Sanitizing the Linux kernel

On KASAN and other Dynamic Bug-finding Tools

Andrey Konovalov, xairy.io

Linux Security Summit Europe
September 16th 2022

Fuzzing is awesome?

Fuzzing is useless

Fuzzing is useless⋆

⋆without dynamic bug detectors

Sanitizing the Linux kernel

On KASAN and other Dynamic Bug-finding Tools

Andrey Konovalov, xairy.io

Linux Security Summit Europe
September 16th 2022

Agenda

1. What are Sanitizers
2. KASAN and its Generic mode
3. More Sanitizers: KMSAN, KCSAN, and UBSAN
4. In-field Sanitizers: KFENCE and SW_TAGS KASAN
5. Sanitizers as a mitigation: HW_TAGS KASAN
6. Extending Sanitizers on the example of KASAN and KMSAN
7. Notes on implementing custom bug detectors

Sanitizers

- A family of bug detectors
- Initially implemented for userspace (ASan, MSan, etc.)
- Later ported to the Linux kernel (KASAN, KMSAN, etc.)

- Features:
    - Easy to use
    - Fast
    - Precise: no false positives
    - Detailed reports

Generic KASAN

KASAN — Kernel Address Sanitizer

- Dynamic memory corruption detector for the Linux kernel
- Finds out-of-bounds, use-after-free, and double/invalid-free bugs
- Supports slab, page_alloc, vmalloc, stack, and global memory
- Requires compiler support: implemented in both Clang and GCC
- Has 3 modes, we'll focus on Generic mode (`CONFIG_KASAN_GENERIC`)
- google.github.io/kernel-sanitizers/KASAN

KASAN parts

- Compiler module (llvm/lib/Transforms/Instrumentation/AddressSanitizer.cpp)

- Runtime part (mm/kasan/ + include/linux/kasan.h + ...)

KASAN parts

- Compiler module (llvm/lib/Transforms/Instrumentation/AddressSanitizer.cpp)
  - Instruments memory accesses when building the kernel
  - Inserts redzones for stack and global variables

- Runtime part (mm/kasan/ + include/linux/kasan.h + ...)

KASAN parts

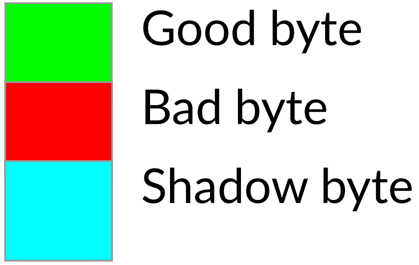
- Compiler module (llvm/lib/Transforms/Instrumentation/AddressSanitizer.cpp)
    - Instruments memory accesses when building the kernel
    - Inserts redzones for stack and global variables

- Runtime part (mm/kasan/ + include/linux/kasan.h + ...)
    - Maintains shadow memory to track memory state
    - Hooks into kernel allocators to track alloc/free events
    - Prints bug reports

