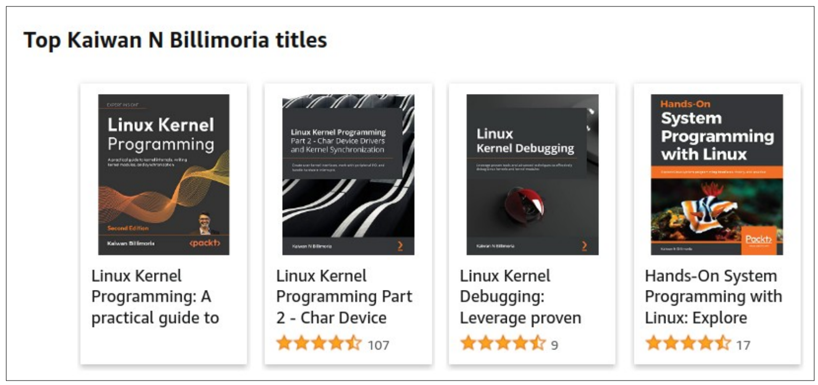
Using ASAN and KASAN and then Interpreting their shadow memory reports

Preliminaries

- whoami :-)
- All-in-one : https://bit.ly/m/kaiwan



  - LinkedIn public profile
  - My Amazon Author page

What I see as one of my key contributions to the community

  - Corporate Training on Linux
    - Brief
    - More detailed
  - My GitHub repos
  - My tech blog

- How many in the audience here use ___ kernel series in *production?*
  - 3.x
  - 4.x
  - 5.x
  - 6.x

Memory Defects (bugs)

- Incorrect memory accesses:
  - Using variables uninitialized, aka Uninitalized Memory Read (UMR) bugs
  - Out-Of-Bounds (OOB) memory accesses (read/write underflow/overflow bugs)
  - Use-After-Free (UAF) and Use-After-Return (UAR) (aka out-of-scope) bugs
  - Double-free bugs
- Memory leakage
- Data races
- (Internal) Fragmentation.

The Sanitizers !

Based on CTI - Compile Time Instrumentation - technology.

Available tooling for kernel memory debugging, in a nutshell:

Name	*Primary Purpose/Use*	*From kernel ver*	*Supported on*
KASAN (Kernel Address SANitizer)	Detects and reports kernel memory problems like uaf (use-after-free) and oob (out-of-bounds) bugs. Requires GCC >= 5.0 (usermode), for kernel requires GCC >= 8.3, any Clang supported by kernel, and SLUB. Some overhead: 1/8 kernel virtual address space reserved for shadowing; every memory access trapped into via compiler instrumentation (using inline code makes it much faster but larger binary image). Tech: **Compile Time Instrumentation (CTI)**; code compiled with `-fsanitize=kernel-address`	4.0	x86_64 from 4.0, ARM64 from 4.4, xtensa, s390 *v5.11 (Feb 15 2021): ARM-32 !*
UBSAN (Undefined Behavior SANitizer)	Catches several types of runtime UB. As with KASAN, it uses Compile Time Instrumentation (CTI) to do so. With UBSAN enabled fully, the kernel code is compiled with the `-fsanitize=undefined` option switch. Catches arithmetic-related UB and memory UB...	4.5	x86_64, ARM, ARM64; gcc 4.9 and gcc 5 onwards
kmemleak	Detect and reports memory leakage within kernel (*only* slab cache layer: kmalloc + friends, and vmalloc).	2.6.31	x86, arm, arm64, powerpc, sparc, sh, microblaze, ppc, mips, s390, metag, tile

SLUB debug	Detect and report slab cache – SLUB – memory errors (via the slub_debug= on kernel cmdline)	2.6.23	<all?>
KMSAN (Kernel Memory SANitizer) (*new*)	Detect UMR defects in the kernel; (similar to userspace MSan); not for production	6.1	Only x86_64. Requires Clang 14.0.6 +
~~Kmemcheck~~ *Removed in 4.15*	~~Detects and reports uninitialized kernel memory accesses. Similar to userland valgrind's memcheck functionality. Pretty major overhead – meant for debug kernel usage.~~	~~2.6.31~~	~~x86 and x86_64 only~~

KASAN Modes

1. Generic mode:
"... This mode consumes about 1/8th of available memory at kernel start and introduces an overhead of ~x1.5 for the rest of the allocations. The performance slowdown is ~x3."

2. Software tag-based; only on arm64… fast enough to be used in near-production

3. Hardware tag-based; "… in production as an in-field bug detector or a security mitigation. This mode is based on the Arm Memory Tagging Extension and is expected to have a very low performance overhead."
(2 and 3 can be and are leveraged by Android).

(Generic mode) KASAN is NOT for production..
Can use **KFENCE** instead; sampling-based approach; fast enough; must run for a long while...

CONFIG_STACKTRACE=y ; helps… 'for better error detection'.

(K)ASAN shadow memory

KASAN requires 1 byte of ‘shadow’ (virtual) memory to track 8 bytes of ‘real’ (virtual) memory
(or 1 bit to track 1 byte).

So, on x86_64, with a 128T:128T :: U:K VM split,
it uses a ‘shadow memory’ region of size 128 TB / 8 = 16 TB (virtual memory).

Use the **procmap** utility to literally ‘see’ the KASAN 16 TB shadow region.
(A request: please try out *procmap*; star it too! Partial screenshot on the right...).

User-space ASAN example   [...]

