



Kernel Sanitizers Office Hours

Hosted by:

Aleksandr Nogikh (Google)

Alexander Potapenko (Google)

Taras Madan (Google)

Justin Stitt (Google)

Kees Cook (Google)

Marco Elver (Google)

Pimyn Girgis (Google)



LINUX
PLUMBERS
CONFERENCE

PRAGUE, CZECHIA / OCT. 5-7, 2026

Agenda

1. Kernel Sanitizers Primer

- Kernel Address Sanitizer (KASAN)
- Kernel Electric Fence (KFENCE)
- Kernel Memory Sanitizer (KMSAN)
- Kernel Concurrency Sanitizer (KCSAN)
- Undefined Behaviour Sanitizer (UBSAN)
- Type-Safe Allocations & Slab Partitioning
- Compiler-Based Context Analysis

2. 2026 Updates & Ongoing Discussions

3. Discussion and Questions

Kernel Sanitizers Primer

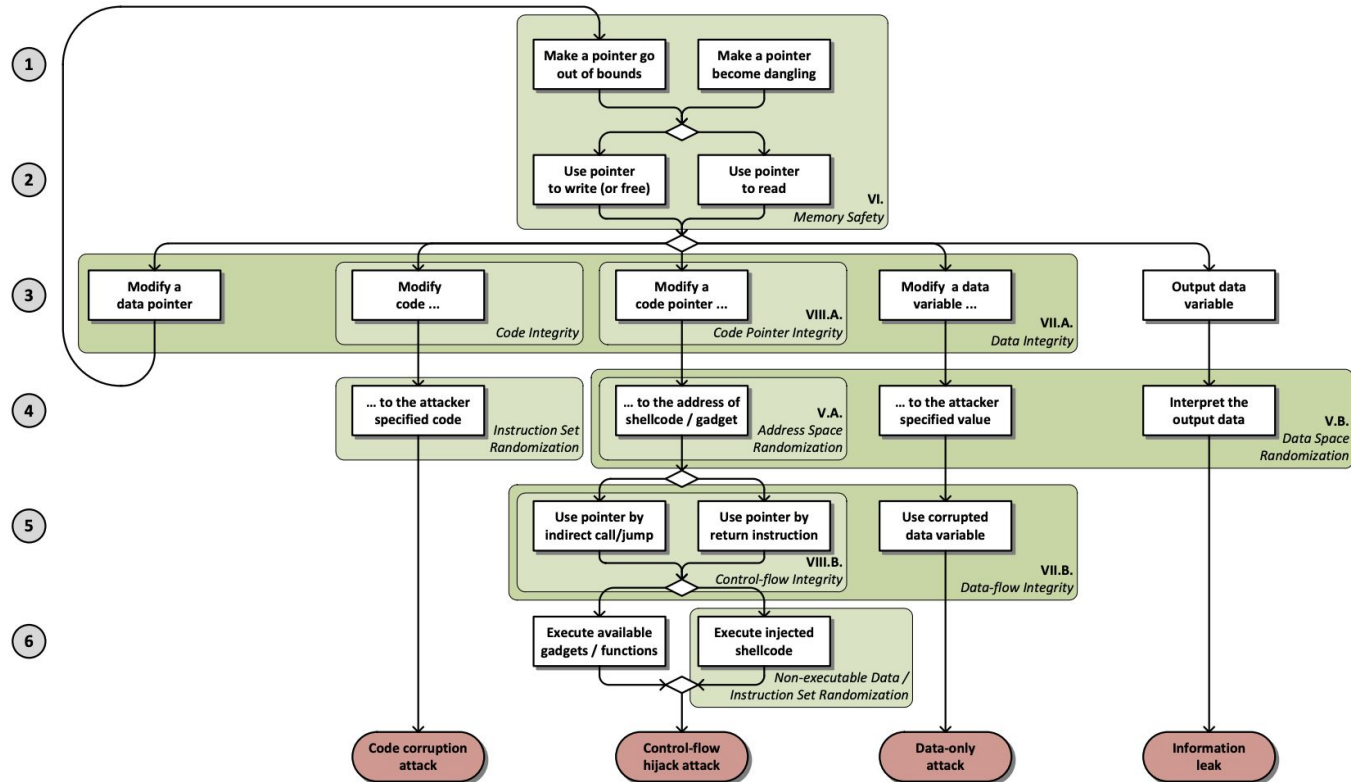
Sanitizers: Deterministic Oracles for LLM Bug Fixing

