Kernel Sanitizers Office Hours

Hosted by: *Alexander Potapenko (Google); Dmitry Vyukov (Google); Kees Cook (Google); Marco Elver (Google); Paul McKenney (Meta)*

Agenda

1. **Kernel Sanitizers Primer**
   - Kernel Address Sanitizer (KASAN)
   - Kernel Memory Sanitizer (KMSAN)
   - Kernel Concurrency Sanitizer (KCSAN)
   - Undefined Behaviour Sanitizer (UBSAN)
2. Discussion and Questions

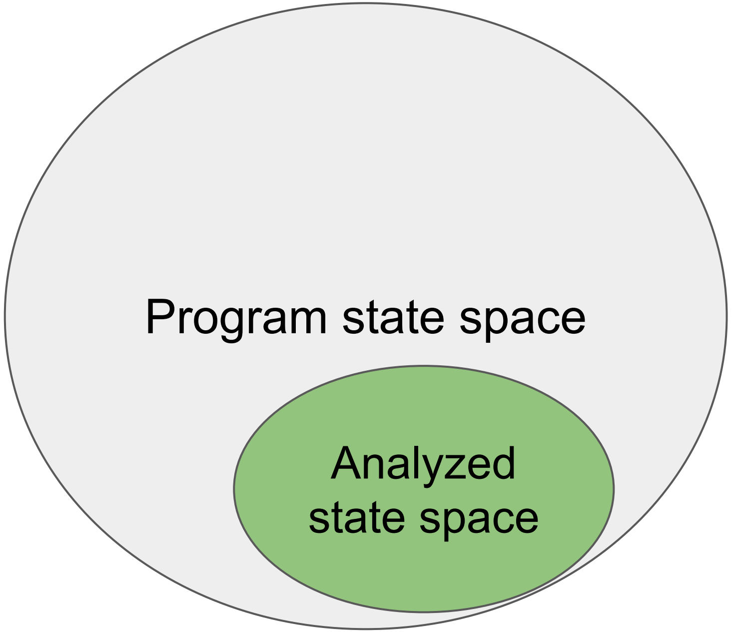
Kernel Sanitizers Primer

Dynamic Analysis

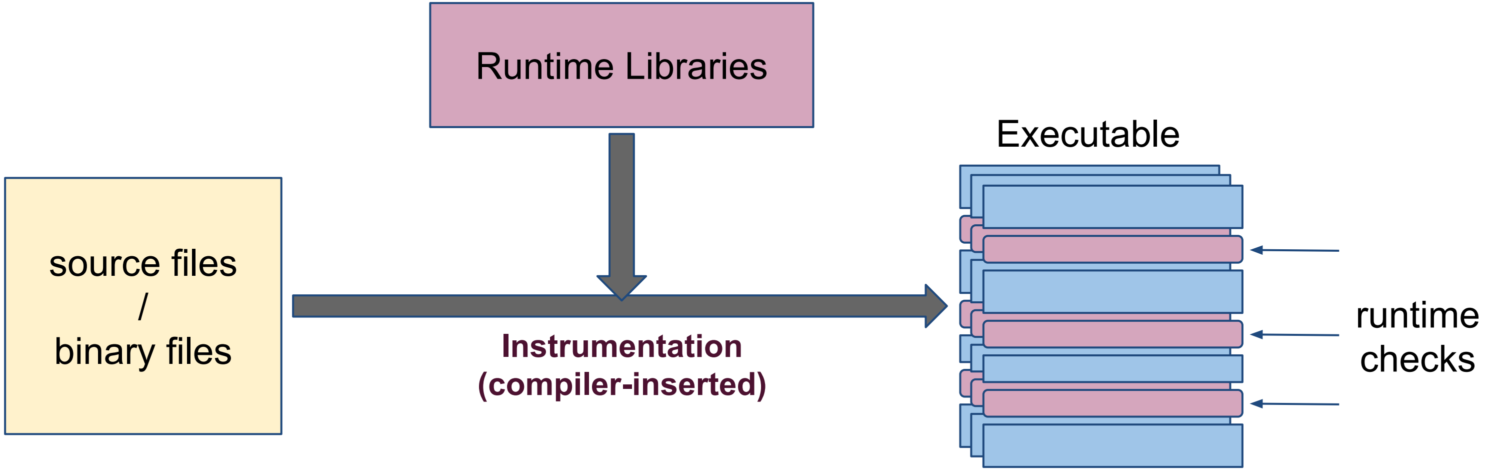
- *Dynamic program analysis* is about analyzing a piece of code "dynamically": the analysis observes the program as it is being executed
- Dynamic analysis reports typically point out *system errors* or *failures*
  - Can rarely deduce the underlying *system fault / bug*
  - Quality of diagnostics often inversely correlated with the performance of a tool