My HOSPL book GitHub repo’s *ch5/* folder with the membugs.c code is here.
So lets try it out with buggy code (*c’mon, spot the bug!*):

```
$ cat membugs.c
...
static void buggy1(void)
{
      char arr[5], tmp[8];

      memset(arr, 'a', 5);
      memset(tmp, 't', 8);
      tmp[7] = '\0';
```

```
    printf("arr = %s\n", arr);  /* Bug: read buffer overflow */
}
```
…
Build using gcc or clang (*prefer clang*);
Use the `-fsanitize=address` compiler option switch.

```
$ clang -g -O0 -Wall -Wextra -DDEBUG -fsanitize=address -fsanitize-address-use-after-scope membugs.c -o membugs_dbg_asan
```
...
(Can replace clang with gcc).

```
$ ASAN_OPTIONS=symbolize=1 ./membugs_dbg_asan 5
arr = 0x7fc572500020 tmp= 0x7fc572500040
=================================================================
==72362==ERROR: AddressSanitizer: stack-buffer-overflow on address
0x7fc572500025 at pc 0x000000442432 bp 0x7ffc6cdd3640 sp 0x7ffc6cdd2dc8
READ of size 6 at 0x7fc572500025 thread T0      << thread T0 => main thread >>
```

1

2

1. It’s a buffer overflow issue; in stack memory
2. It’s a read overflow of size 6 bytes (at address 0x7fc572500025 in the main() thread (T0)

```
    #0 0x442431 in printf_common(void*, char const*, __va_list_tag*)
asan_interceptors.cpp.o
    #1 0x443d6e in printf
(<...>/Hands-on-System-Programming-with-Linux/ch5/membugs_dbg_asan+0x443d6e)
(BuildId: b712d767295861216504d9a7452e2d6a08d25cab)
    #2 0x4f49ec in read_overflow_compilemem <...>/Hands-on-System-Programming-
with-Linux/ch5/membugs.c:241:2
    #3 0x4f509e in process_args
<...>/Hands-on-System-Programming-with-Linux/ch5/membugs.c:354:4
    #4 0x4f51c8 in main
<...>/Hands-on-System-Programming-with-Linux/ch5/membugs.c:398:2
    #5 0x7fc5743d6b89 in __libc_start_call_main (/lib64/libc.so.6+0x27b89)
(BuildId: 7026fe8c129a523e07856d7c96306663ceab6e24)
    #6 0x7fc5743d6c4a in __libc_start_main@GLIBC_2.2.5
(/lib64/libc.so.6+0x27c4a) (BuildId: 7026fe8c129a523e07856d7c96306663ceab6e24)
    #7 0x41d394 in _start
(<...>/Hands-on-System-Programming-with-Linux/ch5/membugs_dbg_asan+0x41d394)
(BuildId: b712d767295861216504d9a7452e2d6a08d25cab)
                                          << read the stack frames bottom-up >>

Address 0x7fc572500025 is located in stack of thread T0 at offset 37 in frame
    #0 0x4f4903 in read_overflow_compilemem <...>/Hands-on-System-Programming-
with-Linux/ch5/membugs.c:234:1

  This frame has 2 object(s):
```

```
    [32, 37) 'arr' (line 235) <== Memory access at offset 37 overflows this
variable
    [64, 72) 'tmp' (line 235)
...
```

3

3. The 'arr' variable lives in the stack frame at a start offset of 32; its size is 5 bytes (as per the code); hence, it spans from byte 32 to 36. BUT a (read) memory access (of size 6 bytes) has occurred at offset 37 – a read overflow bug!

```
Shadow bytes around the buggy address:
  0x7fc5724ffd80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x7fc5724ffe00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x7fc5724ffe80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x7fc5724fff00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x7fc5724fff80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x7fc572500000: f1 f1 f1 f1[05]f2 f2 f2 00 f3 f3 f3 00 00 00 00
  0x7fc572500080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x7fc572500100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x7fc572500180: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x7fc572500200: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x7fc572500280: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
…
```

How to interpret the (K)ASAN shadow memory report

- Firstly, "… `one shadow byte represents 8 application bytes`"
- If a byte is fine - has no memory issues - it shows as `00` (IOW, it's *addressable*)
- If the shadow memory map shows any 'coloured' bytes, look them up and interpret them via the very clear legend seen lower in the ouput; so in this example:
    - `f1` => stack left redzone
    - `f2` => stack mid redzone
    - `f3` => stack right redzone

<< [1] see below for more shadow memory flag byte meaning >>

(A **redzone** is a deliberately placed memory region typically around (and perhaps even in-between) the application / kernel memory bytes in question, and set deliberately to have no permissions, so as to be able to catch erronous memory accesses that are made to any byte within it.)

- If a shadow bytes shows up in square brackets with **any value between `[01]` and `[07]`**, it implies it was '**partially addressable**'.
  - Read it like this: if the value is `[0n]`, then
    - the first `n` bytes of memory are accessible (fine)
    - the remaining `(8-n)` memory bytes aren't legally accessible
    - so, typically, the memory access attempted on the `n`th byte (onward) was faulty in some manner!

- **If the Shadow memory < 0**: A negative value implies the entire 8-byte granule is inaccessible. The particular (negative) values and their meaning (already freed-up memory, red zone region, and so on) are encoded in the kernel header file *mm/kasan/kasan.h*

In this demo run, some partial output from ASAN is reproduced below:

```
arr = 0x7fc572500020 tmp= 0x7fc572500040
...
...
=>0x7fc572500000: f1 f1 f1 f1[05]f2 f2 f2 00 f3 f3 f3 00 00 00 00
```

  - The hex number on the extreme left (with the arrow pointing to it) is where (K)ASAN thinks the bug occurred!
    - We printed out the (virtual) addresses of the vars `arr[]` and `tmp[]` (in purple colour)
    - See how perfectly it matches what (K)ASAN shows !
    - Remember, each byte of shadow memory is 8 bytes of actual memory (or '8 granules')
      - so, starting at `0x7fc572500000,` 0x20 bytes from here is 32 bytes down… Now 32 / 8 = 4, i.e. 4 'granules' of actual memory down…
      - that's why it shows up after four 0xf1 (stack left redzone) granules !

```
=>0x7fc572500000: f1 f1 f1 f1[05]f2 f2 f2 00 f3 f3 f3 00 00 00 00
                  |---- 4---|
                   granules
```

