]* @__msan_param_tls, i32 0, i32 0),
align 8
    %_mscmp = icmp ne i64 %0, 0
    br i1 %_mscmp, label %1, label %2, !prof !2

  1:                                                  ; preds = %entry
    call void @__msan_warning_with_origin(i32 0) #2
    br label %2

  2:                                                  ; preds = %entry, %1
    %3 = load i8, i8* %pa, align 1, !tbaa !3
    %4 = ptrtoint i8* %pa to i64
    %5 = xor i64 %4, 87960930222080
    %6 = inttoptr i64 %5 to i8*
    %_msld = load i8, i8* %6, align 1
    %7 = and i8 %_msld, 3
    %8 = and i8 %3, 3
    %9 = xor i8 %7, -1
    %10 = and i8 %9, %8
    %11 = icmp eq i8 %10, 0
    %12 = icmp ne i8 %7, 0
    %_msprop_icmp = and i1 %12, %11
    %tobool.not = icmp eq i8 %8, 0
    br i1 %_msprop_icmp, label %13, label %14, !prof !2

  13:                                                 ; preds = %2
    call void @__msan_warning_with_origin(i32 0) #2
    br label %14

  14:                                                 ; preds = %2, %13
    br i1 %tobool.not, label %if.end, label %if.then

  if.then:                                            ; preds = %14
    store i64 0, i64* @__msan_va_arg_overflow_size_tls, align 8
    tail call void (...) @bar() #3
    br label %if.end

  if.end:                                             ; preds = %if.then, %14
    ret void
```

+2.7x extra instructions (shown in red)

Origin tracking

Secondary 1:1 mapping: every aligned 4 bytes are assigned a 4-byte origin ID

- allocation stack traces stored in a hashtable
- if 4 bytes have zero shadow, origin is also zero
- origins are propagated along with shadow
  - if an operation produces a poisoned value, pick the origin of either argument that is poisoned
- origin chaining:
  - if a poisoned value is written to memory, create a new origin ID referencing the old one

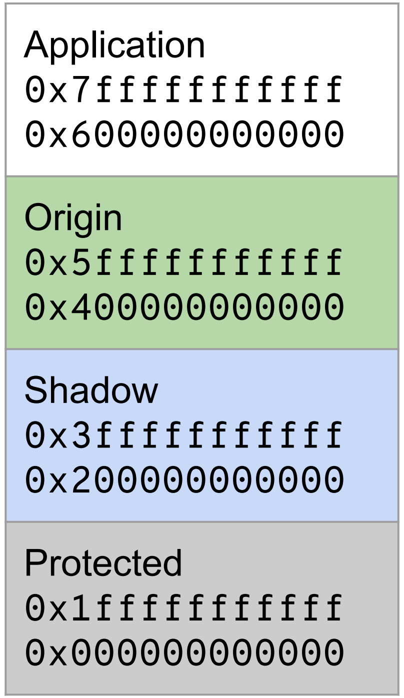
Shadow memory layout

```
Origin(Addr) = (Addr & ~3) - 0x200000000000
```

```
Shadow(Addr) = Addr - 0x400000000000
```

```
Shadow(Origin(Addr)) = Origin(Shadow(Addr))
```

- protected to prevent shadow corruption



How it looks like in the memory

```
char *buf = malloc(8);  // unwind stack here
                        // assume its hash is 0xC1CB9D05
buf:            ?? ?? ?? ?? ?? ?? ?? ??
Shadow(buf):    FF FF FF FF FF FF FF FF
Origin(buf):    05 9D CB C1|05 9D CB C1

strcpy(buf, "Hello!");
buf:            65 48 6C 6C 21 6F 00 ??
Shadow(buf):    00 00 00 00 00 00 00 FF
Origin(buf):    00 00 00 00|05 9D CB C1
```

Example use

```
% cat umr2.cc
#include <stdio.h>

int main(int argc, char** argv) {
  int* a = new int[10];
  a[5] = 0;
  volatile int b = a[argc];
  if (b)
    printf("xx\n");
  return 0;
}

% clang umr2.cc -g -fsanitize=memory \
                  -fsanitize-memory-track-origins=2
```

Example userspace report

```
% ./a.out

WARNING: MemorySanitizer: use-of-uninitialized-value
  #0 0x7f7893912f0b in main umr2.cc:7
  #1 0x7f789249b76c in __libc_start_main libc-start.c:226

