



Kernel Memory Debug Tools

<<

From my [LKD \(Linux Kernel Debugging\) book](#):

“**Memory corruption** can occur due to various bugs or defects:

- *Uninitialized Memory Reads (UMR)*
- *Use After Free (UAF)*
- *Use After Return (UAR)* << aka *Use After Scope (UAS)* >>
- *double-free*
- *memory leakage*
- *illegal Out Of Bounds (OOB) accesses* -that attempt to work upon (read/write/execute) illegal memory regions.

They're unfortunately a very common root cause of bugs. Being able to debug them is a key skill ...”

Type of memory bug or defect	Tool(s)/techniques to detect it
Uninitialized Memory Reads (UMR)	Compiler (warnings) [1], static analysis
Out-of-bounds (OOB) memory accesses: read/write underflow/overflow defects on compile-time and dynamic memory (including the stack)	KASAN [2], SLUB debug
Use-After-Free (UAF) or dangling pointer defects (aka Use-After-Scope (UAS) defects)	KASAN, SLUB debug
Use-After-Return (UAR) aka UAS defects	Compiler (warnings), static analysis
Double-free	Vanilla kernel [3], SLUB debug, KASAN
Memory leakage	kmemleak

Table 5.1 – A summary of tools (and techniques) you can use to detect kernel memory issues

>>

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), OOB (out-of-bounds), double/invalid free bugs. Requires GCC ≥ 8.3 (for usermode GCC ≥ 5.0), 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 <code>-fsanitize=kernel-address</code>	4.0	x86_64 from 4.0, ARM64 from 4.4, xtensa, s390 v5.11 (Feb 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 <code>-fsanitize=undefined</code> 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 (<i>only</i> 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 <code>slub_debug=</code> option on kernel cmdline)	2.6.23	<all?>
KMSAN (Kernel Memory Sanitizer) (<i>new</i>)	Detects UMR defects in the kernel; (similar to userspace MSAN); not for production	6.1	Only x86_64. Requires Clang 14.0.6 +
Kmemcheck <i>Removed in 4.15 *</i>	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

* **Update!** From 4.15, *kmemcheck* has been removed! The commit (16 Nov 2017):

[*kmemcheck: remove annotations:*](#)

“As discussed at LSF/MM, kill kmemcheck.

KASan is a replacement that is able to work without the limitation of kmemcheck (single CPU, slow). KASan is already upstream. We are also not aware of any users of kmemcheck (or users who don't consider KASan as a suitable replacement).

The only objection was that since KASAN wasn't supported by all GCC versions provided by distros at that time we should hold off for 2 years, and try again.

Now that 2 years have passed, and all distros provide gcc that supports KASAN, kill kmemcheck again for the very same reasons.

...”

KASAN – the Kernel Address SANitizer

Official kernel doc: <https://www.kernel.org/doc/html/latest/dev-tools/kasan.html>

Kernel Address SANitizer (KASAN) is a dynamic memory corruption or memory error detector for the Linux kernel, designed to find out-of-bounds and use-after-free bugs.

...

<<

From 2022, LSS Europe: Sanitizing the Linux kernel [slides](#):

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

>>

KASAN uses **compile-time instrumentation to insert validity checks before every memory access**, and therefore requires a compiler version that supports that. From the *official kernel doc*:

- Generic KASAN requires GCC version 8.3.0 or later or any Clang version supported by the kernel.
- Software Tag-Based KASAN requires GCC 11+ or any Clang version supported by the kernel.

- Hardware Tag-Based KASAN requires GCC 10+ or Clang 12+.

...

From my LKD 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+		hardware tag-based		

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

*CTI is Compile Time Instrumentation

Ref:

<https://google.github.io/kernel-sanitizers/KASAN> <excellent>

[2022, LSS Europe: Sanitizing the Linux kernel](#)

[Finding bugs with sanitizers, Jake Edge, LWN, Sept 2022](#)

[KASan for ARM32 Decompression Stop](#)

KASAN Test cases

Refer my LKD book's Ch 5 GitHub test case code:

https://github.com/PacktPublishing/Linux-Kernel-Debugging/tree/main/ch5/kmembugs_test

To run the tests, follow these steps:

0. Ensure you're running a 'debug kernel' with both UBSAN and KASAN configs enabled

1. Run the following command:

```
cd <book_src>/ch5/kmembugs_test
```

2. Load it up:
`./load_testmod`

3. Run our bash script to test:
`sudo ./run_tests`
 ...

An experiment: trying out KASAN on a Raspberry Pi 4 (AArch64)

- will require a custom 64-bit kernel (KASAN is supported only on 64-bit)
- hence, let's build for ourselves a custom 64-bit kernel for the R Pi 4
- ref: <https://www.raspberrypi.org/documentation/linux/kernel/building.md>

...