  - Here, the *partially addressable* memory granule shows as **`[05]`**
    - Thus [K]ASAN is saying that
      - the memory accesses to the first 5 bytes in this granule were fine
      - the remaining (8-5) = 3 bytes aren't legally accessible
      - in effect, the memory access to the `5`th byte of this granule by the app/kernel/module was faulty! Careful: 5th byte implies the byte at index position 4 of course: 0,1,2,3,4
    - Thus here, code-wise:

```
static void buggy1(void)
{
      char arr[5], tmp[8];

      memset(arr, 'a', 5);
```

```
        memset(tmp, 't', 8);
        tmp[7] = '\0';

        printf("arr = %s\n", arr);  /* Bug: read buffer overflow */
}
```

- `arr[5]` has bytes 0-4 valid. But we attempt to access memory (read) beyond this position via the `printf()` ! - it attempted to read memory until the NULL byte which is at index position 7 of `tmp[8]`; this triggered the (read overflow) bug!

```
             arr[5]              tmp[8]          : total of 13 bytes
        0   1   2   3   4   0   1   2   3   4   5   6   7
        0   1   2   3   4   5   6   7   8   9   10  11  12
Value:  a   a   a   a   a   t   t   t   t   t   t   t   0
                            ^
```

Attempting to access this byte is the bug! Why?
As it's beyond the legal bounds of the array `arr[]`...
IOW, a *read overflow defect…*
It's byte # 5 of the faulty memory access, as ASAN's report correctly points out!

[1] Shadow memory - bytes meaning
[*src*]
...
Each byte of the shadow region encodes the status of the address to be accessed as:
`0x00` All 8 bytes are accessible.
`0x00 < N < 0x08` The lower `N` bytes are accessible. Bytes `8 - N` to byte `8` are not.
`0xff` The page was freed.
`0xfe` A redzone for `kmalloc_large`.
`0xfc` A redzone for a slub object.
`0xfb` The object was freed.
`0xf5` Stack use after return.
`0xf8` Stack use after scope.
...

Kernel ASAN - KASAN - example

For kernel: require GCC >= 8.3 or Clang (11+):

```
-fsanitize=kernel-address
```

From my *Linux Kernel Debugging* book:

188 Debugging Kernel Memory Issues – Part 1

The following table neatly summarizes some key information about KASAN:

KASAN Mode	GCC	Clang	Internal working	Platforms supported	Suitable for
Generic KASAN	>= 8.3.0	Any (>= 11 for OOB on global variables)	CTI	x86_64, ARM, ARM64, Xtensa, S390, RISC-V	Development/debug only; global variables also instrumented; SLUB and SLAB implementation
Software tag-based KASAN	Not supported		CTI	Currently only on ARM64 (hardware tag-based: requires ARMv8.5 or later with **Memory Tagging Extension**)	Dev/debug and production; hardware tag-based requires SLUB implementation
Hardware tag-based KASAN	>= 10+	>= 11+	hardware tag-based		

Table 5.2 – Types of KASAN and compiler/hardware support requirements

Even supports ARM-32 from 5.11 ! (unsure about how well it works though).

Kernel Configuration : from 'make menuconfig' :

```
Kernel hacking > Memory Debugging > KASAN: runtime memory debugger
```

```
$ grep KASAN /boot/config-6.5.9
CONFIG_KASAN_SHADOW_OFFSET=0xdffffc0000000000
CONFIG_HAVE_ARCH_KASAN=y
CONFIG_HAVE_ARCH_KASAN_VMALLOC=y
CONFIG_CC_HAS_KASAN_GENERIC=y
CONFIG_KASAN=y
CONFIG_CC_HAS_KASAN_MEMINTRINSIC_PREFIX=y
CONFIG_KASAN_GENERIC=y
CONFIG_KASAN_OUTLINE=y
# CONFIG_KASAN_INLINE is not set
CONFIG_KASAN_STACK=y
CONFIG_KASAN_VMALLOC=y
```

```
CONFIG_KASAN_MODULE_TEST=m
$
```

So the kernel itself (+ all modules) is now built with the `-fsanitize=kernel-address` compiler option on.

*"… KASAN works essentially by being able to check every single memory access; it does this by using a technique called **Compile Time Instrumentation (CTI)**.*

Put very simplistically, the compiler inserts function calls (__asan_load() and __asan_store*()) before every 1-, 2-, 4-, 8-, or 16-byte memory access.*

Thus, the runtime can figure out whether the access is valid or not (by checking the corresponding shadow memory bytes).

Now, there are two broad ways the compiler can perform this instrumentation: outline and inline. Outline instrumentation has the compiler inserting actual function calls (as just mentioned); inline instrumentation achieves the same thing but in a time-optimized manner by directly inserting the code (and not having the overhead of a function call)! …"

Outline instrumentation (default): smaller (kernel image) but slower
Inline instrumentation : larger (kernel image) but faster

Can use GCC or clang. NOTE : the kernel image and all modules must be built with the same compiler of course.
Can check with `CONFIG_CC_IS_GCC=y` or `CONFIG_CC_IS_CLANG=y`.
(Recommended : clang 11 +).

Trying out KASAN

With the Kunit test module

You can use the kernel's builtin KUnit test infrastructure to run KASAN test cases!
With
`CONFIG_KASAN_KUNIT_TEST=m`

```
# modprobe test_kasan
# dmesg
...
```

With a custom module that runs test cases:

From my *Linux Kernel Debugging* book, *Ch 5* GitHub repo:
https://github.com/PacktPublishing/Linux-Kernel-Debugging/tree/main/ch5/kmembugs_test .

Build the module with the compiler specified via the `CC=gcc|clang` environment variable (the `load_testmod` script automates this).
Simply run the `run_tests` script to then run test cases on a KASAN-enabled kernel.