Uninitialized value was stored to memory at
  #0 0x7f78938b5c25 in __msan_chain_origin msan.cc:484
  #1 0x7f7893912ecd in main umr2.cc:6

Uninitialized value was created by a heap allocation
  #0 0x7f7893901cbd in operator new[](unsigned long) msan_new_delete.cc:44
  #1 0x7f7893912e06 in main umr2.cc:4
```

KernelMemorySanitizer

KernelMemorySanitizer (KMSAN)

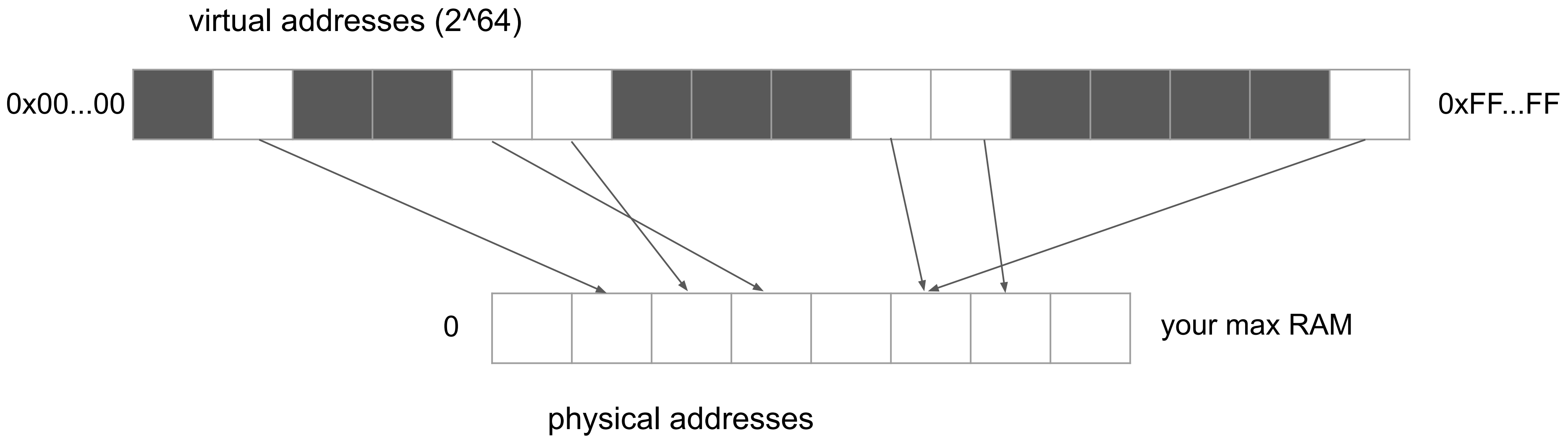
Kernel tool that detects uses of uninitialized values, maintained out-of-tree since 2017 (with rebases to every kernel release)

- `CONFIG_KMSAN=y`
- (almost the) same LLVM instrumentation: `-fsanitize=kernel-memory`
- runtime library to create and track uninit values
- x86_64 only
- detects uses of uninit values, infoleaks (to userspace, network, USB, DMA) and use-after-free bugs
- origin tracking is always on

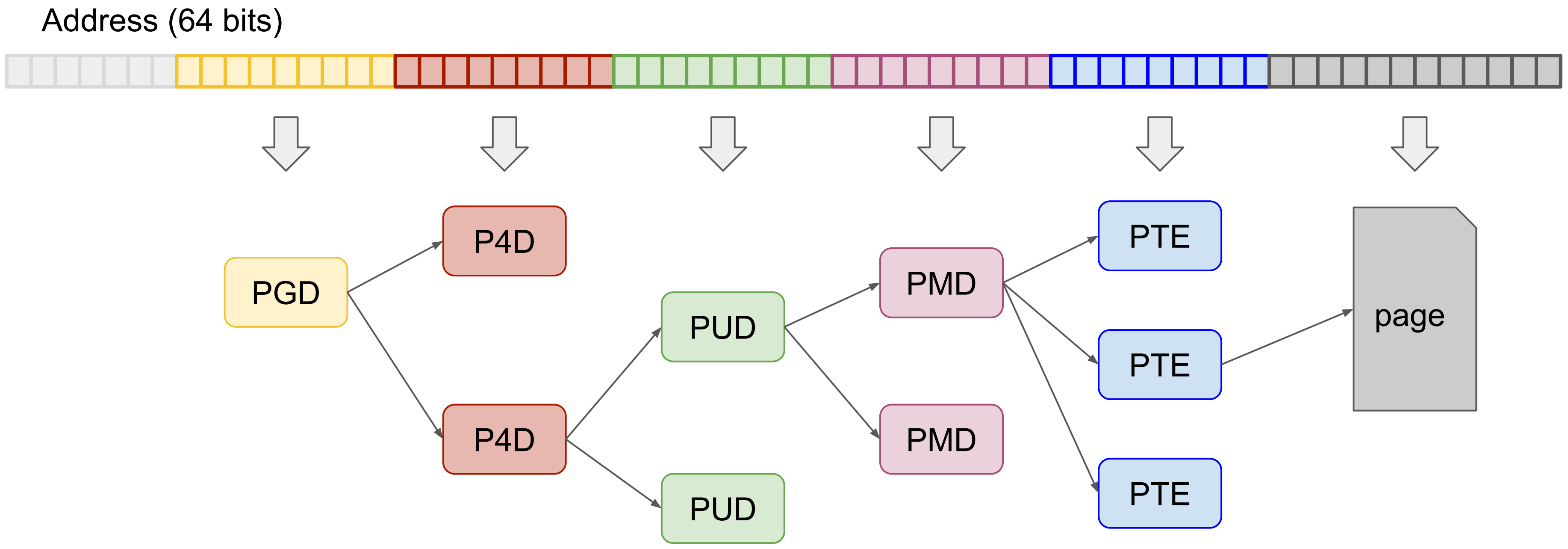
Allocating memory in the kernel

- memblock - at early boot time
- page allocator - to allocate page-aligned data
- SLAB/SLUB/SLOB - to allocate small objects in the heap:
  - kmalloc()
  - kmem_cache_alloc()
- vmalloc() - to allocate big consecutive memory chunks in virtual memory
- local variables - for small variables on the stack
- global variables - for global variables :)
- per-CPU variables - to avoid synchronization issues

Virtual memory in the x86_64 kernel