Dynamic Analysis

- Only the state space that was *covered* during execution is analyzed



Dynamic Analysis



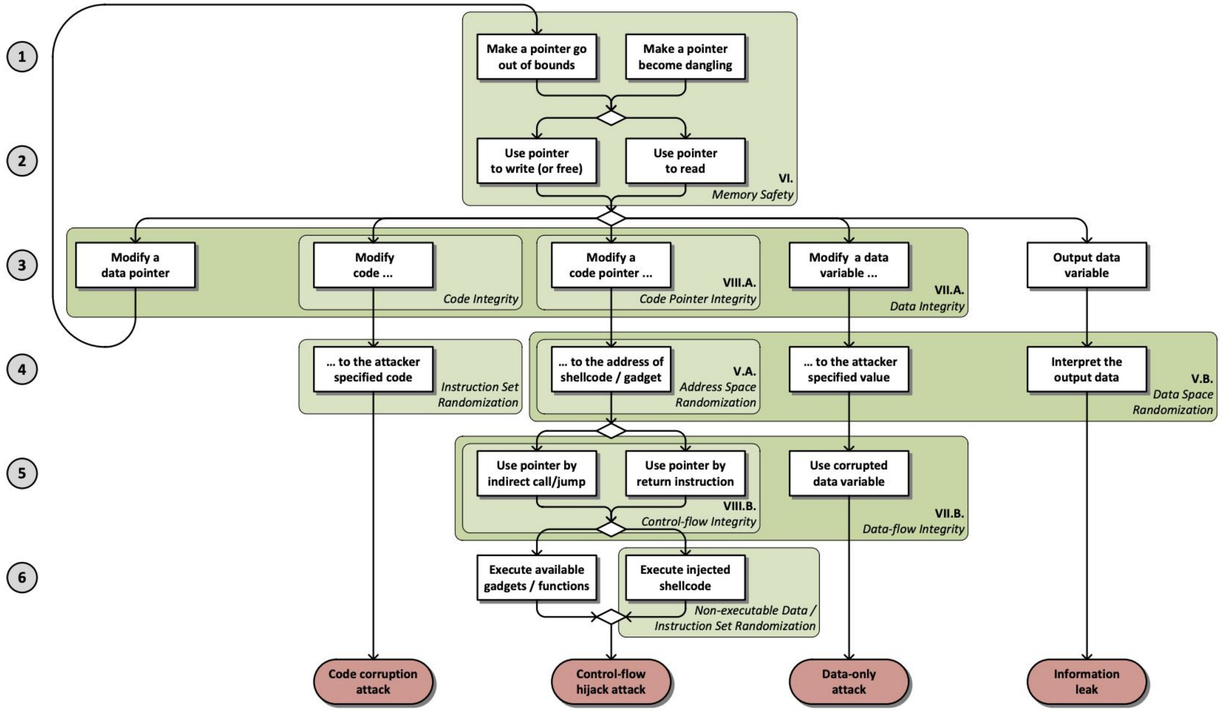
Undefined Behavior

Why “undefined behavior”?