Shadow memory

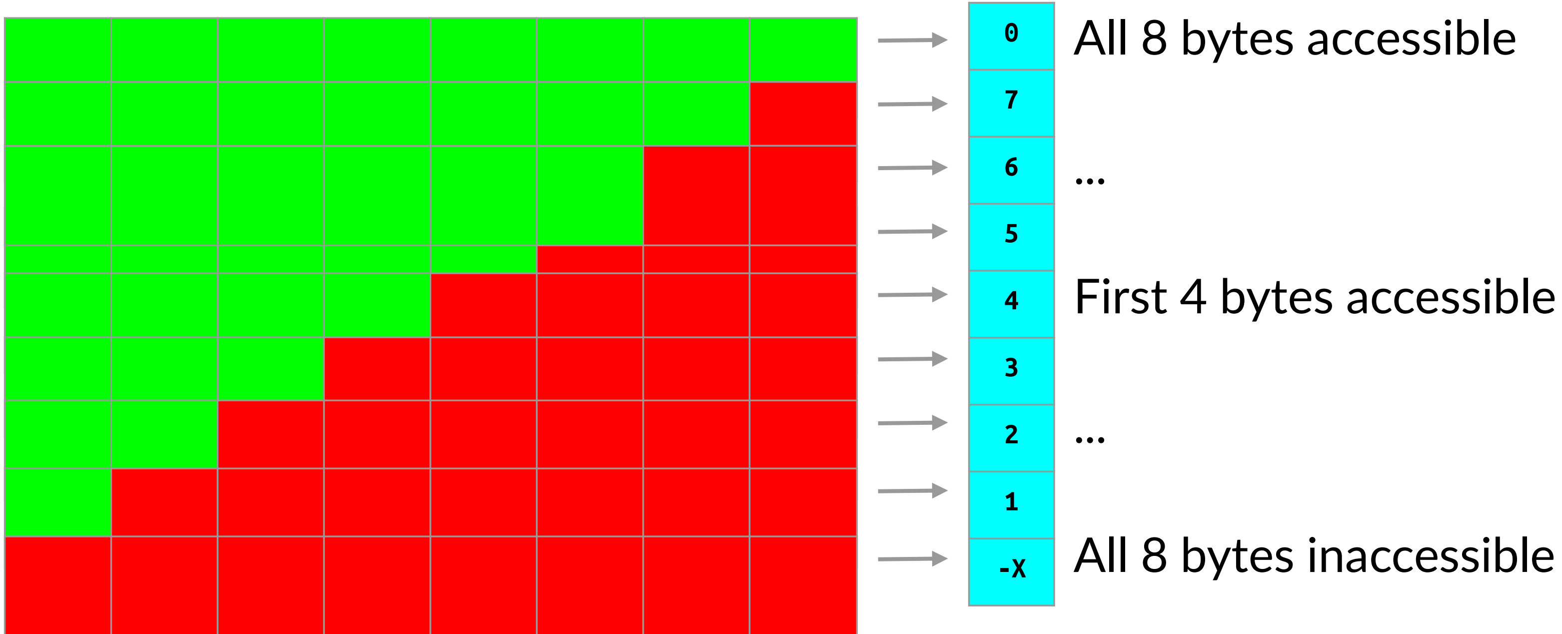
- Compiler module
  - Instruments memory accesses when building the kernel
  - Inserts redzones for stack and global variables
- Runtime part
  - Maintains **shadow memory** to track memory state
  - Hooks into kernel allocators to track alloc/free events
  - Prints bug reports

Shadow byte



Any 8 aligned bytes usually have only 9 states:

N good bytes and 8 - N bad bytes ($0 \le N \le 8$)



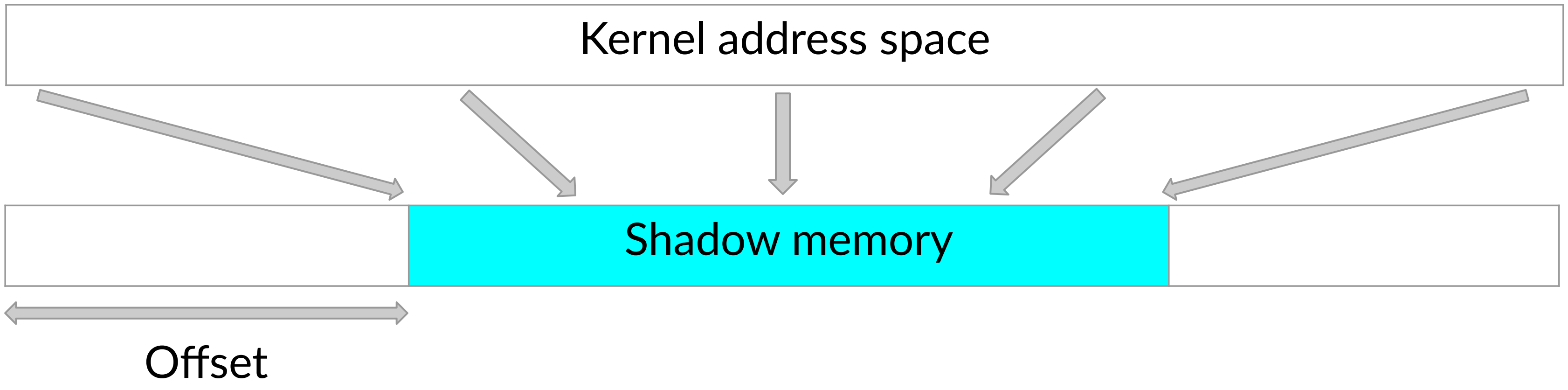
Shadow byte values for inaccessible memory

```
#define KASAN_PAGE_FREE         0xFF  /* freed page */
#define KASAN_PAGE_REDZONE      0xFE  /* redzone for kmalloc_large allocation */
#define KASAN_SLAB_REDZONE      0xFC  /* redzone for slab object */
#define KASAN_SLAB_FREE         0xFB  /* freed slab object */
#define KASAN_VMALLOC_INVALID   0xF8  /* inaccessible space in vmap area */
#define KASAN_SLAB_FREETRACK    0xFA  /* freed slab object with free track */
#define KASAN_GLOBAL_REDZONE    0xF9  /* redzone for global variable */
#define KASAN_STACK_LEFT        0xF1
#define KASAN_STACK_MID         0xF2
#define KASAN_STACK_RIGHT       0xF3
#define KASAN_STACK_PARTIAL     0xF4
```