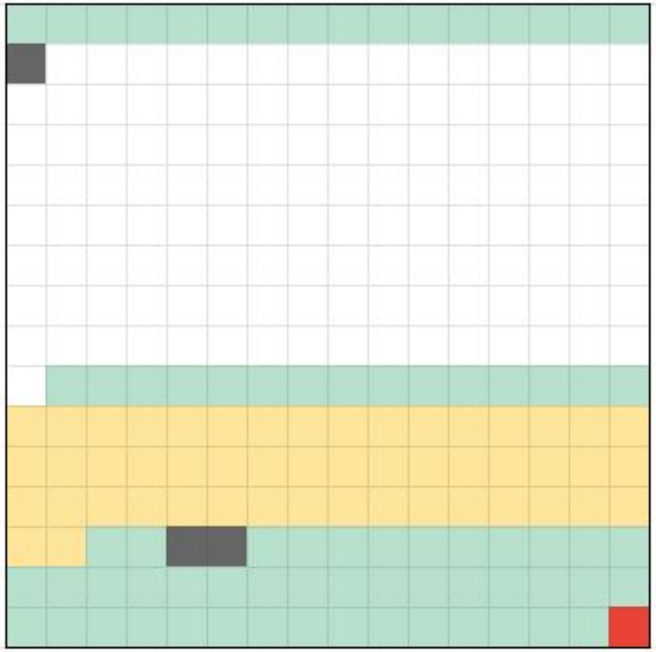
Multi-level page tables on x86_64



See also "Five-level page tables" at LWN.net

x86_64 kernel virtual memory layout (approximate)

🟩	- unused gaps
⬜	- direct mapping of physical memory.
🟨	- vmalloc area
⬛	- boring kernel mappings
🟥	- interesting kernel mappings (`.data`, `.text`, `cpu_entry_area` etc.)



1 square is 256TB, whereas the amount of physical memory on the machine is probably <1TB.

More accurate information in Documentation/x86/x86_64/mm.rst

Tracked memory

- Physical memory mapping
    - memblock
    - page allocator
        - heap
        - per-CPU variables
    - kernel-space memory
        - stack
        - globals (.data and .bss)
- vmalloc() memory
    - vmalloc() allocations
    - vmap() and ioremap()

Mapping metadata for kernel memory

Must reserve 2/3 physical memory in advance:

- Biggest problem: unknown size of phys memory
- cannot initialize early - by the time of KMSAN initialization the memory can get fragmented already

As a result

- hard to reserve memory in advance
  - need a dynamic mapping between kernel addresses and shadow addresses
- cannot calculate shadow addresses inline:
  - instrumentation different from userspace MSan (call to runtime instead of arithmetic instructions)
  - also slower

Let's allocate separate pages!