Screenshots follow, showing the setup and a few test runs (followed by a table showing which memory defects KASAN actually caught):

```
kmembugs_test $ ./load_testmod
Kernel ver: 6.5.9
UBSAN enabled
Generic KASAN enabled
This kernel has been built with gcc

--- Building : KDIR=/lib/modules/6.5.9/build ARCH= CROSS_COMPILE= ccflags-y=-DDEBUG -g -ggdb -gdwarf-4 -Wall -fno-omit-f
rame-pointer -fvar-tracking-assignments -DDYNAMIC_DEBUG_MODULE ---
gcc (Ubuntu 13.2.0-4ubuntu3) 13.2.0

make -C /lib/modules/6.5.9/build M=/home/c2kp/kaiwanTECH/Linux-Kernel-Debugging/ch5/kmembugs_test modules
make[1]: Entering directory '/home/c2kp/kernels/linux-6.5.9'
make[1]: Leaving directory '/home/c2kp/kernels/linux-6.5.9'
[126230.617317] test_kmembugs:kmembugs_test_init(): KASAN configured
[126230.617322] test_kmembugs:kmembugs_test_init(): CONFIG_UBSAN configured
[126230.617325] test_kmembugs:kmembugs_test_init(): CONFIG_DEBUG_KMEMLEAK NOT configured
[126230.617351] debugfs file 1 <debugfs_mountpt>/test_kmembugs/lkd_dbgfs_run_testcase created
[126230.617354] debugfs entry initialized
kmembugs_test $ lsmod |grep kmembugs
test_kmembugs          45056  0
kmembugs_test $ _
```

<< P.T.O. → >>

(Tip:
In case you get compile errors like '*-fvar-tracking-assignments: unknown option …*' , it's typically the case that you're using a lower compiler version than is required. This happened to me when running the testcase on Ubuntu 22.04; switching to an Ubuntu 23.10 VM solved it (as it installed the later version compiler).

Running the test cases via the run_tests helper script

```
kmembugs_test $ sudo ./run_tests
Debugfs file: /sys/kernel/debug/test_kmembugs/lkd_dbgfs_run_testcase

Generic KASAN: enabled
UBSAN: enabled
KMEMLEAK: disabled

Select testcase to run:
1  Uninitialized Memory Read - UMR
2  Use After Return - UAR

Memory leakage
3.1  simple memory leakage testcase1
3.2  simple memory leakage testcase2 - caller to free memory
3.3  simple memory leakage testcase3 - memleak in interrupt ctx

OOB accesses on static (compile-time) global memory + on stack local memory
4.1  Read  (right) overflow
4.2  Write (right) overflow
4.3  Read  (left) underflow
4.4  Write (left) underflow

OOB accesses on dynamic (kmalloc-ed) memory
5.1  Read  (right) overflow
5.2  Write (right) overflow
5.3  Read  (left) underflow
5.4  Write (left) underflow

6  Use After Free - UAF
7  Double-free

UBSAN arithmetic UB testcases
8.1  add overflow
8.2  sub overflow
8.3  mul overflow
8.4  negate overflow
8.5  shift OOB
8.6  OOB
8.7  load invalid value
8.8  misaligned access
8.9  object size mismatch

9  copy_[to|from]_user*() tests
10 UMR on slab (SLUB) memory

(Type in the testcase number to run):
```