Shadow memory region

- Contains shadow bytes for each mapped region of kernel memory
- Memory-to-shadow mapping scheme:

```
Shadow = (Addr >> 3) + Offset
```



x86-64 kernel memory layout (4-level page tables)

```
...
ffff800000000000 | ffff87ffffffffff |   8 TB | ... guard hole, also reserved for hpv.
ffff880000000000 | ffff887fffffffff | 0.5 TB | LDT remap for PTI
ffff888000000000 | ffffc87fffffffff |  64 TB | mapping of phys. memory (page_offset_base)
ffffc88000000000 | ffffc8ffffffffff | 0.5 TB | ... unused hole
ffffc90000000000 | ffffe8ffffffffff |  32 TB | vmalloc/ioremap space (vmalloc_base)
ffffe90000000000 | ffffe9ffffffffff |   1 TB | ... unused hole
ffffea0000000000 | ffffeaffffffffff |   1 TB | virtual memory map (vmemmap_base)
ffffeb0000000000 | ffffebffffffffff |   1 TB | ... unused hole
ffffec0000000000 | fffffbffffffffff |  16 TB | KASAN shadow memory
...
```

Shadow mapping routines for x86-64

- During early boot, shadow mapped to a zero page
- Once page tables are initialized, proper shadow is mapped
- After boot, shadow gets mapped for vmalloc/vmap areas

Instrumentation of memory accesses

- Compiler module
    - **Instruments memory accesses** when building the kernel
    - Inserts redzones for stack and global variables

- Runtime part
    - Maintains shadow memory to track memory state
    - Hooks into kernel allocators to track alloc/free events
    - Prints bug reports

Compiler instrumentation: 8-byte access

```
*a = ...;
```

```
char *shadow = (a >> 3) + Offset;
if (*shadow)
   kasan_report(a);
*a = ...;
```

Instrumentation: N-byte access (N = 1, 2, 4)

```
*a = ...;
```

```
char *shadow = (a >> 3) + Offset;
if (*shadow && *shadow < (a & 7) + N)
    kasan_report(a);
*a = ...;
```

Allocators hooks

- Compiler module
  - Instruments memory accesses when building the kernel
  - Inserts redzones for stack and global variables

- Runtime part
  - Maintains shadow memory to track memory state
  - **Hooks into kernel allocators** to track alloc/free events
  - Prints bug reports

Allocators hooks

- KASAN need to keep shadow up-to-date
- This requires tracking of alloc/free events

- KASAN adds hooks to kernel allocators
  - SLUB/SLAB, page_alloc, vmalloc (grep code for "`kasan_`")

Slab layout (for SLUB)

- Slab is fully poisoned (marked as inaccessible) when allocated
- Objects in slab are unpoisoned when allocated
- Always-poisoned redzones between objects (⇒ fewer objects fit in slab)