```
struct page {    // see include/linux/mm_types.h
  ...
  struct page *shadow, *origin;  // NULL for metadata pages
};


shadow(addr) = page_address(virt_to_page(addr)->shadow)) +
               addr & PAGE_SIZE

origin(addr) = page_address(virt_to_page(addr)->origin)) +
               ALIGN_DOWN(addr, 4) & PAGE_SIZE
```

Shadow mapping

```
shadow(addr) = page_address(virt_to_page(addr)->shadow)) +
               addr & PAGE_SIZE
```

`page_address()` - given a `struct page*`, return its virtual address

`virt_to_page()` - given a virtual address, return the corresponding `struct page`

At kernel startup...

- kmsan_initialize_shadow():
  - scan all existing memory ranges that don't have shadow yet
  - allocate shadow and origin pages for them from the memblock allocator.
  - covers .data and .bss
- kmsan_memblock_free_pages():
  - when memblock is torn down, it passes all unused pages to the SLUB allocator
    - hijack this process and use 2/3 of those pages as metadata for the remaining 1/3
    - the kernel will never see them again 😈

See also mm/kmsan/kmsan_init.c.

Shadow mapping

```
shadow(addr) = page_address(virt_to_page(addr)->shadow)) +
               addr & PAGE_SIZE
```

`page_address()` - given a `struct page*`, return its virtual address

`virt_to_page()` - given a virtual address, return the corresponding `struct page`

Ok, but not every virtual address has a `struct page`!

- actually, only pages in the physical mapping do

What about virtual mappings?

```
int vmap_pages_range_noflush(unsigned long addr,
                             unsigned long end,
                             pgprot_t prot, struct page **pages,
                             unsigned int page_shift);
```

- map pages to `[addr, end)` within `[VMALLOC_START, VMALLOC_END)`
  - fixed address range (1.25 PB) that can be shrinked

Splitting the vmalloc area

Before: `[VMALLOC_START, VMALLOC_END)` – 1.25PB vmalloc area

After:

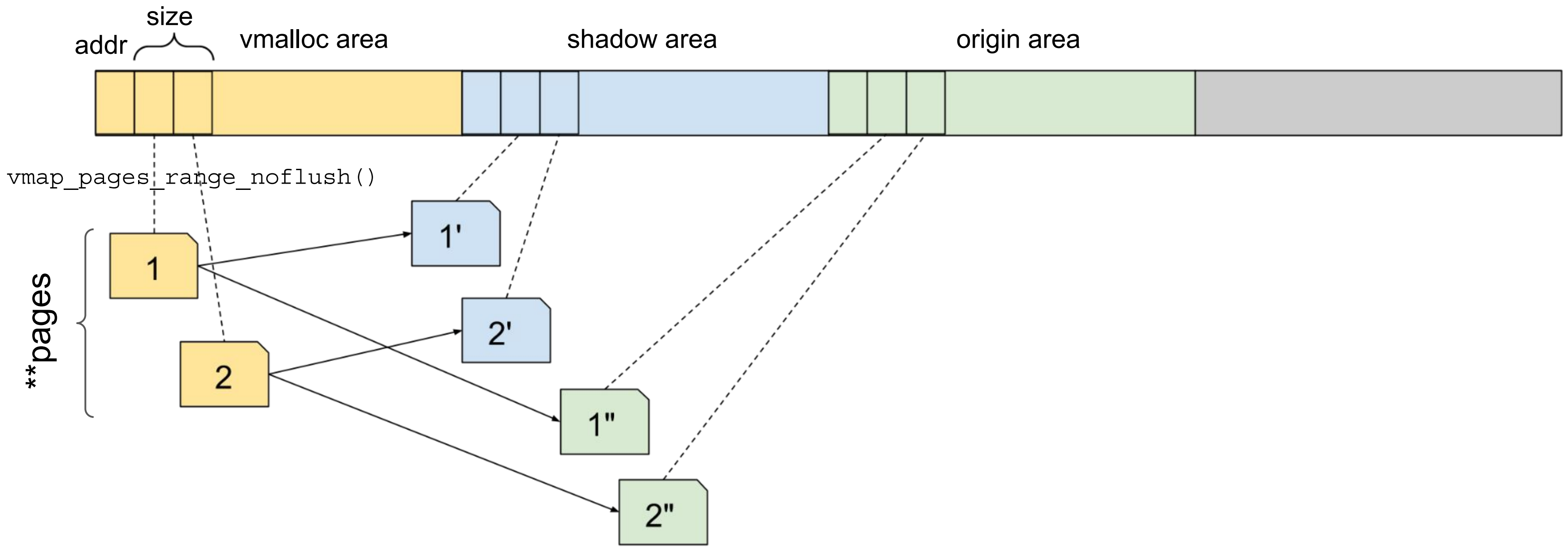
- `[VMALLOC_START, VMALLOC_END)` – 320TB vmalloc area (still a lot!)
- `[KMSAN_VMALLOC_SHADOW_OFFSET, KMSAN_VMALLOC_ORIGIN_OFFSET)`

  – 320TB shadow for vmalloc
- `[KMSAN_VMALLOC_ORIGIN_OFFSET, KMSAN_VMALLOC_META_END)`

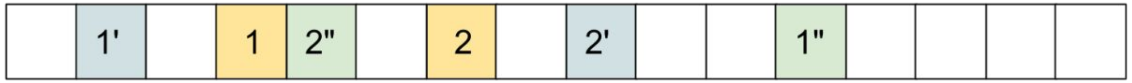
  – 320TB origins for vmalloc
- `[KMSAN_VMALLOC_META_END, VMEMORY_END)` – 320TB, almost unused

See arch/x86/include/asm/pgtable_64_types.h in KMSAN repo for details.

Creating a virtual mapping



Meanwhile in the physical page address range:



Corner case: cpu_entry_area

A per-CPU user-accessible data region that is used to jump into the kernel.

Located outside physical map and vmalloc area

- needs special handling