Within the source tree:

```
make mrproper
KERNEL=kernel8
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- bcm2711_defconfig
```

Optional, tweak config:
`make ARCH=arm64 menuconfig`

Now it's **important** to turn on, under the *Kernel Hacking* menu

- Compile-time checks and compiler options > kernel debug
- Memory debugging > KASAN: runtime memory debugger

Build:

```
[time] make -j['n'] ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- Image modules dtbs
```

Install kernel modules directly onto the [u]SDcard rootfs (ext4) partition (**careful !!! ensure `INSTALL_MOD_PATH` is correctly set!**) ; for example:

```
sudo env PATH=$PATH make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu-  
INSTALL_MOD_PATH=/media/${USER}/rootfs/ modules_install
```

“Finally, copy the kernel and Device Tree blobs onto the SD card, making sure to back up your old kernel:”

```
sudo cp mnt/fat32/$KERNEL.img mnt/fat32/$KERNEL-backup.img
sudo cp arch/arm64/boot/Image mnt/fat32/$KERNEL.img
sudo cp arch/arm64/boot/dts/broadcom/*.dtb mnt/fat32/
sudo cp arch/arm64/boot/dts/overlays/*.dtb* mnt/fat32/overlays/
sudo cp arch/arm64/boot/dts/overlays/README mnt/fat32/overlays/
sudo umount mnt/fat32
sudo umount mnt/ext4
```

(Substitute `mnt/fat32` and `mnt/ext4` with appropriate values for the SD card's boot and rootfs mount partitions).

Tip:

- in the SDcard *boot* partition, in the file *config.txt* : add the lines
 kernel=kernel8.img
 arm64_bit=1
-

Now that we're all set, lets run our test cases via our (deliberately buggy!) *membugs_kasan.ko* LKM.

Note-

- Pass *kasan_multi_shot* (add to kernel cmdline via */boot/cmdline.txt*):
From <https://www.kernel.org/doc/html/latest/admin-guide/kernel-parameters.html>
 kasan_multi_shot
 [KNL] Enforce KASAN (Kernel Address Sanitizer) to print
 report on every invalid memory access. Without this
 parameter KASAN will print report **only for the first**
 invalid access.
- I had to cross-compile the kernel module on the host system (as it won't build on the target, the R Pi 4, as it's a custom kernel and no kernel headers package is present):

```
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu-
```

... then transfer it (scp) to the target and try it out there...


```

rpi4 membugs_kasan $ ./test.sh 3
-rw-r--r-- 1 pi pi 278384 Feb 15 12:26 membugs_kasan.ko
sudo rmmod membugs_kasan 2>/dev/null ; sudo dmesg -C; sudo insmod ./membugs_kasan.ko testcase=3; dmesg
[ 4142.967905] membugs_kasan: inserted; testcase = 3
[ 4142.967940] kp = 0xffffffff8066367000
[ 4142.967947] ** Test case :: OOB write underflow dynamic-mem [func oob_write_overflow_dynmem()] **
[ 4142.967953] *p = 0xffffffff8066367000
[ 4142.967961] =====
[ 4142.967967] BUG: KASAN: slab-out-of-bounds in membugs_kasan_init+0x1b4/0x1000 [membugs_kasan]
[ 4142.967986] Write of size 2056 at addr ffffffff8066367000 by task insmod/1027

[ 4142.968000] CPU: 0 PID: 1027 Comm: insmod Tainted: G          0      5.4.70-rt40-v8-dbgk+ #1
[ 4142.968010] Hardware name: Raspberry Pi 4 Model B Rev 1.4 (DT)
[ 4142.968015] Call trace:
[ 4142.968018] dump_backtrace+0x0/0x280
[ 4142.968031] show_stack+0x28/0x38
[ 4142.968037] dump_stack+0xf8/0x180
[ 4142.968045] print_address_description.isra.0+0x74/0x354
[ 4142.968056] __kasan_report+0x134/0x230
[ 4142.968062] kasan_report+0xc/0x18
[ 4142.968069] check_memory_region+0x154/0x1a8
[ 4142.968074] memset+0x30/0x50
[ 4142.968081] membugs_kasan_init+0x1b4/0x1000 [membugs_kasan]
[ 4142.968091] do_one_initcall+0xbc/0x6d8
[ 4142.968098] do_init_module+0x18c/0x630
[ 4142.968105] load_module+0x4ee8/0x6638
[ 4142.968110] __do_sys_finit_module+0x158/0x170
[ 4142.968116] __arm64_sys_finit_module+0x74/0xa8
[ 4142.968121] el0_svc_common.constprop.0+0x114/0x3a0
[ 4142.968129] el0_svc_compat_handler+0x48/0x80
[ 4142.968136] el0_svc_compat+0x8/0x10

[ ... ]