Slab layout without KASAN:

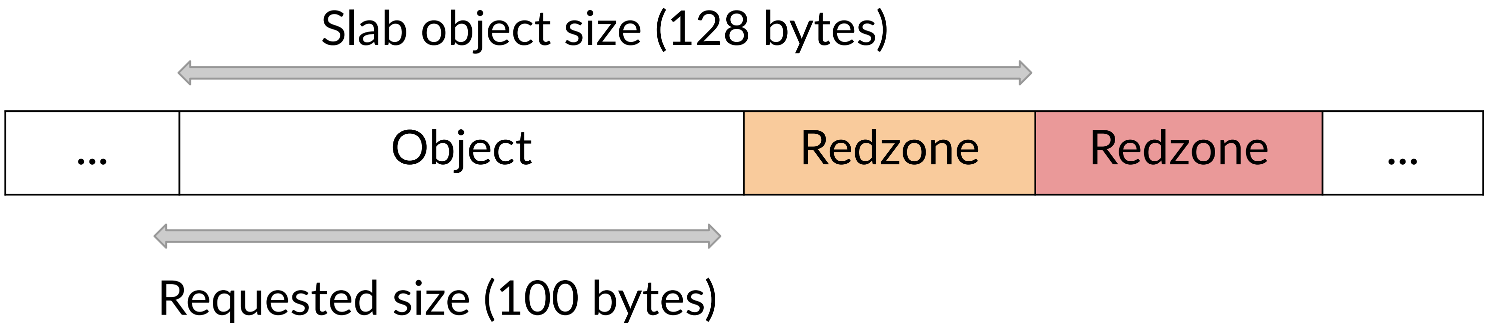
Free object	Allocated object	...	Allocated object

Slab layout with KASAN:

Free	Redzone	Allocated	Redzone	...	Allocated	Redzone

Additional redzones for kmalloc'ed objects

- kmalloc chooses the best fitting cache to serve objects
  - An allocation of 100 bytes is served from kmalloc-128
- KASAN poisons the unused tail (but [unpoisons]() if ksize is called)



Quarantine for freed memory

- When memory is freed, it's typically immediately reallocated
  - ⇒ Detecting use-after-free is hard
- KASAN implements quarantine for slab objects
  - Freed objects are not returned to allocator immediately
  - Instead, they are put into a delayed reuse queue
  - ⇒ Higher chance to detect use-after-free

Redzones for stack and globals

- Compiler module
  - Instruments memory accesses when building the kernel
  - Inserts **redzones for stack and global variables**

- Runtime part
  - Maintains shadow memory to track memory state
  - Hooks into kernel allocators to track alloc/free events
  - Prints bug reports

Compiler instrumentation: stack variables [1/3]

```
void foo() {

  char x[10];


  <------------- Original function code ------------->

}
```

Compiler instrumentation: stack variables [2/3]

```
void foo() {
  char rz1[32];
  char x[10];
  char rz2[22];

  <------------- Original function code ------------->

}
```

Compiler instrumentation: stack variables [3/3]

```
void foo() {
  char rz1[32];
  char x[10];
  char rz2[22];
  // Compiler-inserted code to unpoison x and poison redzones.
  <------------- Original function code ------------->
  // Compiler-inserted code to unpoison stack frame.
}
```

Compiler instrumentation: global variables

```
char x[10];
```

```
struct {
   char original[10];
   char redzone[54]; // Poisoned via constructors
} x;
```

Reports for found bugs

- Compiler module
    - Instruments memory accesses when building the kernel
    - Inserts redzones for stack and global variables

- Runtime part
    - Maintains shadow memory to track memory state
    - Hooks into kernel allocators to track alloc/free events
    - **Prints bug reports**

Reporting bugs

- When KASAN detects a bug, it prints a bug report
- Reports should be detailed to be useful

- Among other things, KASAN keeps and prints alloc and free stack traces
  - For slab (and page_alloc via page owner)
  - Stack traces saved in stack depot
  - Alloc stack trace handle is stored in redzone
  - Free stack trace handle is stored either in object or in redzone