```
DEFINE_PER_CPU(char[CPU_ENTRY_AREA_SIZE], cpu_entry_area_shadow);

DEFINE_PER_CPU(char[CPU_ENTRY_AREA_SIZE], cpu_entry_area_origin);
```

kmsan_get_metadata(**address**, **is_origin**)

condition	shadow start	origin start
addr ∈ [VMALLOC_BEGIN, VMALLOC_END)	VMALLOC_SHADOW_OFFSET	VMALLOC_ORIGIN_OFFSET
addr ∈ [MODULES_VADDR, MODULES_END)	KMSAN_MODULES_SHADOW_START	KMSAN_MODULES_ORIGIN_START
addr ∈ CPU entry area	cpu_entry_area_shadow	cpu_entry_area_origin
virt_to_page(addr) is valid	virt_to_page(address) ->shadow	virt_to_page(address) ->origin
otherwise	NULL	NULL

KMSAN instrumentation at a glance

```
__msan_get_context_state()
__msan_poison_alloca(addr, size, descr)
__msan_unpoison_alloca(addr, size)
__msan_metadata_ptr_for_{load,store}_{1,2,4,8}(addr)
__msan_metadata_ptr_for_{load,store}_n(addr, size)
__msan_instrument_asm_store(addr, size)
__msan_{memcpy,memmove}(dst, src, size)
__msan_memset(dst, c, n)
__msan_chain_origin(origin)
__msan_warning(origin)
```

See also mm/kmsan/kmsan_instr.c (kernel part), MemorySanitizer.cpp (compiler part)

__msan_get_context_state()

```
struct kmsan_context_state {
  char param_tls[...];
  char retval_tls[...];
  char va_arg_tls[...];
  char va_arg_origin_tls[...];
  u64 va_arg_overflow_size_tls;
  depot_stack_handle_t param_origin_tls[...];
  depot_stack_handle_t retval_origin_tls;
};
```

- fully maintained by the compiler instrumentation, which inserts a call to `__msan_get_context_state()` at function prologue to get the `kmsan_context_state` pointer for the current task.

Parameter passing