```

```

[ 4142.968337] The buggy address belongs to the object at ffffffff8066367000
[ 4142.968343] which belongs to the cache kmalloc-2k of size 2048
[ 4142.968343] The buggy address is located 0 bytes inside of
[ 4142.968350] 2048-byte region [fffffff8066367000, ffffffff8066367800)
[ 4142.968350] The buggy address belongs to the page:
[ 4142.968354] page:ffffffffff0178d800 refcount:1 mapcount:0 mapping:fffff806d403400 index:0x0 compound_mapcount: 0
[ 4142.968362] flags: 0x4000000000010200(slab|head)
[ 4142.968377] raw: 4000000000010200 dead000000000100 dead000000000122 ffffff806d403400
[ 4142.968386] raw: 0000000000000000 0000000080080008 00000001ffffffff 0000000000000000
[ 4142.968390] page dumped because: kasan: bad access detected

[ 4142.968396] Memory state around the buggy address:
[ 4142.968402] ffffffff8066367700: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[ 4142.968408] ffffffff8066367780: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[ 4142.968414] >fffff8066367800: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
[ 4142.968418] ^
[ 4142.968423] ffffffff8066367880: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
[ 4142.968428] ffffffff8066367900: fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
[ 4142.968432] =====
[ 4142.968435] Disabling lock debugging due to kernel taint
rpi4 membugs_kasan $ █

```

Testcase 5

```

...
[ 81.460753] addr ffffffff80693d76f4 is located in stack of task insmod/582 at
[ 81.460757] membugs_kasan_init+0x0/0x1000 [membugs_kasan]

[ 81.460767] this frame has 2 objects:
[ 81.460774] [32, 52) 'arr'

```

```
[ 81.460779] [96, 105) 'src'

...
[ 81.460807]
=====
[ 81.460811] arr[5] = -48
[ 81.460816]
=====
[ 81.460818] BUG: KASAN: stack-out-of-bounds in
membugs_kasan_init+0x238/0x1000 [membugs_kasan]
[ 81.460825] Read of size 4 at addr ffffffff80693d76f8 by task insmod/582

[ 81.460831] CPU: 3 PID: 582 Comm: insmod Tainted: G      B      0      5.4.70-
rt40-v8-dbgk+ #1
[ 81.460836] Hardware name: Raspberry Pi 4 Model B Rev 1.4 (DT)
[ 81.460839] Call trace:
[ 81.460840] dump_backtrace+0x0/0x280
[ 81.460845] show_stack+0x28/0x38
[ 81.460849] dump_stack+0xf8/0x180
[ 81.460853] print_address_description.isra.0+0x74/0x354
[ 81.460858] __kasan_report+0x134/0x230
[ 81.460862] kasan_report+0xc/0x18
[ 81.460866] __asan_report_load4_noabort+0x1c/0x28
[ 81.460870] membugs_kasan_init+0x238/0x1000 [membugs_kasan]
[ 81.460876] do_one_initcall+0xbc/0x6d8
[ 81.460880] do_init_module+0x18c/0x630
[ 81.460883] load_module+0x4ee8/0x6638

...
```

Test cases - Summary

On the Raspberry Pi 4 w/ custom debug kernel : *Caught by KASAN ?*

test case 1 : uninitialized var test case : **no**

test case 2 : out-of-bounds : write overflow [on compile-time memory] : **no**

test case 3 : out-of-bounds : write overflow [on dynamic memory] : **yes** (see screenshots above)

test case 4 : out-of-bounds : write underflow : **yes**

test case 5 : out-of-bounds : read overflow [on compile-time memory]: **yes**

test case 6 : out-of-bounds : read overflow [on dynamic memory]: **yes**

test case 7 : out-of-bounds : read underflow: **yes**

test case 8 : UAF (use-after-free): **yes**

test case 9 : UAR (use-after-return): **no**

test case 10 : double-free : **yes**

[4600.201389] BUG: KASAN: double-free or invalid-free in
membugs_kasan_init+0x46c/0x1000 [membugs_kasan]

test case 11 : memory leak : simple leak: **no**

ALL types - Summary

Test case #	Test Case	Vanilla kernel [1]	With <code>slub_debug=on</code> kernel command-line [1]	With KASAN enabled [2]
1	UMR – uninitialized memory read	N	N	N (use UBSan) *[A]
2	OOB – out-of-bounds – write overflow on compile-time memory	N	N	N *[B]
3	OOB – out-of-bounds – write overflow on dynamic memory	N	Y	Y
4	OOB – out-of-bounds – write underflow	N	Y	Y
5	OOB – out-of-bounds – read overflow on compile-time memory	N	N	Y
6	OOB – out-of-bounds – read overflow on dynamic memory	N	N	Y
7	OOB – out-of-bounds – read underflow	N	N	Y
8	UAF – use-after-free	Y (Oops/seg faults)	Y (iffy, not reliable)	Y
9	UAR – use-after-return	N	N	N *[C]
10	Double-free	Y	Y	Y
11	(simple) memory leakage	? (insmod hangs)	N	N (use kmemleak)