Report example: slab out-of-bounds [1/5]

```
void kmalloc_oob_right(void)
{
    char *ptr;
    size_t size = 115;

    ptr = kmalloc(size, GFP_KERNEL);
    ptr[size] = 'x';
}
```

Report example: slab out-of-bounds [2/5]

```
BUG: KASAN: slab-out-of-bounds in kmalloc_oob_right+0x408/0x454
Write of size 1 at addr ffff00000d039a73 by task kunit_try_catch/96

CPU: 0 PID: 96 Comm: kunit_try_catch Tainted: G                N 6.0.0-rc3
#32
Hardware name: linux,dummy-virt (DT)
Call trace:
 ...
 __asan_report_store1_noabort+0x2c/0x38
 kmalloc_oob_right+0x408/0x454
```

Report example: slab out-of-bounds [3/5]

```
Allocated by task 96:
 kasan_set_track+0x3c/0x6c
 kasan_save_alloc_info+0x24/0x30
 __kasan_kmalloc+0x90/0xa8
 kmem_cache_alloc_trace+0x1e8/0x340
 kmalloc_oob_right+0x100/0x454
 kunit_try_run_case+0xec/0x2a4
 kunit_generic_run_threadfn_adapter+0x54/0x80
 kthread+0x220/0x2f4
 ret_from_fork+0x10/0x20
```

Report example: slab out-of-bounds [4/5]

```
The buggy address belongs to the object at ffff00000d039a00
 which belongs to the cache kmalloc-128 of size 128
The buggy address is located 115 bytes inside of
 128-byte region [ffff00000d039a00, ffff00000d039a80)

The buggy address belongs to the physical page:
page:(____ptrval____) refcount:1 mapcount:0 mapping:0000000000000000 index:0x0
pfn:0x4d039
flags: 0xffff00000000200(slab|node=0|zone=0|lastcpupid=0xffff)
raw: 0ffff00000000200 0000000000000000 dead000000000122 ffff000006c01300
raw: 0000000000000000 0000000080100010 00000001ffffffff 0000000000000000
page dumped because: kasan: bad access detected
```

Report example: slab out-of-bounds [5/5]

```
Memory state around the buggy address:
 ffff00000d039900: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 ffff00000d039980: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
>ffff00000d039a00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 fc
                                                                ^
 ffff00000d039a80: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
 ffff00000d039b00: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
```

Other assorted parts

- Compiler module
  - Instruments memory accesses when building the kernel
  - Inserts redzones for stack and global variables

- Runtime part
  - Maintains shadow memory to track memory state
  - Hooks into kernel allocators to track alloc/free events
  - Prints bug reports

Generic KASAN notes and parts

- Some kernel parts are not instrumented by compiler
  - Assembly; early boot code, allocators (grep for `KASAN_SANITIZE`)
- `CONFIG_KASAN_OUTLINE`: outline instrumentation mode
  - Compiler adds function calls; slower, but image is smaller

KASAN notes and parts

- KASAN has 3 modes
  - Generic: just covered, supported by many architectures
  - Software Tag-Based: arm64-only, software Memory Tagging
  - Hardware Tag-Based: arm64-only, Arm Memory Tagging Extension
- Test suite
  - KUnit-compatible tests
  - KUnit-incompatible tests
- Bugs and features are tracked in Bugzilla
  - Some modes are missing certain features

Generic KASAN summary

- Dynamic memory corruption detector for the Linux kernel
- Finds out-of-bounds, use-after-free, and double/invalid-free bugs
- Supports slab, page_alloc, vmalloc, stack, and global memory
- Requires compiler support: implemented in both Clang and GCC
- google.github.io/kernel-sanitizers/KASAN

- Relatively fast: ~x2 slowdown
- RAM impact: shadow (1/8 RAM) + quarantine (1/32 RAM) + ~x1.5 for slab
- Basic usage: enable and run tests or fuzzer

More Sanitizers

More Sanitizers

- KASAN takes care of out-of-bounds and use-after-free bugs
- What about the rest?