```
int sum(int a, int b) {
  struct kmsan_context_state *kcs = __msan_get_context_state();
  int s_a = ((int)kcs->param_tls)[0];  // shadow of a
  int s_b = ((int)kcs->param_tls)[1];  // shadow of b
...
  result = a + b;
  s_result = s_a | s_b;
  ((int)kcs->retval_tls)[0] = s_result;  // returned shadow
  return result;
}
```

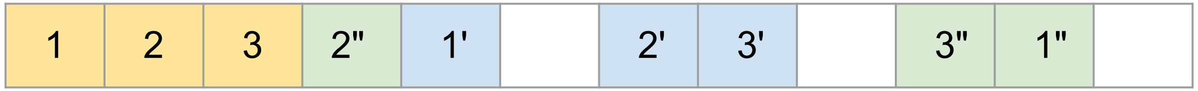
Handling locals

```
int sum(int a, int b) {
  int result;
  __msan_poison_alloca(&result, sizeof(int), "----result@sum");
  ...
}
```

Instrumenting loads and stores

```
struct shadow_origin_ptr {
  void *s, *o;
}

void check_flag(int *flag) {
  ...
  struct shadow_origin_ptr m =
         __msan_metadata_ptr_for_load_4(flag);
  if (m.s)
    __msan_warning(m.o);
  if (*flag) {
    // do something
  }
}
```

Accessing non-contiguous shadow



Userspace: shadow and origins for [a, b) and [b, c) are always contiguous.

Kernel: data and metadata pages can be interleaved

Cannot do something like __msan_write_shadow_4(addr, shadow_value)

- hard to pass values with sizeof() > 8

Every function reading and writing arbitrary-sized memory must operate with chunks <= PAGE_SIZE

- e.g. kmsan_memset(), kmsan_memcpy(), kmsan_memmove()

Inline metadata accesses

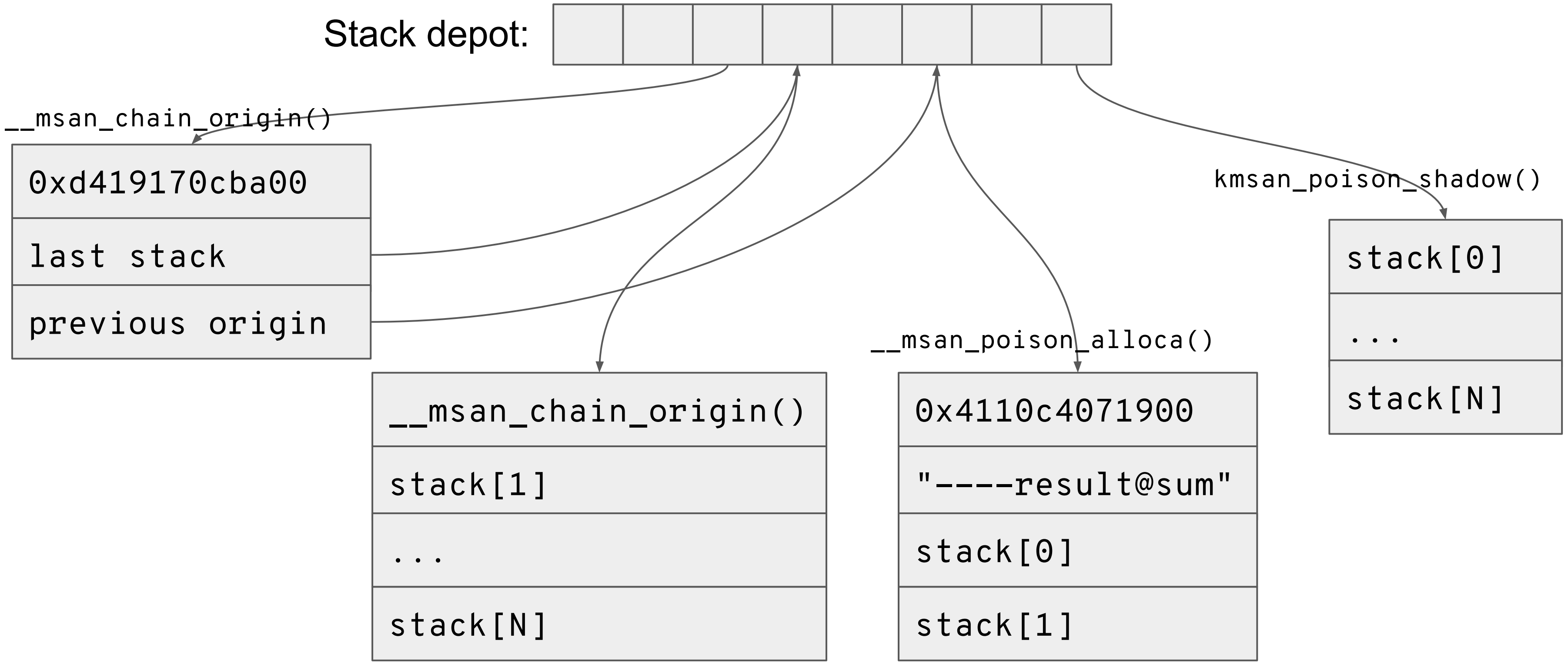
```
struct shadow_origin_ptr kmsan_get_shadow_origin_ptr(
    void *addr, u64 size, bool store)