[1] tested on an x86_64 Ubuntu 18.04.2 LTS (4.15.0-55-generic) VM

[2] tested with KASAN on an x86_64 Fedora 29 (custom 5.0.0 kernel) VM and on a Raspberry Pi 4 ARMv8 running a custom 5.4.70 kernel

*[A] UMR : modern compilers definitely warn regarding this defect, and some do seem to initialize local vars to zero! Also, use [UBSan](#).

*[B] Caught by compiler, as a *Warning* ! (gcc ver in this case):

membugs_kasan/membugs_kasan.c:201:10: **warning: iteration 5 invokes undefined behavior** [-Waggressive-loop-optimizations]

```

201 |   arr[i] = 100; /* Bug: 'arr' overflows on i==5,
    |       ~~~~~^~~~~

```

```

/.../membugs_kasan.c:200:2: note: within this loop
200 |   for (i = 0; i <= 5000; i++) {
    |   ^~~

```

Also **caught by *cppcheck* static analysis**

**[C] : UAR caught by our 'better' Makefile's *sa_cppcheck* static analysis !*

```

... static analysis with cppcheck ...

cppcheck -v --force --enable=all -i .tmp_versions/ -i *.mod.c -i bkp/ --suppress=missingIncludeSystem .
Checking membugs_kasan.c ...
Defines:
Undefines:
Includes:
Platform:Native
membugs_kasan.c:118:9: error: Returning pointer to local variable 'name' that will be invalid when returning. [returnDanglingLifetime]
return name;
^
membugs_kasan.c:118:9: note: Array decayed to pointer here.
return name;
^
membugs_kasan.c:108:7: note: Variable created here.
char name[32];
^
membugs_kasan.c:118:9: note: Returning pointer to local variable 'name' that will be invalid when returning.
return name;
^
membugs_kasan.c:201:6: error: Array 'arr[5]' accessed at index 5000, which is out of bounds. [arrayIndexOutOfBounds]
arr[i] = 100; /* Bug: 'arr' overflows on i==5,
^
membugs_kasan.c:204:8: error: Array 's_arr[5]' accessed at index 5000, which is out of bounds. [arrayIndexOutOfBounds]
s_arr[i] = 200;
^
membugs_kasan.c:132:6: style: Condition 'qs' is always true [knownConditionTrueFalse]
if (qs) {
^
membugs_kasan.c:125:11: note: Assignment 'qs=1', assigned value is 1
int qs = 1;
^
membugs_kasan.c:132:6: note: Condition 'qs' is always true
if (qs) {
^
membugs_kasan.c:285:7: style: Variable 'res' is reassigned a value before the old one has been used. [redundantAssignment]
res = uar();
^
membugs_kasan.c:284:7: note: res is assigned
res = kmalloc(32, GFP_KERNEL);
^
membugs_kasan.c:285:7: note: res is overwritten
res = uar();
^
membugs_kasan.c:213:6: error: Uninitialized variable: x [uninitvar]
if (x)
^
membugs_kasan.c:159:17: style: Unused variable: arr2 [unusedVariable]
int i, arr[5], arr2[7];
^
membugs_kasan.c:192:17: style: Unused variable: arr2 [unusedVariable]
int i, arr[5], arr2[7];
^
membugs_kasan $

```

What this exercise proves / takeaways:

- The vanilla kernel and the `slub_debug=on` kernel command-line option catch only a couple of the 11 common memory bugs seen
- **KASAN** (Kernel Address Sanitizer) directly catches the majority of them ! (7 of 11)
- Among the four that weren't directly caught by KASAN:
 - UMR bug: recent compilers catch / warn on this (upgrade your toolchain?) (see **[A]* above);
 - also caught by the *cppcheck* static analyzer
 - in addition, use **UBSan**
 - OOB write on compile-time memory bug: recent compilers catch / warn on this (upgrade your toolchain?) (see **[B]* above);
 - also caught by the *cppcheck* static analyzer