Silent bugs and delayed crashes are nearly impossible for LLM agents to verify

Why sanitizers are essential for AI-driven bug reproduction and repair:

- *Turn silent corruption into immediate, deterministic signals*
 - Catch bugs at the exact faulting instruction instead of corrupting state silently
 - Eliminate flaky crashes so agents get a reliable pass/fail reproduction oracle
- *Provide rich, structured diagnostics for root-cause reasoning*
 - Supply alloc/free/fault stack traces, slab object types, and memory/lock state
 - Give LLMs the full causal chain needed to localize bugs without guesswork
- *Enable closed-loop verification for automated patch generation*
 - Reproducer + KASAN / KMSAN / KCSAN / UBSAN / KUnit confirms a candidate fix works
 - Shift-left compiler checks (Context Analysis, bounds) reject broken patches early

Memory Safety Errors



Memory-safety errors are the root cause of most security attacks [Szekeres et al. Oakland'13]

Out-of-bounds accesses

- Accesses memory beyond the allocated memory
 - No bounds checking by default
 - Compiler may sometimes warn (if it can infer array size)
- May read random data, or corrupt other kernel state!
 - Can be exploited to leak memory, or control kernel in unintended ways!

```
void print_upper_buggy(const char *str)
{
    char buf[10];
    strcpy(buf, str); // unchecked strcpy!
    for (char *c = buf; *c; ++c)
        *c = toupper(*c);
    pr_info("%s\n", buf);
}
```

Heap use-after-free

- Accesses recently unallocated heap memory
 - Memory may already have been recycled
- May read random data, or corrupt other kernel state!
 - Can be exploited to leak memory, or control kernel in unintended ways!

```
void print_upper_buggy(const char *str)
{
    char *buf = kmalloc(strlen(str), GFP_KERNEL);
    if (WARN_ON(!buf)) return;
    strcpy(buf, str);
    for (char *c = buf; *c; ++c)
        *c = toupper(*c);
    kfree(buf);           // whoops!
    pr_info("%s\n", buf); // use-after-free!
}
```

Stack use-after-return

- Access to memory in invalid stack frame
 - Stack memory may already have been reused in the next call
- May read random data, or corrupt other kernel state!
 - Can be exploited to leak memory, or control kernel in unintended ways!

```
const char *strtoupper_buggy(const char *str)
{
    char buf[64];
    strncpy(buf, str, sizeof(buf));
    for (char *c = buf; *c; ++c)
        *c = toupper(*c);
    return buf; // return of pointer to stack var!
}
```

Kernel Address Sanitizer (KASAN)

Detects: out-of-bounds accesses, heap use-after-free, and stack use-after-returns

Usage [docs.kernel.org/dev-tools/kasan.html]:

- *Generic (default): CONFIG_KASAN=y*
 - For debugging and testing kernels
 - Not recommended for production kernels!
- *Software tags: CONFIG_KASAN=y + CONFIG_KASAN_SW_TAGS=y*
 - For debugging and testing kernels
 - Lower overhead vs. generic, but also not recommended for production kernels!
 - Generic & Software Tag-Based modes support Rust (CONFIG_RUST=y)
- *Hardware tags: CONFIG_KASAN=y + CONFIG_KASAN_HW_TAGS=y*
 - Currently requires *Arm64 Memory Tagging Extension (MTE)*
 - Usable in production kernels!
 - Supports store-only checking via `kasan.write_only=on`

Kernel Electric Fence (KFENCE)

Detects: heap out-of-bounds accesses, heap use-after-free

Usage [docs.kernel.org/dev-tools/kfence.html]:

- CONFIG_KFENCE=y
- Low-overhead **sampling**-based memory safety error detector
- *Designed for production kernels deployed at scale!*
- Configurable fault behavior: kfence.fault={report,oops,panic}
- Burst mode (kfence.burst=N) for higher-probability bug reproduction
- Automatically disabled when KASAN HW tags (MTE) are enabled

Uses of uninitialized memory

- Access memory that has not been initialized
- May read random data or even old data from recycled memory!
 - Could be exploited to leak sensitive data!

```
void hello_tux_buggy(const char *name)
{
    char buf[10];
    if (name) strcpy(buf, name, sizeof(buf));
    if (buf[0] == 't' && buf[1] == 'u' && buf[2] == 'x')
        pr_info("hello world\n");
}
```

Kernel Memory Sanitizer (KMSAN)

Detects: uses-of-uninit, kernel-user-space information leaks

Usage [docs.kernel.org/dev-tools/kmsan.html]:

- CONFIG_KMSAN=y
- For debugging and testing kernels
- Not recommended for production kernels!