Select 5.1 : OOB accesses on dynamic (kmalloc-ed) memory

5.1 Read (right) overflow

Here’s the relevant module code (link):

```
/* OOB on dynamic (kmalloc-ed) mem: OOB read/write (right) overflow */
int dynamic_mem_oob_right(int mode)
{
	volatile char *kptr, ch = 0;
	char *volatile ptr;
	size_t sz = 32;

	kptr = kmalloc(sz, GFP_KERNEL);
	if (unlikely(!kptr))
		return -ENOMEM;
	ptr = (char *)kptr + sz + 3; // right OOB

	if (mode == READ) {
	/* Interesting: this OOB access isn't caught by UBSAN but is caught by 
KASAN! */
		ch = *(volatile char *)ptr; // invalid, OOB right write
		/* ... but these below OOB accesses are caught by KASAN/UBSAN.
		 * We conclude that *only* the index-based accesses are caught by 
UBSAN.
		 */
		ch = kptr[sz + 3];	// invalid, OOB right read   << the bug we 
trigger here >>
	} else if (mode == WRITE) {
	/* Interesting: this OOB access isn't caught by UBSAN but is caught by 
KASAN! */
		*(volatile char *)ptr = 'x';
		/* ... but these below OOB accesses are caught by KASAN/UBSAN.
		 * We conclude that *only* the index-based accesses are caught by 
UBSAN.
		 */
		kptr[sz] = 'x';   // invalid, OOB right write
	}

	kfree((char *)kptr);
	return 0;
}

...



# dmesg
[ ... ]
```

```
-------- Running testcase "5.1" via test module now...
[132045.099885] testcase to run: 5.1
[132045.099927] ==================================================================
[132045.099949] BUG: KASAN: slab-out-of-bounds in dynamic_mem_oob_right+0xab/0x130 [test_kmembugs]
[132045.099961] Read of size 1 at addr ffff88801088a723 by task run_tests/37657

[132045.099967] CPU: 5 PID: 37657 Comm: run_tests Tainted: G    B      OE      6.5.9 #2 b2f304818a
[132045.099973] Hardware name: innotek GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
[132045.099976] Call Trace:
[132045.099978]  <TASK>
[132045.099981]  dump_stack_lvl+0x5f/0xc0
[132045.099987]  print_report+0xd2/0x670
[132045.099992]  ? __pfx__raw_spin_lock_irqsave+0x10/0x10
[132045.099997]  ? __virt_addr_valid+0x103/0x180
[132045.100003]  ? kasan_complete_mode_report_info+0x40/0x230
[132045.100009]  kasan_report+0xd7/0x120
[132045.100013]  ? dynamic_mem_oob_right+0xab/0x130 [test_kmembugs 141be6d7073a312520df34fba13a0cd
[132045.100024]  ? dynamic_mem_oob_right+0xab/0x130 [test_kmembugs 141be6d7073a312520df34fba13a0cd
[132045.100035]  __asan_load1+0x6c/0x80
[132045.100039]  dynamic_mem_oob_right+0xab/0x130 [test_kmembugs 141be6d7073a312520df34fba13a0cde3
[132045.100051]  ? __pfx_dynamic_mem_oob_right+0x10/0x10 [test_kmembugs 141be6d7073a312520df34fba1
[132045.100063]  dbgfs_run_testcase+0x396/0x490 [test_kmembugs 141be6d7073a312520df34fba13a0cde3b5
[132045.100073]  ? __pfx_dbgfs_run_testcase+0x10/0x10 [test_kmembugs 141be6d7073a312520df34fba13a0
[132045.100083]  ? locks_remove_posix+0xbd/0x310
[132045.100089]  full_proxy_write+0x99/0xd0
[132045.100096]  vfs_write+0x1b8/0x7a0
[132045.100102]  ? __pfx_vfs_write+0x10/0x10
[132045.100108]  ? __kasan_check_read+0x11/0x20
[132045.100112]  ? __fget_light+0x1c6/0x220
[132045.100119]  ksys_write+0xd9/0x180
[132045.100124]  ? __pfx_ksys_write+0x10/0x10
[132045.100129]  ? syscall_enter_from_user_mode+0x17/0x70
[132045.100134]  ? do_syscall_64+0x36/0x90
[132045.100139]  __x64_sys_write+0x42/0x60
[132045.100144]  do_syscall_64+0x5c/0x90
[132045.100147]  ? syscall_exit_to_user_mode+0x3b/0x60
[132045.100152]  ? do_syscall_64+0x68/0x90
[132045.100156]  ? exc_page_fault+0x91/0x140
[132045.100161]  entry_SYSCALL_64_after_hwframe+0x6e/0xd8
[132045.100166] RIP: 0033:0x7f512911b214
[132045.100181] Code: c7 00 16 00 00 00 b8 ff ff ff ff c3 66 2e 0f 1f 84 00 00 00 00 00 f3 0f 1e f
3 ec 28 48 89 54 24 18 48
[132045.100185] RSP: 002b:00007ffefbcdd1e8 EFLAGS: 00000202 ORIG_RAX: 0000000000000001
[132045.100190] RAX: ffffffffffffffda RBX: 0000000000000004 RCX: 00007f512911b214
[132045.100193] RDX: 0000000000000004 RSI: 000055b9243fb8b0 RDI: 0000000000000001
[132045.100196] RBP: 000055b9243fb8b0 R08: 0000000000000073 R09: 0000000000000000
[132045.100199] R10: 0000000000000000 R11: 0000000000000202 R12: 0000000000000004
```

KASAN catches it!

```
[132045.099949] BUG: KASAN: slab-out-of-bounds in
dynamic_mem_oob_right+0xab/0x130 [test_kmembugs]
[132045.099961] Read of size 1 at addr ffff88801088a723 by task
run_tests/37657
```
…'

Quick Debug Tips

- Can see :

`function_name+x/y [module_name]` (here:)
`dynamic_mem_oob_right+0xab/0x130`

x : ‘distance’ in bytes (of the machine code) from the start of the function
y : length of the function in bytes.

(With this info, using `objdump -dS <x.ko>` or `readelf` or `GDB` can be very helpful …).

- Read the stack trace bottom-up; the call trace is indeed a huge clue! (showing the ‘history’, how we got here)
- Ignore (stack) frames that begin ‘`?`’; it’s likely a leftover ‘blip’ from earlier usage of the same (kernel) stack memory

[…]

```
[132045.100208]  </TASK>

[132045.100212] Allocated by task 37657:
[132045.100214]  kasan_save_stack+0x38/0x70
[132045.100219]  kasan_set_track+0x25/0x40
[132045.100223]  kasan_save_alloc_info+0x1e/0x40
[132045.100227]  __kasan_kmalloc+0xc3/0xd0
[132045.100231]  kmalloc_trace+0x48/0xc0
[132045.100234]  dynamic_mem_oob_right+0x86/0x130 [test_kmembugs]
[132045.100242]  dbgfs_run_testcase+0x396/0x490 [test_kmembugs]
[132045.100250]  full_proxy_write+0x99/0xd0
[132045.100255]  vfs_write+0x1b8/0x7a0
[132045.100258]  ksys_write+0xd9/0x180
[132045.100261]  __x64_sys_write+0x42/0x60
[132045.100265]  do_syscall_64+0x5c/0x90
[132045.100268]  entry_SYSCALL_64_after_hwframe+0x6e/0xd8

[132045.100273] The buggy address belongs to the object at ffff88801088a700
                 which belongs to the cache kmalloc-32 of size 32
[132045.100276] The buggy address is located 3 bytes to the right of
                 allocated 32-byte region [ffff88801088a700, ffff88801088a720)

[132045.100282] The buggy address belongs to the physical page:
[132045.100285] page:0000000067c46aeb refcount:1 mapcount:0 mapping:0000000000000000 index:0x0 pfn:0x1088a
[132045.100310] ksm flags: 0xfffffc0000200(slab|node=0|zone=1|lastcpupid=0x1fffff)
[132045.100314] page_type: 0xffffffff()
[132045.100319] raw: 000fffffc0000200 ffff888001042500 ffffea000023c140 dead000000000003
[132045.100322] raw: 0000000000000000 0000000080400040 00000001ffffffff 0000000000000000
[132045.100324] page dumped because: kasan: bad access detected

[132045.100328] Memory state around the buggy address:
[132045.100331]  ffff88801088a600: 00 00 05 fc fc fc fc fc 00 00 01 fc fc fc fc fc
[132045.100351]  ffff88801088a680: 00 00 00 00 fc fc fc fc 00 00 00 fc fc fc fc fc
[132045.100354] >ffff88801088a700: 00 00 00 00 fc fc fc fc 00 00 07 fc fc fc fc fc
[132045.100357]                                ^
[132045.100359]  ffff88801088a780: 00 00 01 fc fc fc fc fc fb fb fb fb fc fc fc fc
[132045.100363]  ffff88801088a800: 00 00 00 00 fc fc fc fc 00 00 00 00 fc fc fc fc
[132045.100365] ==================================================================
```