- UAR bug: caught by our ‘better’ Makefile’s sa_cppcheck target via static analysis ! (see [*\[C\]](#) above)
- memory leak bug: use the [kmemleak](#) tool (covered next).
- **REALLY KEY POINT** : one tool or analysis type cannot and will not catch all possible bugs (*there is no silver bullet!*); **use several**: compiler warnings (-Wall -Wextra -Wpedantic), static and dynamic analyzers.
 - *[Our ‘better’ Makefile](#) tries to enforce exactly this!*

syzbot (syzkaller robot) – a system that “*continuously fuzzes main Linux kernel branches and automatically reports found bugs to kernel mailing lists*” – tests for memory bugs using KASAN and KMSAN:

<https://github.com/google/syzkaller/blob/master/docs/syzbot.md#kmsan-bugs>

<<

Another example of KASAN catching a UAF bug – this vuln was exploited by Alexander Popov (author of the very useful [kconfig-hardened-check script](#)!) to demo a PoC exploit on Ubuntu 18.04 against the V4L2 subsystem:

[CVE-2019-18683: Exploiting a Linux kernel vulnerability in the V4L2 subsystem, A Popov, Feb 2020](#)

...

Deceived V4L2 subsystem

Meanwhile, streaming has fully stopped. The last reference to /dev/video0 is released and the V4L2 subsystem calls vb2_core_queue_release(), which is responsible for freeing up resources. It in turn calls __vb2_queue_free(), which frees our vb2_buffer that was added to the queue when the exploit won the race.

But the driver is not aware of this and still holds the reference to the freed object. When streaming is started again on the next exploit loop, vivid driver touches the freed object that is caught by KASAN:

(see partial screenshot below)

```

=====
BUG: KASAN: use-after-free in vid_cap_buf_queue+0x188/0x1c0
Write of size 8 at addr ffff8880798223a0 by task v4l2-crasher/300

CPU: 1 PID: 300 Comm: v4l2-crasher Tainted: G      W      5.4.0-rc2+ #3
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS ?-20190727_073836-buildvm-ppc64le-16.ppc.fedoraproject.org
Call Trace:
 dump_stack+0x5b/0x90
 print_address_description.constprop.0+0x16/0x200
 ? vid_cap_buf_queue+0x188/0x1c0
 ? vid_cap_buf_queue+0x188/0x1c0
 __kasan_report.cold+0x1a/0x41
 ? vid_cap_buf_queue+0x188/0x1c0
 kasan_report+0xe/0x20
 vid_cap_buf_queue+0x188/0x1c0
 vb2_start_streaming+0x222/0x460
 vb2_core_streamon+0x111/0x240
 __vb2_init_fileio+0x816/0xa30
 __vb2_perform_fileio+0xa88/0x1120
 ? kmsg_dump_rewind_nolock+0xd4/0xd4
 ? vb2_thread_start+0x300/0x300
 ? __mutex_lock_interruptible_slowpath+0x10/0x10
 vb2_fop_read+0x249/0x3e0
 v4l2_read+0x1bf/0x240
 vfs_read+0xf6/0x2d0
 ksys_read+0xe8/0x1c0
 ? kernel_write+0x120/0x120
 ? __ia32_sys_nanosleep_time32+0x1c0/0x1c0
 ? do_user_addr_fault+0x433/0x8d0
 do_syscall_64+0x89/0x2e0
 ? prepare_exit_to_usermode+0xec/0x190
 entry_SYSCALL_64_after_hwframe+0x44/0xa9
RIP: 0033:0x7f3a8ec8222d
Code: c1 20 00 00 75 10 b8 00 00 00 0f 05 48 3d 01 f0 ff ff 73 31 c3 48 83 ec 08 e8 4e fc ff ff 48 89 04 24 b8
RSP: 002b:00007f3a8d0d0e80 EFLAGS: 00000293 ORIG_RAX: 0000000000000000
RAX: ffffffff80000000 RBX: 0000000000000000 RCX: 00007f3a8ec8222d
RDX: 000000000000fffd RSI: 00007f3a8d8d3000 RDI: 0000000000000003

```

...

(In fact, Popov noticed this vuln (vulnerability) only as he used [Google's Syzkaller fuzzer](#) to 'fuzz' the kernel; it produced this UAF, which he then investigated...).

>>

More details on KASAN

Source: LWN, Sept 2014

Finding places where the kernel accesses memory that it shouldn't is the goal for the kernel address sanitizer (KASan). It uses a combination of a new GCC feature to instrument memory accesses and "shadow memory" to track which addresses are legitimate, so that it can complain loudly when the kernel reads or writes anywhere else. While KASan shares some attributes with other kernel debugging features, it has its own advantages—and has already been used to find real kernel bugs.