To mitigate stack uninit bugs in production, use:
CONFIG_INIT_STACK_ALL_ZERO=y (-ftrivial-auto-var-init=zero)

Data Races

Data Races in the Linux Kernel

Data races (✗) occur if:

- Concurrent conflicting accesses
 - they conflict if they access the same location and at least one is a write, ...
- **and** at least one is a plain access.

Thread 0	Thread 1
✗ ... = x + 1;	x = 0xf0f0;
✗ ... = x + 1;	WRITE_ONCE(x, 0xf0f0);
✗ ... = READ_ONCE(x) + 1;	x = 0xf0f0;
✗ ... = READ_ONCE(x) + 1;	x++;
✗ x = 0xff00;	x = 0xff;
✓ ... = READ_ONCE(x) + 1;	WRITE_ONCE(x, 0xf0f0);
✓ WRITE_ONCE(x, 0xff00);	WRITE_ONCE(x, 0xff);

Kernel Concurrency Sanitizer (KCSAN)

Usage [docs.kernel.org/dev-tools/kcsan.html]:

- CONFIG_KCSAN=y
- For debugging and testing kernel
- Not recommended for production kernels!
- **Suggested config:** CONFIG_KCSAN_STRICT=y (since 5.17)
 - "Strict" LKMM rules (but as of 7.3 still noisy)
 - Includes weak memory modeling (detect missing memory barriers)

Other Types of Undefined Behavior

“Undefined” Behaviour Sanitizer: CONFIG_UBSAN=y

Behavioral toggle:

- Trap instead of warning: CONFIG_UBSAN_TRAP=y

Production ready:

- Detect out of range shifts: CONFIG_UBSAN_SHIFT=y
- Detect out of bounds array/pointer indexes: CONFIG_UBSAN_BOUNDS=y

Pedantic:

- Non-boolean type used as bool: CONFIG_UBSAN_BOOL=y
- Value assigned to enum not in enum declaration: CONFIG_UBSAN_ENUM=y

Under development:

- Semantic Fault, "[Overflow Behavior Types](#)" that are hooked up to CONFIG_UBSAN_INTEGER_WRAP=y

Trap instead of warning: UBSAN_TRAP=y

For the various individual tests under the UBSAN prefix, the TRAP setting determines how the kernel should behave when detecting an issue. Normally, a warning with details is reported, and execution continues without correcting the issue (but the kernel image is about 5% larger from all the text and handling):

```
UBSAN: array-index-out-of-bounds in drivers/gpu/drm/v3d/v3d_sched.c:320:3  
index 7 is out of range for type '__u32 [7]'
```

Under UBSAN_TRAP=y, a much more terse BUG is reported, and the thread is terminated:

```
Internal error: UBSAN: shift out of bounds: 00000000f2005514 [#1] PREEMPT SMP
```

See [warn_limit](#) sysctl for a more flexible way to turn WARN into BUG

Detect out of range shifts: UBSAN_SHIFT=y

```
int negative = -1;
u16 bit_field = ...;
...
use_some_bits(bit_field << negative); // catch “negative” shift
```

```
int has_sign = INT_MAX;
...
use_some_bits(has_sign << 4); // catch shift of signed bit
```

<https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/log/?qt=grep&q=shift-out-of-bounds>

228 fixes in 8 years

Detect out of bounds array indexes: UBSAN_BOUNDS=y

```
int array[16];
int index = 16;
...
do_something(array[index]); // catch index outside of [0..15]

struct foo {
    int numBars;
    struct bar bar[] __counted_by(numBars);
} *p = kmalloc_flex(*p, bar, 8);
...
do_something(p->bar[index]); // catch index outside of [0..(p->numBars-1)]
```

<https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/log/?qt=grep&q=index-out-of-bounds>

177 fixes in 6 years

Depends on -fstrict-flex-arrays=3, real flexible arrays with __counted_by() (>550 structs), pointer bounds via __counted_by_ptr(), and kmalloc_flex().