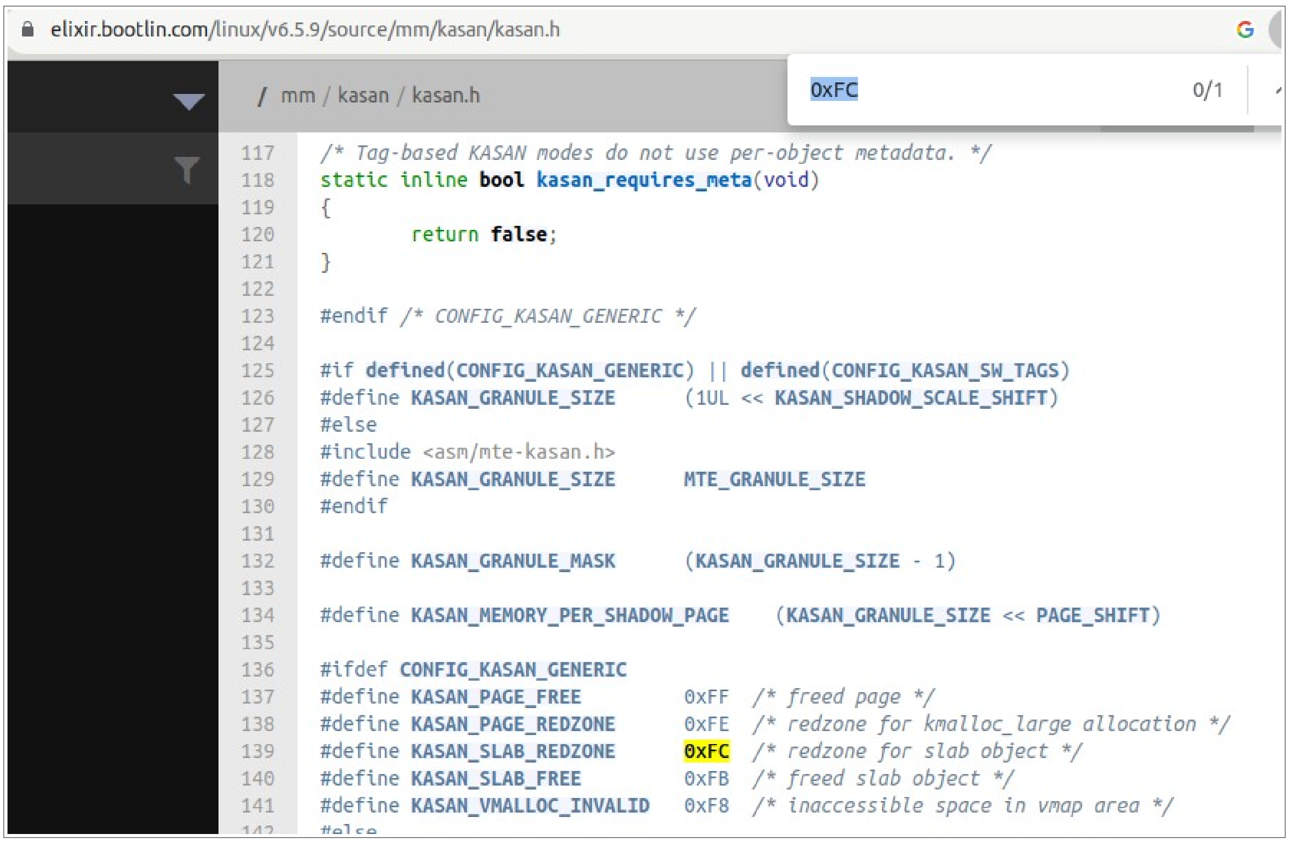
The kernel virtual address region where the bad access occurred...

Reproduced:

```
>ffff88801088a700: 00 00 00 00 fc fc fc fc 00 00 07 fc fc fc fc fc
                               ^
```

Interpret the KASAN shadow memory report, as before:

- `00` => the 8-byte memory granule is okay (legal access)
- we have four of them (zero pairs); thus the 4*8 = 32 bytes from `ffff88801088a700` was legally accessed
- the next byte shows as **`0xfc`** ; it's defined here: https://elixir.bootlin.com/linux/v6.5.9/source/mm/kasan/kasan.h#L139 :




```
117 /* Tag-based KASAN modes do not use per-object metadata. */
118 static inline bool kasan_requires_meta(void)
119 {
120         return false;
121 }
122
123 #endif /* CONFIG_KASAN_GENERIC */
124
125 #if defined(CONFIG_KASAN_GENERIC) || defined(CONFIG_KASAN_SW_TAGS)
126 #define KASAN_GRANULE_SIZE      (1UL << KASAN_SHADOW_SCALE_SHIFT)
127 #else
128 #include <asm/mte-kasan.h>
129 #define KASAN_GRANULE_SIZE      MTE_GRANULE_SIZE
130 #endif
131
132 #define KASAN_GRANULE_MASK      (KASAN_GRANULE_SIZE - 1)
133
134 #define KASAN_MEMORY_PER_SHADOW_PAGE    (KASAN_GRANULE_SIZE << PAGE_SHIFT)
135
136 #ifdef CONFIG_KASAN_GENERIC
137 #define KASAN_PAGE_FREE         0xFF  /* freed page */
138 #define KASAN_PAGE_REDZONE      0xFE  /* redzone for kmalloc_large allocation */
139 #define KASAN_SLAB_REDZONE      0xFC  /* redzone for slab object */
140 #define KASAN_SLAB_FREE         0xFB  /* freed slab object */
141 #define KASAN_VMALLOC_INVALID   0xF8  /* inaccessible space in vmap area */
142 #else
```

Ah, it's - just as with user-space ASAN - a redzone (here, for slab memory). The buggy access occurred there, which implies we 'spilled over' / did a 'right' overflow into the redzone following the slab memory region of 32 bytes that we allocated. This is precisely the case.