...

The basic idea behind KASan is to **allocate a map (aka "shadow region") that represents each eight bytes of kernel address space with one byte in the map**. For x86_64 (which is the only architecture supported << recently, on ver 4.4, support begun for ARM64 as well >>), that means setting aside (but not allocating) **16TB of address space to handle the entire 128TB that the kernel can address. Each byte in the map encodes the legality of a kernel access to the corresponding bytes in the full address space.**

The encoding is fairly straightforward. A 0 means that all eight bytes are legal for access, while 1–7

indicate the number of consecutive bytes at the beginning of the eight-byte region that are valid (so 2 means that the first two bytes are valid, the last six are not). Negative values are for different types of non-accessible memory (free memory, redzones of various sorts, etc.).

...

As pages are allocated by the kernel, they are marked as accessible in the shadow region; likewise, as pages are freed, they are marked inaccessible. The **SLUB** allocator has been **modified to update the shadow map** for its allocations and deallocations.

In the patch set's first message, Ryabinin outlined the differences between KASan and a few other kernel memory-debugging tools. Since KASan uses compile-time instrumentation, it is much faster than kmemcheck, but it cannot detect reads of uninitialized memory as kmemcheck does.

While both `DEBUG_PAGEALLOC` and `SLUB_DEBUG` are faster than KASan, neither can detect all of the illegal accesses that KASan does (`DEBUG_PAGEALLOC` only has page-level granularity and `SLUB_DEBUG` is unable to detect some bad reads).

...

In addition, Sasha Levin listed several kernel bugs that he had found using the Trinity fuzzer << *interesting testing framework!* >> with the first version of the KASan patch set. It would seem that there is an unfilled niche in the kernel's memory debugging that KASan could help fill.

Detailed, useful: [*How to use KASAN to debug memory corruption in OpenStack environment*](#)

Also : <https://www.synopsys.com/blogs/software-security/linux-kernel-vulnerabilities/>
(with KASan fuzzing, some previously passing test cases failed!)

New!

From 5.8, we have the really powerful **Kernel Concurrency Sanitizer (KCSAN)**: helps catch concurrency bugs!

<https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html>

The LKD book delves into it!

UBSAN – Undefined Behavior SANitizer

From my LKD book

“The kernel's **Undefined Behavior Sanitizer (UBSAN)** 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. The UB caught by UBSAN includes the following:

- Arithmetic-related UB:
 - Arithmetic overflow / underflow / divide by zero / and so on...
 - OOB accesses while bit shifting
- Memory-related UB:
 - OOB accesses on arrays
 - NULL pointer dereferences
 - Misaligned memory accesses
 - Object size mismatches”

Ref:

<https://docs.kernel.org/dev-tools/ubsan.html>

“**Clang** currently seems to have a significant advantage over GCC – especially from our point of view – generating superior diagnostics as well as being able to intelligently generate code avoiding OOB accesses. This is critical. It paves the way to superior code. We saw (in the previous section on KASAN) that faulty left-OOB accesses on global memory, not reliably caught by GCC (versions 9.3, 10, and 11), are caught with Clang! The Android project is a key user of Clang, among many others.”

To employ clang with kernel modules, you need to build **both** the kernel and your modules with it:

kernel:

```
make -j8 CC=clang
```

modules:

```
make CC=clang
```

From my LKD book

KFENCE – Kernel Electric-Fence

- 5.12 onwards
- “low-overhead sampling-based memory safety error detector of heap use-after-free, invalid-

- free, and out-of-bounds access errors.”
- KFENCE has been designed **for use in production systems**; KASAN's overhead would be too high for typical production systems and is suitable only on debug / development systems. KFENCE's performance overhead is minimal – close to zero.
 - KFENCE works on a sampling-based design. It trades precision for performance, thus, with sufficiently lengthy uptime, KFENCE is almost certain to catch bugs! One way to have a **really long total uptime** is by deploying it across a fleet of machines.
 - In effect, KASAN will catch all memory defects, but at a rather high performance cost. KFENCE also can catch all memory defects, at virtually no performance cost, but it takes time (very long uptimes are required, as it's a sampling-based approach). Thus, to catch memory defects on debug and development systems, use KASAN (and KFENCE, perhaps); to do the same on production systems, use KFENCE.