- C designed for fine-grained control over low-level details, such as how memory is organized (essential in kernel development)
- *Unsafe languages* simply say: *some well-typed programs are* ***undefined*** 💥
  - Trade-off: simpler type system + higher performance (no dynamic error checking)
- *Safe languages* with manual memory management hard to design & implement
  - Rust is considered safe in its “safe” subset

Memory Safety Errors

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!

```c
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!

```c
const char *strtoupper_buggy(const char *str)
{
    char buf[64];
    strlcpy(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!
- *Hardware tags: CONFIG_KASAN=y + CONFIG_KASAN_HW_TAGS=y*
  - Currently requires *Arm64 Memory Tagging Extension (MTE)*
  - Usable in production kernels!

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!

```c
void hello_tux_buggy(const char *name)
{
    char buf[10];
    strlcpy(buf, str, sizeof(buf));
    if (buf[0] == 't' && buf[1] == 'u' && buf[2] == 'x')
        printf("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` (-trivial-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 6.11 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 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, arithmetic wrap-around: 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
110 fixes in 5 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 num_bars;
    struct bar[] __counted_by(num_bars);
} *p = kmalloc(struct_size(p, bar, 8), GFP_KERNEL);
...
do_something(p->array[index]); // catch index outside of [0..(p->num_bars-1)]
```

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/log/?qt=grep&q=shift-out-of-bounds
93 fixes in 5 years
Depends on the kernel's default use of `-fstrict-flex-arrays=3` and the hundreds of refactoring patches to move from old `array[1]`/`array[0]` style "fake" flexible arrays to real flexible arrays, and related changes.

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

- Technically working …
  - GCC & Clang: `-fsanitize={signed-integer-overflow,pointer-overflow}`
  - Clang: has; GCC: **Needed**: `-fsanitize=unsigned-integer-overflow`
- … but there are some significant behavioral caveats related to `-fwrapv` and `-fwrapv-pointer` (enabled via kernel's use of `-fno-strict-overflow`)
  - "It's not an undefined behavior to wrap around."
  - Clang: 19+; GCC: **Needed**
- 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**
- Type filtering support allows instrumentation to be toggled for specific types
  - Clang: 20?; GCC: **Needed**
- Add annotations in kernel for *unexpected* wrap-around types (`size_t` first)
  - Clang: 20?; 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	Thread 1

`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);`
	`/* 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

`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);`
	`/* update_foo_lock does not need to be held! */`
	`... = READ_ONCE(shared_foo);`
		`/* Bug! */`
		`WRITE_ONCE(shared_foo, 42);`

Concurrency bugs that are not data races

Thread 0	Thread 1	Thread 2

`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);`
	`/* update_foo_lock does not`
	`need to be held! */`
	`... = READ_ONCE(shared_foo);`
		~~`/* Bug! */`~~
		~~`WRITE_ONCE(shared_foo, 42);`~~

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(`*var*`)` `ASSERT_EXCLUSIVE_WRITER_SCOPED(`*var*`)`	✘	✔
`ASSERT_EXCLUSIVE_ACCESS(`*var*`)` `ASSERT_EXCLUSIVE_ACCESS_SCOPED(`*var*`)`	✘	✘
`ASSERT_EXCLUSIVE_BITS(`*var*, *mask*`)`	`~mask` ✔ `mask` ✘	✔

Agenda

1. Kernel Sanitizers Primer
    - Kernel Address Sanitizer (KASAN)
    - Kernel Memory Sanitizer (KMSAN)
    - Kernel Concurrency Sanitizer (KCSAN)
    - Undefined Behaviour Sanitizer (UBSAN)
2. **Discussion and Questions**

Discussion and Questions

- Share your experience. Have sanitizers been helpful, not so helpful?
- Rust and kernel sanitizers?
- Fixing data races?
- …

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**

📖 "Who's afraid of a big bad optimizing compiler?", LWN 2019. URL: https://lwn.net/Articles/793253/

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