- But what access? read or write? This line gives the answer:
  `Read of size 1 at addr ffff88801088a723 by task run_tests/37657`

So, we had a read overflow - an **OOB (Out Of Bounds)** - defect / bug here.
(FYI, the line of code that triggered it:

```
    ch = kptr[sz + 3];     // invalid, OOB right read   << the bug we trigger
here >>
```

sz == 32; so sz+3 is 36, so we're doing `ch = kptr[36]` **which, clearly, is a read overflow**!)

This portion of the KASAN report literally spells it out:

```
...
    The buggy address is located 3 bytes to the right of
     allocated 32-byte region [ffff88801088a700, ffff88801088a720)
...
```

One more test case

4.4 Write (left) underflow

```
...
-------- Running testcase "4.4" via test module now...
[ 1705.566424] testcase to run: 4.4
[ 1705.566430]
==================================================================
[ 1705.566523] BUG: KASAN: global-out-of-bounds in
global_mem_oob_left+0x19c/0x200 [test_kmembugs]
[ 1705.566622] Write of size 1 at addr ffffffffc0a849bd by task run_tests/1780

[ ... ]
```

```
[ 1705.585054]  </TASK>

[ 1705.589542] The buggy address belongs to the variable:
[ 1705.591720]  global_arr3+0x3d/0xfffffffffffffc680 [test_kmembugs]

[ 1705.596728] Memory state around the buggy address:
[ 1705.599028]  ffffffffc0a84880: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[ 1705.601477]  ffffffffc0a84900: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[ 1705.604089] >ffffffffc0a84980: 00 02 f9 f9 f9 f9 f9 f9 00 02 f9 f9 f9 f9 f9 f9
[ 1705.606380]                                      ^
[ 1705.608632]  ffffffffc0a84a00: 00 02 f9 f9 f9 f9 f9 f9 00 00 00 00 f9 f9 f9 f9
[ 1705.611014]  ffffffffc0a84a80: 01 f9 f9 f9 f9 f9 f9 f9 00 f9 f9 f9 f9 f9 f9 f9
[ 1705.613267] ==================================================================
[ 1705.615590] Disabling lock debugging due to kernel taint
[ 1705.615623] ==================================================================
[ 1705.618064] BUG: KASAN: global-out-of-bounds in global_mem_oob_left+0x1b0/0x200 [test_kmembugs]
[ 1705.620666] Write of size 1 at addr ffffffffc0a849bd by task run_tests/1780
```

```
        >ffffffffc0a84980: 00 02 f9 f9 f9 f9 f9 f9 00 02 f9 f9 f9 f9 f9 f9
                                               ^
```

What's `0xf9` mean?? Lookup the kernel header…
It's the redzone for global data memory.

mm/kasan/kasan.h

```
...
#ifdef CONFIG_KASAN_GENERIC

#define KASAN_SLAB_FREETRACK  0xFA  /* freed slab object with free track */
#define KASAN_GLOBAL_REDZONE  0xF9  /* redzone for global variable */

/* Stack redzone shadow values. Compiler ABI, do not change. */
#define KASAN_STACK_LEFT      0xF1
#define KASAN_STACK_MID       0xF2
#define KASAN_STACK_RIGHT     0xF3
#define KASAN_STACK_PARTIAL   0xF4

/* alloca redzone shadow values. */
#define KASAN_ALLOCA_LEFT     0xCA
#define KASAN_ALLOCA_RIGHT    0xCB
...
```

Recall, the KASAN report showed:

```
Write of size 1 at addr ffffffffc0a849bd by task run_tests/1780
...

So ffffffffc0a849bd - ffffffffc0a84980 = 0x3D = 61.
```

Key line:

```
  >ffffffffc0a84980: 00 02 f9 f9 f9 f9 f9 f9 00 02 f9 f9 f9 f9 f9 f9
                                         ^

  >ffffffffc0a84980: 00 02 f9 f9 f9 f9 f9 f9 00 02 f9 f9 f9 f9 f9 f9
                                         ^
```

global ‘left’ redzone   global ‘right’ redzone

Aha; the bug occurred at the ‘left’ redzone (as the ‘^’ up arrow points there), indicating an underflow issue! (left => underflow (read/write) ; right => overflow (read/write)).

Next, in-between the left and right redzones must be the actual memory object… here, we see

`00 02`

in-between.

So, think on it, each shadow byte here is a memory granule representing 8 actual memory bytes; so here, 2*8 = 16 bytes.. The first one is fine (value `00`), BUT, the second granule is `02` implying that of the 8 bytes ONLY the first two are okay! So, the memory region must be 8+2 = 10 bytes in length. (This turns out to be correct; see the code below).