Ref:

<https://www.kernel.org/doc/html/latest/dev-tools/kfence.html#kernel-electric-fence-kfence>

SLUB Debug

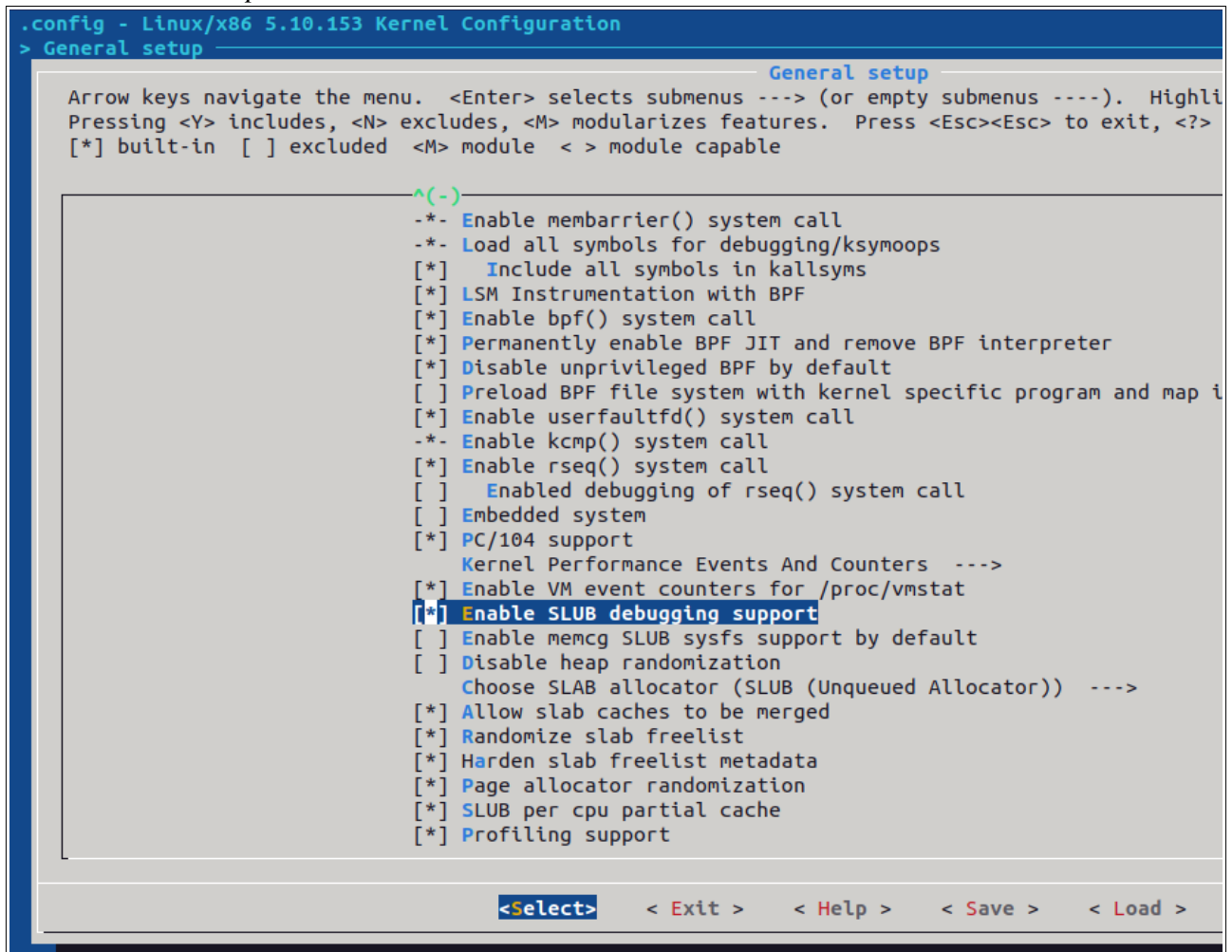
Very useful: *Short users guide for SLUB*

<https://www.kernel.org/doc/Documentation/vm/slub.txt>

These techniques work ONLY on SLUB implementation.