if:
   kernel is booting up
   or we are in the runtime
   or metadata for [addr, addr+size) is non-contiguous
   or kmsan_get_metadata() returns NULL
then:
   return pointers to dummy metadata load/store pages
else:
   return { kmsan_get_metadata(addr, false),
            kmsan_get_metadata(addr, true) }
```

Handling assembly: __msan_instrument_asm_store()

Inline assembly may initialize its arguments, but LLVM IR has little idea about it

```
void arch_atomic64_inc(atomic64_t *v) {
    asm volatile(LOCK_PREFIX "incq %0"
        : "=m" (v->counter)
        : "m" (v->counter) : "memory");
    __msan_instrument_asm_store(v->counter, 8);
}
```

Origin handling: stack depot



Error reporting

`kmsan_report(...)` dumps the access stack and unrolls and prints the origin chain

- triggered by `__msan_warning(origin)` or
- `kmsan_internal_check_memory(addr, size, user_addr, reason)`
  - check shadow for [addr, addr+size), report any nonzero shadow byte
  - origin is picked from the checked memory
- Reasons:
  - REASON_COPY_TO_USER: report a "kernel-infoleak", include `user_addr`
  - REASON_SUBMIT_URB: report a "kernel-usb-infoleak"
  - REASON_ANY: report an "uninit-value" error
  - one of the lower origin bits also stores the use-after-free flag

Example report

```
BUG: KMSAN: uninit-value in ieee802154_rx+0x19c4/0x20e0
net/mac802154/rx.c:284

Call Trace:
 __msan_warning+0x5c/0xa0 mm/kmsan/kmsan_instr.c:197
 ieee802154_subif_frame net/mac802154/rx.c:69 [inline]
 __ieee802154_rx_handle_packet net/mac802154/rx.c:212 [inline]
 ieee802154_rx+0x19c4/0x20e0 net/mac802154/rx.c:284

Uninit was stored to memory at:
 kmsan_internal_chain_origin+0xad/0x130 mm/kmsan/kmsan.c:289
 __msan_memcpy+0x46/0x60 mm/kmsan/kmsan_instr.c:110
 ieee802154_parse_frame_start net/mac802154/rx.c:156 [inline]
 ieee802154_rx+0xd92/0x20e0 net/mac802154/rx.c:284

Local variable ----hdr.i@ieee802154_rx created at:
 __ieee802154_rx_handle_packet net/mac802154/rx.c:196 [inline]
 ieee802154_rx+0xb3d/0x20e0 net/mac802154/rx.c:284
```

Subsystem-specific hooks: mm/kmsan/kmsan_hooks.c

- task creation/deletion: set up per-task data (`kmsan_context_state` etc.)
  - `kmsan_task_create()`, `kmsan_task_exit()`
- heap management: poison/unpoison memory allocations
  - `kmsan_slab_alloc(), kmsan_slab_free(), kmsan_kmalloc_large(), kmsan_kfree_large()`
- handling virtual mappings: create shadow/origin mappings
  - `kmsan_ioremap_page_range(), kmsan_iounmap_page_range(), kmsan_unmap_kernel_range()`
- copying data to/from kernel: check for infoleaks, unpoison incoming data
  - `kmsan_copy_to_user(), kmsan_check_skb(), kmsan_handle_urb(), kmsan_handle_dma()`
- entry points: `func(struct pt_regs *regs)`
  - `kmsan_unpoison_pt_regs()`

KMSAN checks API: include/linux/kmsan-checks.h

- `KMSAN_INIT_VALUE(value)`
  - take uninitialized 1-, 2-, 4-, 8-byte value, return initialized value