KMSAN — Kernel Memory Sanitizer

- Detects uses of uninitialized memory
  - Including information leaks to userspace or to hardware
- Uses compiler instrumentation and shadow memory
  - Shadow memory describes which memory is initialized
- On the way into mainline
- google.github.io/kernel-sanitizers/KMSAN

KCSAN — Kernel Concurrency Sanitizer

- Detects data races
- Uses compiler instrumentation and "soft" watchpoints
  - Stalls on access watchpoints and checks if memory value changed
- google.github.io/kernel-sanitizers/KCSAN

- Also see KTSAN
  - Failed attempt to implement a Happens-Before data race detector

UBSAN — [Kernel] Undefined Behavior Sanitizer

- Detects certain types of undefined behaviour
- Uses compiler instrumentation
- No `K` in its `CONFIG_` name
- kernel.org/doc/html/latest/dev-tools/ubsan.html

- [After-talk update:] Some UBSAN checks can be enabled in production

In-field Sanitizers

In-field Sanitizers

- So far, mentioned Sanitizers were intended for testing and fuzzing

- What about running Sanitizers with real workloads?
  - In dogfood (internal beta testing with real-world usage)
  - In production

KFENCE — Kernel Electric-Fence

- Detects out-of-bounds, use-after-free, and double/invalid-free for slab
- Places allocated object next to a protected guard page
- Uses sampling to make the overhead unnoticeable
    - ⇒ Probabilistics, need to be deployed across fleet of machines
- Can be used in production or in dogfood
- Not a "Sanitizer" by name but is one by spirit
- google.github.io/kernel-sanitizers/KFENCE

Software Tag-Based KASAN

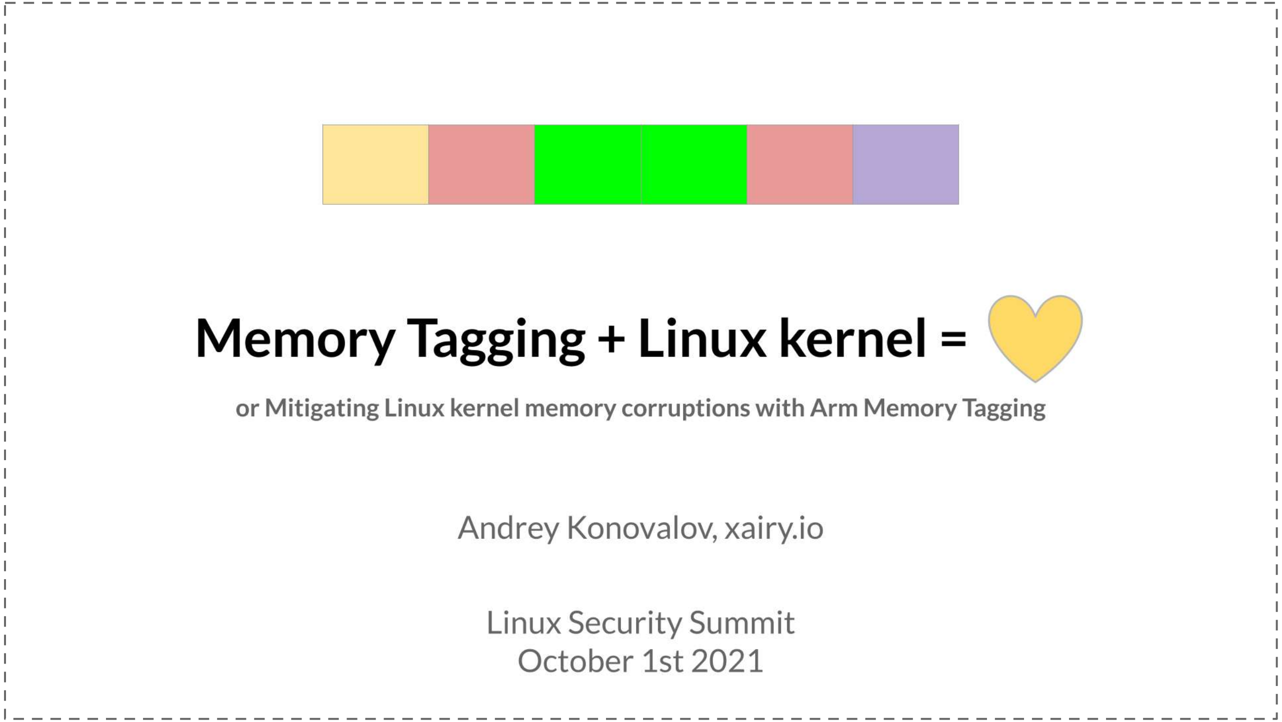
- KASAN mode based on software Memory Tagging approach
  - Tag checks are inserted via compiler instrumentation
- Similar performance impact to Generic: ~x2
- Less RAM impact than Generic: less shadow, no quarantine
  - Recommended for arm64 devices with limited RAM, e.g. Android
- Bigger performance/RAM impact than KFENCE
  - OK for dogfood
  - Prepares for Arm Memory Tagging Extension
- Memory Tagging for the kernel: Tag-Based KASAN [video]