Type-Safe Allocations & Slab Hardening

Compiler-assisted bounds enforcement, type-aware allocators, and heap isolation

Advancements in 2026:

- *Type-aware allocation APIs: `kmalloc_obj()`, `kmalloc_objs()`, `kmalloc_flex()`*
 - Treewide adoption (>8,000 files) replacing error-prone `kmalloc(sizeof(...))`
 - `kmalloc_flex()` auto-initializes `__counted_by()` counters via `__flex_counter()`
- *Expanded bounds checking: `__counted_by()` & `__counted_by_ptr()`*
 - `__counted_by_ptr()` brings runtime bounds checking to struct pointer members
 - Pitfalls: delayed counter updates, byte vs. element counts, untrusted shared memory
- *Type-based slab partitioning: `CONFIG_KMALLOC_PARTITION_TYPED=y`*
 - Uses Clang 22+ allocation tokens (`__builtin_infer_alloc_token()`)
 - Deterministically isolates types & separates pointer-containing objects from data buffers

Semantic Faults

Semantic Faults

- Faults that don't cause “undefined behavior”, but still result in system errors
- System deviates from its intended behavior
- *Who defines intended behavior?*
 - Formal specification, reference implementation, documentation, manual
 - *Worst case:* not written down, but in programmer's head
- Much harder to detect
 - Tests
 - **Assertions**
 - **Defensive programming style**
 - ...

UBSAN_INTEGER_WRAP=y Detect wrapping arithmetic

- For the Linux kernel, we need "[idiom exclusions](#)" to avoid instrumenting cases where wrap-around is either already checked, or is not part of program flow:
 - `if (var + offset < var)`
 - `while (var--)`
 - `-1UL, -2UL, ...`
 - Clang: [19+](#); GCC: **Needed**
- Implement "Overflow Behavior Type" annotations in kernel for explicitly declared overflow behavior (wrap or trap), starting with `size_t` (replacing `UBSAN_INTEGER_WRAP / __signed_wrap`):
 - `typedef unsigned long __ob_trap size_t;`
 - Clang: [22?](#); GCC: **Needed**

Concurrency bugs that are not data races

Thread 0

```
spin_lock(&update_foo_lock);  
/* Careful! There should be no other  
writers to shared_foo! Readers ok. */  
WRITE_ONCE(shared_foo, ...);  
spin_unlock(&update_foo_lock);
```

Concurrency bugs that are not data races

Thread 0

```
spin_lock(&update_foo_lock);  
/* Careful! There should be no other  
writers to shared_foo! Readers ok. */  
WRITE_ONCE(shared_foo, ...);  
spin_unlock(&update_foo_lock);
```

Thread 1

```
/* update_foo_lock does not  
need to be held! */  
... = READ_ONCE(shared_foo);
```

Concurrency bugs that are not data races

Thread 0	Thread 1	Thread 2
<pre>spin_lock(&update_foo_lock); /* Careful! There should be no other writers to shared_foo! Readers ok. */ WRITE_ONCE(shared_foo, ...); spin_unlock(&update_foo_lock);</pre>	<pre>/* update_foo_lock does not need to be held! */ ... = READ_ONCE(shared_foo);</pre>	<pre>/* Bug! */ WRITE_ONCE(shared_foo, 42);</pre>

Concurrency bugs that are not data races

Thread 0	Thread 1	Thread 2
<pre>spin_lock(&update_foo_lock); /* No other writers to shared_foo. */ ASSERT_EXCLUSIVE_WRITER(shared_foo); WRITE_ONCE(shared_foo, ...); spin_unlock(&update_foo_lock);</pre>	<pre>/* update_foo_lock does not need to be held! */ ... = READ_ONCE(shared_foo);</pre>	<pre>/* Bug! */ WRITE_ONCE(shared_foo, 42);</pre>

How KCSAN can help find more bugs

- ASSERT_EXCLUSIVE family of macros:
 - Specify properties of concurrent code, where bugs are not normal data races.

	concurrent writes		concurrent reads
ASSERT_EXCLUSIVE_WRITER(<i>var</i>) ASSERT_EXCLUSIVE_WRITER_SCOPED(<i>var</i>)	✗		✓
ASSERT_EXCLUSIVE_ACCESS(<i>var</i>) ASSERT_EXCLUSIVE_ACCESS_SCOPED(<i>var</i>)	✗		✗
ASSERT_EXCLUSIVE_BITS(<i>var</i> , <i>mask</i>)	~mask ✓	mask ✗	✓