The line above says “`Write of size 1 …`”.
So now we know: *it’s a write underflow on global data.*

Here’s the relevant module code:

```
...
#define ARRSZ    10
char global_arr1[ARRSZ];
char global_arr2[ARRSZ];
char global_arr3[ARRSZ];
...
      global_mem_oob_left(WRITE, global_arr2);
...

int global_mem_oob_left(int mode, char *p)
{
      volatile char w, x, y, z;
      volatile char local_arr[20];
      char *volatile ptr = p - 3; // left OOB

      if (mode == READ) {
            [ ... ]
      } else if (mode == WRITE) {
            /* Interesting: this OOB access isn't caught by UBSAN but is caught
by KASAN! */
            *(volatile char *)ptr = 'w';
            ...
```

<<

Sample : Extract from my Linux Kernel Debugging book:

...

```
ptr = kmalloc(123, GFP_KERNEL);
...
ptr[123] = 'x';
...
```

198 Debugging Kernel Memory Issues – Part 1

Now, Generic KASAN's memory granule size is 8 bytes. So, among the 123 bytes allocated, the fifteenth memory granule is the one being written to (as *8 * 15 = 120*). The diagram that follows clearly shows the memory buffer and how it's been overflowed:

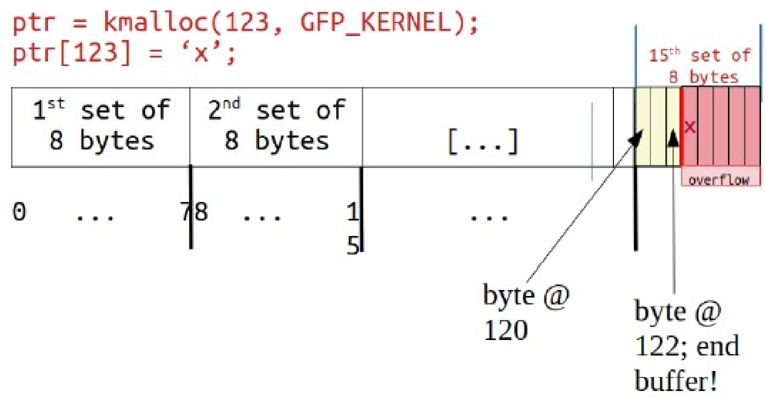


Figure 5.5 – The kmalloc'ed memory (slab) buffer that was overflowed

…

Shadow memory report:

```
...
>ffff8880318ad980: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03
                                                                 ^
...
```

>>

Table : which memory defects KASAN actually caught:
Extract from my *Linux Kernel Debugging* book:

KASAN – tabulating the results

What memory corruption bugs (defects) does KASAN actually manage to, and not manage to, catch? From our test runs, we tabulated the results in the table that follows. Do study it carefully, along with the notes that go with it:

Testcase # [1]	Memory defect type (below) / Infrastructure used (right)	Distro kernel [2]	Compiler warning? [3]	With KASAN [4]	With UBSAN [5]
Defects not covered by the kernel's KUnit test_kasan.ko module					
1	Uninitialized Memory Read – UMR	N	Y [C1]	N	N
2	Use After Return – UAR	N	Y [C2]	N [SA]	N [SA]
3	Memory leakage [6]	N	N	N	N
Defects covered by the kernel's KUnit test_kasan.ko module					
4	OOB accesses on static global (compile-time) memory				
4.1	Read (right) overflow		N	Y [K1]	
4.2	Write (right) overflow		N	Y [K1]	
4.3	Read (left) underflow		N	Y [K2]	
4.4	Write (left) underflow	N [V1]	N	Y [K2]	Y [U1,U2]
4	OOB accesses on static global (compile-time) stack local memory				
4.1	Read (right) overflow	N [V1]	N	Y [K3]	Y [U1,U2]
4.2	Write (right) overflow		N	Y [K2]	
4.3	Read (left) underflow		N	Y [K2]	
4.4	Write (left) underflow				
5	OOB accesses on dynamic (kmalloc-ed slab) memory				
5.1	Read (right) overflow				
5.2	Write (right) overflow				
5.3	Read (left) underflow				
5.4	Write (left) underflow	N	N	Y [K4]	N
6	Use After Free - UAF	N	N	Y [K5]	N
7	Double-free	Y [V2]	N	Y [K6]	N

8	Arithmetic UB (*via the kernel's test_ubsan.ko module*)				
8.1	Add overflow				Y
8.2	Sub(tract) overflow				**N**
8.3	Mul(tiply) overflow				**N**
8.4	Negate overflow				**N**
	Div by zero	N	N	N	Y
8.5	Bit shift OOB	Y [U3]		Y [U3]	Y [U3]
Other than arithmetic UB defects (copied from the kernel's KUnit test_ubsan.ko module)					
8.6	OOB	Y [U3]		Y [U3]	Y [U3]
8.7	Load invalid value	Y [U3]	N	Y [U3]	Y [U3]
8.8	Misaligned access	N		N	**N**
8.9	Object size mismatch	Y [U3]		Y [U3]	Y [U3]
9	OOB on `copy_[to|from]_user*()`	N	Y [C3]	Y [K4]	N

Table 5.3 – Summary of memory defect and arithmetic UB test cases caught (or not) by KASAN

You'll find the explanations for the footnote notations seen in the table (such as [C1], [U1], and so on) below.

Test environment

- [1] The test case number: do refer to the source of the test kernel module to see it – `ch5/kmembugs_test/kmembugs_test.c`, the debugfs entry creation and usage in `debugfs_kmembugs.c`, and the bash scripts `load_testmod` and `run_tests`, all within the same folder.
- [2] The compiler used here is GCC version 9.3.0 on x86_64 Ubuntu Linux. A later section - *Using Clang 13 on Ubuntu 21.10* - covers using the **Clang 13** compiler.
- [3] To test with KASAN, I had to boot via our custom debug kernel (5.10.60-dbg01) with `CONFIG_KASAN=y` and `CONFIG_KASAN_GENERIC=y`. We assume the Generic KASAN variant is being used.
- Test cases 4.1 through 4.4 work both upon static (compile-time allocated) global memory as well as stack local memory. That's why the test case numbers are 4.x in both.

The UMR bug isn't caught by KASAN..
Just as in userspace ASAN doesn't catch it. ; but **MSAN** - Memory Sanitizer - does!

So, the kernel now (6.1 +) has **KMSAN** - Kernel Memory Sanitizer - which does catch it!

Done :-) *

Thank you!

* No, we're never really 'done'.
<*Q. What's the biggest room in the world?*
A. The room for improvement.
>