Sanitizers as a mitigation

Hardware Tag-Based KASAN

- KASAN mode based on Arm Memory Tagging Extension (MTE)
  - Tag checks performed by CPU
- Either a production mitigation or a fast in-field bug detector
- RAM impact: ~3% for storing memory tags
- Performance impact: still unknown, no CPUs released yet
  - <10% expected in Sync mode
  - ~0% expected in Async mode

LSS 2021 talk on Hardware Tag-Based KASAN



xairy.io/talks/:

- Slides
- Video

Production stack traces for KASAN

- Having proper HW_TAGS KASAN reports from production requires:
    - Fast stack trace collection
    - Memory-bounded stack trace storage

- Plan:
    - Use Shadow Call Stack for collecting traces (arm64 maintainers resist)
    - Store stack trace handles in stack ring instead of redzones (in mm)
    - Memory-bounded mode for stack depot
    - Potentially, use sampling to limit the performance impact

Extending Sanitizers

Extending Sanitizers

- Basic usage: just enable
- Advanced usage: extend

- Sanitizers provide frameworks
  - KASAN — for marking memory accessible/inaccessible
  - KMSAN — for checking whether memory is initialized

Extending KASAN: custom redzones

- `sk_buff.head` data buffer allocated as a single chunk
    - Has `skb_shared_info` placed at the end
- Overflowing data buffer into `skb_shared_info` is not caught by KASAN
    - So called intra-object-overflow
    - I targeted this skb behavior in my CVE-2017-1000112 exploit
- Possible KASAN extension:
    - Add a redzone between data buffer and `skb_shared_info`
    - Related issue tracked in KASAN: poison skb linear data tail

Extending KASAN: add support for other allocators

- Example: percpu allocator is not supported by KASAN

Percpu allocator: fairly used

block/bdev.c, line 500
block/bio.c, line 1716
block/blk-iocost.c, line 2845
block/blk-mq.c, line 3840
crypto/cryptd.c, line 105
fs/aio.c, line 782
fs/ext4/mballoc.c, line 3484
fs/gfs2/ops_fstype.c, line 82
fs/namespace.c, line 214
fs/nfs/iostat.h, line 68
fs/squashfs/decompressor_multi_percpu.c, line 35
fs/xfs/xfs_super.c, 3 times
include/scsi/libfc.h, line 836
kernel/bpf/bpf_lru_list.c, 2 times
kernel/bpf/devmap.c, line 1075
kernel/bpf/percpu_freelist.c, line 10

net/caif/caif_dev.c, line 99
net/caif/cffrml.c, line 40
net/can/raw.c, line 352
net/core/dev.c, line 10594
net/core/gro_cells.c, line 73
net/core/neighbour.c, line 1755
net/core/page_pool.c, line 184
net/core/sock.c, line 3668
net/dccp/proto.c, line 1081
net/ipv4/af_inet.c, 6 times
net/ipv4/fib_trie.c, line 2427
net/ipv6/af_inet6.c, 4 times
net/ipv6/seg6_hmac.c, 2 times
net/mpls/af_mpls.c, line 1459
net/mptcp/mib.c, line 73
net/netfilter/ipvs/ip_vs_ctl.c, 3 times