Compiler-Based Context Analysis

Compiler-Based Context Analysis

Detects: locking rule and execution context violations at compile time

Usage [docs.kernel.org/dev-tools/context-analysis.html]:

- *Static analysis: CONFIG_WARN_CONTEXT_ANALYSIS=y (Clang 23+)*
 - Opt-in per subsystem/object via Makefile: `CONTEXT_ANALYSIS := y`
 - Replaces legacy Sparse lock context tracking with compiler-enforced rules
- *Key annotations (<linux/compiler-context-analysis.h>):*
 - Data: `__guarded_by(&lock)`, `__pt_guarded_by(&lock)`; Escape: `context_unsafe(...)`
 - Functions: `__must_hold(&lock)`, `__acquires(&lock)`, `__releases(&lock)`
- *Synchronization primitives & runtime integration:*
 - `spinlock_t`, `rwlock_t`, `mutex`, `rt_mutex`, `rwsem`, `seqlock`, `RCU`, `SRCU`, `local_lock`, `ww_mutex`
 - Integrates with `guard()`, `guard(spinlock_init)(&lock)`, `lockdep_assert_held()`, `data_race()`

Context Analysis in Action: CONTEXT_ANALYSIS := y

```
struct my_data {
    spinlock_t lock;
    int counter __guarded_by(&lock);
};

void init_my_data(struct my_data *d) {
    guard(spinlock_init>(&d->lock); // scoped init guard
    d->counter = 0;                  // OK: init context active
}

void update_locked(struct my_data *d) __must_hold(&d->lock) {
    d->counter++;                    // OK: caller holds lock
}
```

Enabled in: kernel/sched/, kernel/futex/, kernel/locking/, kernel/kcsan/, mm/kfence/, block/, drivers/nvme/, ...

Agenda

1. Kernel Sanitizers Primer

- Kernel Address Sanitizer (KASAN)
- Kernel Electric Fence (KFENCE)
- Kernel Memory Sanitizer (KMSAN)
- Kernel Concurrency Sanitizer (KCSAN)
- Undefined Behaviour Sanitizer (UBSAN)
- Type-Safe Allocations & Slab Partitioning
- Compiler-Based Context Analysis

2. **2026 Updates & Ongoing Discussions**

3. **Discussion and Questions**

Discussion

Future Ideas for Sanitizers

1. Context Analysis: Expand `CONTEXT_ANALYSIS := y` across more kernel subsystems
2. Bounds checking: Resolve `__counted_by()` / `kmalloc_flex()` corner cases & expand `__counted_by_ptr()`
3. KMSAN infoleak annotations & KCSAN task interleaving control / reproducers
4. Identifying high-impact outstanding Feature Requests / Bugs
 - a. https://bugzilla.kernel.org/buglist.cgi?bug_status=__open__&component=Sanitizers
 - b. Contributions welcome!

How to shift-left more?

Success criteria: Kernel Sanitizers find fewer and fewer bugs!

Cheaply and easily detect bugs before ever being committed:

- Easy way to run tests quickly with sanitizers? (run as part of default **make**?)
 - Must be very very fast!
- More always-on compiler diagnostics / static analysis?
 - *Locking- and Context Analysis*: CONFIG_WARN_CONTEXT_ANALYSIS=y (CONTEXT_ANALYSIS := y)
 - Type-safe allocations (kmalloc_obj / kmalloc_flex) & CONFIG_KMALLOC_PARTITION_TYPED=y
- Rust
 - New subsystems/drivers; KASAN (Generic & SW_TAGS), KCFI, KCOV, Rust inline helpers

Q & A

Bonus Material

Data Races

Data Races

- C-language and compilers evolved oblivious to concurrency
- Optimizing compilers are becoming more creative
 - load tearing,
 - store tearing,
 - load fusing,
 - store fusing,
 - code reordering,
 - invented loads,
 - invented stores,
 - ... and more!