-
  ```
  kmsan_poison_shadow(void *address, size_t size,
                      gfp_t flags)
  ```
- `kmsan_unpoison_shadow(void *address, size_t size)`
- `kmsan_check_memory(void *address, size_t size)`

Disabling instrumentation: `__no_sanitize_memory`

aka `__attribute__((no_sanitize("kernel-memory")))`

Drops most of instrumentation, but prevents false positives:

- unpoison all locals (__msan_unpoison_alloca(addr, size))
- do not propagate metadata
- all writes unpoison memory
- skip all shadow checks
- return initialized values

Functions marked with `__no_sanitize_memory` may still call normally instrumented functions.

Disabling instrumentation: KMSAN_SANITIZE

Makefile directive that removes `-fsanitize=kernel-memory` from CFLAGS:

```
KMSAN_SANITIZE_cpu_entry_area.o := n    # don't instrument
                                        # this file
KMSAN_SANITIZE := n          # don't instrument this dir
```

Reasons:

- too early to call KMSAN runtime (e.g. bootloader)
- code called from KMSAN runtime (lockdep, kcov, stackdepot)
- mm/kmsan/*.c

Instrumented vs. non-instrumented code

By Luv Ya! Cookies&Cakes - Own work, CC BY-SA 4.0,

Drawing the border

- limited use of `KMSAN_SANITIZE`
  - Boot-time code, KMSAN runtime + a handful of other cases
- `__no_sanitize_memory` for several tricky functions
  - scheduler, page heap, SLUB
- `kmsan_enter_runtime()/kmsan_leave_runtime()` in hooks
  - surround places that may call instrumented code
  - avoid reentrancy
- explicitly poison/unpoison/check memory used by non-instrumented code
  - e.g. when calling C functions from assembly
- Disable non-instrumentable code:
  - no JIT in BPF, no hardware-accelerated crypto

Automatic memory initialization

Uninits are unlikely to disappear

... the problem of leaking uninitialized kernel memory to user space is not caused merely by simple programming errors. Instead, it is deeply rooted in the nature of the C programming language, and has been around since the very early days of privilege separation in operating systems.

- Mateusz Jurczyk, Project Zero.

Alternative to KMSAN: initialize all memory

- No information leaks
- Deterministic execution

Already piloted in user-facing products by Microsoft, Google and reportedly Apple.

Stack initialization

Clang: `CONFIG_INIT_STACK_ALL_ZERO=y`

- zero-initializes all locals

GCC: `CONFIG_GCC_PLUGIN_STRUCTLEAK_BYREF_ALL=y`

- zero-initializes all locals passed by reference

Performance penalty is 0-4% (typically 0%)

More options in security/Kconfig.hardening

O brave new world!

So I'd like the zeroing of local variables to be a native compiler option, so that we can (_eventually_ - these things take a long time) just start saying "ok, we simply consider stack variables to be always initialized".

Linus Torvalds

Heap initialization

These are in your kernel already:

`init_on_alloc=1` (also `INIT_ON_ALLOC_DEFAULT_ON=y`)

- zero-initializes allocated memory
- cache-friendly, noticeably faster (0%-1.5%, but up to 7%)

`init_on_free=1` (also `INIT_ON_FREE_DEFAULT_ON=y`)

- zero-initializes freed memory
- minimizes the lifetime of sensitive data
- slower (0%-3%, up to 8%-10% and more)

See also: PAX_MEMORY_SANITIZE

Opt-outs are inevitable

Again - I don't think we want a world where everything is force-initialized. There _are_ going to be situations where that just hurts too much. But if we get to a place where we are zero-initialized by default, and have to explicitly mark the unsafe things (and we'll have comments not just about how they get initialized, but also about why that particular thing is so performance-critical), that would be a good place to be.

Linus Torvalds

Plan for now

- send KMSAN for upstream review
- continue reporting uninit bugs
- continue pushing for total memory initialization

Assorted useful links (check other pages as well!)

KASAN, KCSAN, KMSAN, KFENCE - kernel bug detection tools

LLVM Language Reference Manual - for those who want to look closer at the compiler instrumentation.

Linux Insides - a Gitbook by 0xAX on Linux internals

Elixir Cross Referencer - indexed and cross-referenced Linux source

Q&A

Thank you!