net/netfilter/nf_conntrack_core.c, line 2791
net/netfilter/nf_synproxy_core.c, line 348
net/netfilter/nft_set_pipapo.c, 4 times
net/openvswitch/actions.c, 2 times
net/packet/af_packet.c, line 1230
net/sched/act_api.c, line 733
net/sched/cls_basic.c, line 213
net/sched/cls_matchall.c, line 221
net/sched/cls_u32.c, line 1054
net/sched/sch_generic.c, line 957
net/sctp/protocol.c, line 1227
net/smc/smc_stats.c, line 26
net/tipc/crypto.c, line 535
net/tls/tls_main.c, line 1009
net/xfrm/xfrm_ipcomp.c, 2 times
net/xfrm/xfrm_policy.c, line 3994

Percpu allocator: no sanitization (out-of-bounds)

```
void percpu_oob(struct kunit *test) {
    char __percpu *ptr = __alloc_percpu(128, 8);
    char *c_ptr = this_cpu_ptr(ptr);
    KUNIT_EXPECT_KASAN_FAIL(test, c_ptr[128] = 0x42);
    free_percpu(ptr);
}
```

```
# percpu_oob: EXPECTATION FAILED at mm/kasan/kasan_test.c:1386
KASAN failure expected in "c_ptr[128] = 0x42", but none occurred
not ok 57 - percpu_oob
```

Percpu allocator: no sanitization (use-after-free)

```
void percpu_uaf(struct kunit *test) {
    char __percpu *ptr = __alloc_percpu(128, 8);
    char *c_ptr = this_cpu_ptr(ptr);
    free_percpu(ptr);
    KUNIT_EXPECT_KASAN_FAIL(test, c_ptr[0] = 0x42);
}
```

```
# percpu_uaf: EXPECTATION FAILED at mm/kasan/kasan_test.c:1398
KASAN failure expected in "c_ptr[0] = 0x42", but none occurred
not ok 58 - percpu_uaf
```

Percpu allocator: need KASAN annotations

- Tracked in [KASAN: sanitize per-cpu allocations]
- Annotations will allow detecting more bugs

- Similar extensions also applicable to other allocators
  - Including custom allocators in drivers
  - Android allocators?

Extending KMSAN: USB annotations

- KMSAN was extended to detect information leaks over USB
- As a result, syzbot found bugs:
  - KMSAN: kernel-usb-infoleak in hif_usb_send
  - KMSAN: kernel-usb-infoleak in usbnet_write_cmd
  - KMSAN: kernel-usb-infoleak in hid_submit_ctrl
  - KMSAN: kernel-usb-infoleak in pcan_usb_wait_rsp
  - KMSAN: kernel-usb-infoleak in ttusb_dec_send_command
  - KMSAN: kernel-usb-infoleak in pcan_usb_pro_send_req
  - KMSAN: kernel-usb-infoleak in pcan_usb_pro_init

Outro

Finding more bugs

1. Improving the fuzzer
2. Improving the bug detectors

Know Sanitizers

- Testing and fuzzing
  - KASAN — out-of-bounds, use-after-free, invalid/double-free
  - KMSAN — uses of uninitialized memory, information leaks
  - KCSAN — data races
  - UBSAN — undefined behavior
- Dogfood or production
  - KFENCE — slab memory corruptions, sampling-based
  - SW_TAGS KASAN — memory corruptions, arm64-only
- As a mitigation
  - HW_TAGS KASAN — memory corruptions, arm64-only, requires Arm MTE

Make better bug detectors

- Extend Sanitizers
  - Add KASAN redzones to detect intra-object-overflows
  - Add KASAN annotations to more allocators
  - Add KMSAN annotations to detect leaks across security boundaries

- Make your own bug detectors
  - For other types of bugs (type confusions?)
  - For logical bugs (missing TLB flushes?)
  - Take inspiration from Sanitizers (compiler instrumentation)

Thank you!

google.github.io/kernel-sanitizers

Andrey Konovalov, xairy.io