Need to tell compiler about concurrent code

Data Races

Defined via language's memory consistency model:

- C-language and compilers no longer oblivious to concurrency:
 - C11 introduced memory model: "data races cause undefined behaviour"
 - **Not Linux's model!**
- Linux has its own memory model, giving semantics to concurrent code
 - **Linux Kernel Memory Consistency Model (LKMM)**
 - Implemented by relying on parts of the C standard, the two C implementations (GCC & Clang/LLVM), architecture-specific code, and also coding guidelines (along with some luck that none of the supported C compilers "miscompile" our concurrent code)

Data Races

Data-race-free code has several benefits:

1. **Well-defined.** Avoids having to reason about compiler and architecture.
 - Avoid having to reason "Is this data race benign?"
2. **Fewer bugs.** Data races can also indicate higher-level race-condition bugs.
 - E.g. failing to synchronize accesses using spinlocks, mutexes, RCU, etc.
3. **Prevent bugs,** and countless hours debugging elusive race conditions!

Data Races in the Linux Kernel

Data races (✗) occur if:

- Concurrent conflicting accesses
 - they conflict if they access the same location and at least one is a write, ...
- **and** at least one is a plain access.

Thread 0	Thread 1
✗ ... = x + 1;	x = 0xf0f0;
✗ ... = x + 1;	WRITE_ONCE(x, 0xf0f0);
✗ ... = READ_ONCE(x) + 1;	x = 0xf0f0;
✗ ... = READ_ONCE(x) + 1;	x++;
✗ x = 0xff00;	x = 0xff;
✓ ... = READ_ONCE(x) + 1;	WRITE_ONCE(x, 0xf0f0);
✓ WRITE_ONCE(x, 0xff00);	WRITE_ONCE(x, 0xff);

Intentional Data Races

- The Linux kernel says that data races do not result in undefined behaviour of the whole kernel
- Locally "undefined" behaviour: where code still operates correctly even with potentially random data, data races are tolerated (truly "benign" data races)
- Mark such data races with "data_race(..data-racy expression ..)"
 - Helps tooling understand they are intentional
 - Document intent (e.g. debugging-only checks)

For more guidance: tools/memory-model/Documentation/access-marking.txt

2026 Updates & Ongoing Discussions

KASAN: 2026 Updates & Ongoing Discussions

Architecture expansion, eBPF/Rust coverage, and core fixes

Merged updates & active LKML / RFC threads:

- *x86 Tag-Based KASAN via Intel LAM ([PATCH v12])*
 - Uses 4-bit pointer tags in bits [60:57] via Linear Address Masking (Clang, outline)
 - ~2x lower memory overhead on x86_64 vs. Generic mode (44 GB vs. 82 GB on 512 GB)
- *Expanded runtime checks: Rust, eBPF JIT, and s390 lazy MMU*
 - Rust SW_TAGS support landed; CONFIG_BPF_JIT_KASAN merged in bpf-next (x86_64)
 - vrealloc() poisoning & overflow fixes; s390-specific lazy MMU PTE poisoning ([PATCH v8])
- *Quarantine fixes, intra-object redzones & Stack Depot RFC*
 - Fixed quarantine CPU hotplug shrink race, size accounting & SW_TAGS bug reports
 - Discussions: skb_shared_info redzone; trie stackdepot RFC (debated: complexity/overhead)

Rust + KASAN: 2026 Status & Open Questions

All three KASAN modes now wired up for Rust, but test coverage remains minimal

2026 changes, architecture constraints, and blind spots:

- *2026 toolchain & mode updates (rustc >= 1.96.0 + Clang)*
 - SW_TAGS support (CONFIG_KASAN_SW_TAGS) added via -Zsanitizer=kernel-hwaddress
 - Enforced CC_IS_CLANG for KASAN+RUST; disabled on 32-bit ARM & LoongArch (!KASAN)
- *Why the base mechanism works (shared LLVM passes & C slab)*
 - Clang and rustc run identical LLVM ASan/HWASan passes; core crate built from source
 - KBox / KVec / Arc allocate via C kcalloc/kfree; HW_TAGS (MTE) works without rustc flags
- *Confidence gaps & missing coverage to address*
 - Only 1 KUnit test (rust_uaf on KVec) - no tests for stack, globals, or memintrinsics
 - No #[sanitize(address = "off")] yet; Rust excluded from Clang LTO (RUST_INLINE_HELPERS)

KFENCE: 2026 Updates & Ongoing Discussions

Production fault controls, SLUB/memcg cleanups, and lockless freeing

Merged updates & active LKML / RFC threads:

- *Production fault controls & Arm64 MTE coexistence (merged)*
 - New boot parameter: `kfence.fault={report,oops,panic}` for custom fault handling
 - Auto-disables KFENCE when KASAN HW tags (MTE) are enabled at boot or runtime
- *SLUB cleanups, Context Analysis & lifecycle hardening (merged)*
 - Removed `slabobj_ext` metadata & redundant SLUB post-alloc zeroing for KFENCE
 - Enabled `CONTEXT_ANALYSIS`; randomized init freelist; disabled on shutdown
- *Ongoing: lockless `kfree_nolock()` & hibernation page mapping*
 - Supporting KFENCE objects in `kfree_nolock()` ([PATCH RFC]) via `irq_work` & `llist`
 - Explicit hibernation page mapping ([PATCH v2]) to keep pool pages mapped

KMSAN & KASAN on Interrupt / Entry Paths

Eliminating KMSAN false positives on IRQ/exception entry & checking noinstr

Problem, kernel + LLVM design, and KASAN bonus:

- *Root cause: IRQ/exception context clobbering & uninstrumented noinstr*
 - Interrupts clobber interrupted task's parameter metadata; noinstr skips setup
 - Per-IRQ-level & exception KMSAN contexts with refresh at IRQ transitions
- *Sanitizer-neutral region markers & Clang/LLVM support*
 - instrumentation_begin()/end() call sanitizer_instrumentation_{begin,end}()
 - 3 stacked LLVM PRs (common region analysis, KMSAN, ASan/KASAN)

KMSAN: 2026 Updates & Ongoing Discussions

Page allocator corner cases, DMA & virtual mappings, and arch fixes

Merged updates & active LKML / RFC threads:

- *Page poisoning, DMA handling & early-boot fixes (merged)*
 - Fixed poisoning of high-order non-compound pages and added KUnit tests
 - Fixed kmsan_handle_dma() phys-to-page conversion & early-boot kmalloc hook
- *FORTIFY_SOURCE memset() & virtual mapping metadata*
 - Re-enabled FORTIFY_SOURCE memset() bounds checks under CONFIG_KMSAN (merged)
 - Fixing ioremap error cleanup & iounmap metadata teardown ([PATCH 1/2 & 2/2])
- *s390 architecture & instrumentation fixes (merged)*
 - Moved s390 arch_local_irq_*() & rseq helpers out-of-line to cut build/inline bloat
 - Unpoisoned s390 CPACF crypto buffers & fixed return_address KMSAN checks

KCSAN: 2026 Updates & Ongoing Discussions

NMI safety, compiler diagnostic fixes, and fleet report deduplication

Merged updates & active LKML / RFC threads:

- *Core runtime & compiler diagnostic refinements (merged)*
 - Avoids unintended watchpoint access checking inside NMIs
 - Silenced -Wmaybe-uninitialized on `__kcsan_check_access()`; enabled `CONTEXT_ANALYSIS`
- *Architecture, assertion & runtime parameter fixes*
 - Disabled KCSAN in arm64 delay.o (no recursive delays); fixed `ASSERT_EXCLUSIVE_BITS`
 - Fixed div-by-zero when zeroing `udelay_* / skip_watch` sysfs params ([PATCH])
- *LKML discussion: Fleet report deduplication & dynamic filtering*
 - Deduplicating reports beyond `CONFIG_KCSAN_REPORT_ONCE_IN_MS` without allocs
 - Using debugfs whitelist/blacklist filters from user-space runners (e.g. syzbot)

KCOV: 2026 Updates & Ongoing Discussions

PREEMPT_RT support, coverage filtering, memory tracing & delay injection

Merged updates & active LKML / RFC threads:

- *Simultaneous modes & PREEMPT_RT support (merged)*
 - Simultaneous KCOV_ENABLE (local) and KCOV_REMOTE_ENABLE on the same task
 - Fixed PREEMPT_RT races (saved state in task_struct); Rust & context analysis
- *Suppressing timer & scheduler coverage leaks ([PATCH v4])*
 - Nestable KCOV_PAUSED & kcov_pause guard in scheduler and hrtimer paths
 - Stops instrumented callees from polluting traces (+42-54% RT syzkaller corpus)
- *Extended traces, Memory Tracing & Delay Injection ([RFC v3])*
 - KCOV_TRACE_PC_EXT (LLVM 23+): entry/exit records reconstruct call stacks
 - KCOV_MEMORY + KCOV_SET_DI: traces memory via KASAN hooks & injects delays