Kernel Hacking / Memory Debugging

Under *General Setup*:



```
.config - Linux/x86 5.10.153 Kernel Configuration
> General setup

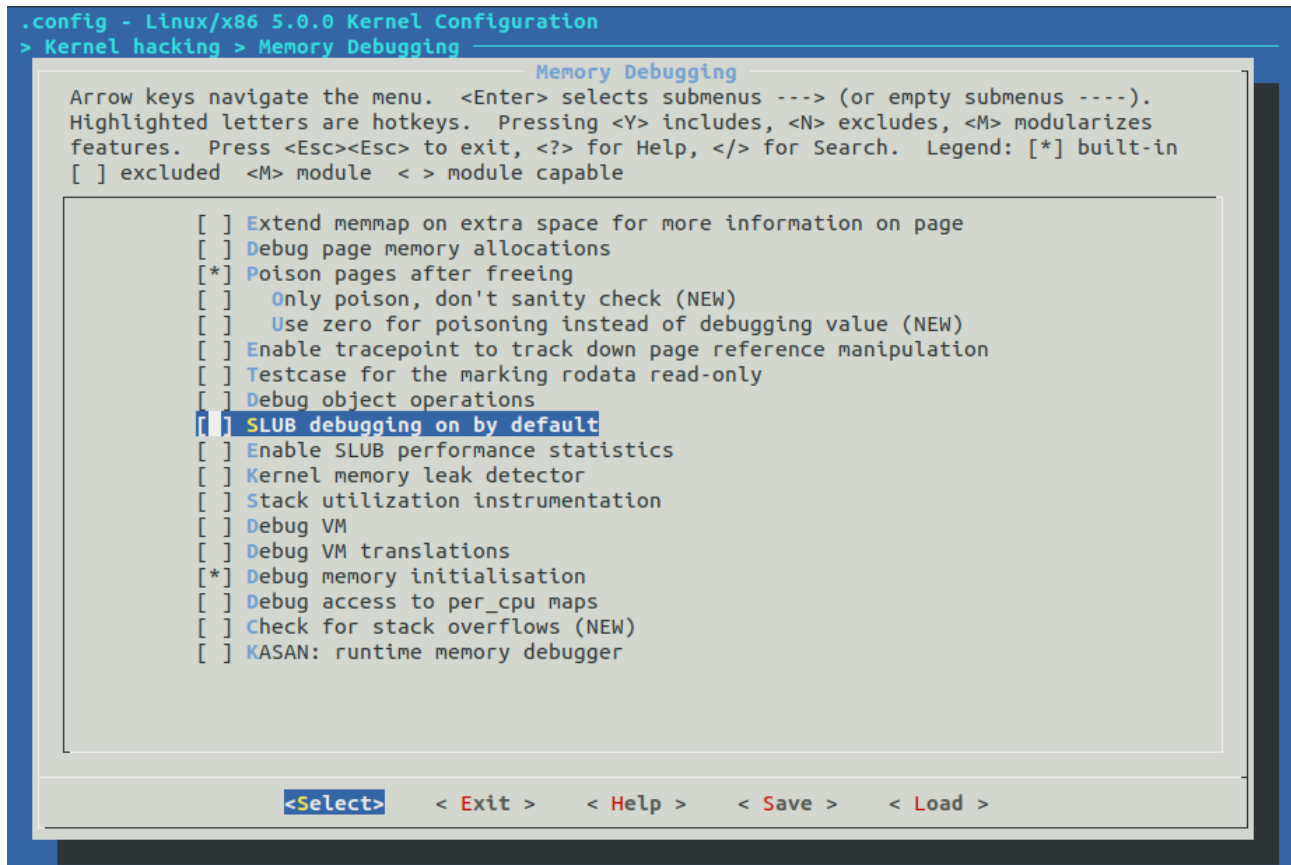
                                General setup
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highli
Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?>
[*] built-in [ ] excluded <M> module < > module capable

^(-)
-- Enable membarrier() system call
-- Load all symbols for debugging/ksymoops
[*] Include all symbols in kallsyms
[*] LSM Instrumentation with BPF
[*] Enable bpf() system call
[*] Permanently enable BPF JIT and remove BPF interpreter
[*] Disable unprivileged BPF by default
[ ] Preload BPF file system with kernel specific program and map i
[*] Enable userfaultfd() system call
-- Enable kcmp() system call
[*] Enable rseq() system call
[ ] Enabled debugging of rseq() system call
[ ] Embedded system
[*] PC/104 support
    Kernel Performance Events And Counters --->
[*] Enable VM event counters for /proc/vmstat
[*] Enable SLUB debugging support
[ ] Enable memcg SLUB sysfs support by default
[ ] Disable heap randomization
    Choose SLAB allocator (SLUB (Unqueued Allocator)) --->
[*] Allow slab caches to be merged
[*] Randomize slab freelist
[*] Harden slab freelist metadata
[*] Page allocator randomization
[*] SLUB per cpu partial cache
[*] Profiling support

<Select>  < Exit >  < Help >  < Save >  < Load >
```

Leave SLUB debugging ‘off’ by default.

Under *Kernel Hacking | Memory Debugging*



```

$ grep SLUB_DEBUG /boot/config-5.15.0-53-generic
CONFIG_SLUB_DEBUG=y
# CONFIG_SLUB_DEBUG_ON is not set

```

Leave the second option – `CONFIG_SLUB_DEBUG_ON` - off... only enable it when required via the `slub_debug=` kernel parameter.

Poison Pages after Freeing

CONFIG_PAGE_POISONING:

Fill the pages with poison patterns after `free_pages()` and verify the patterns before `alloc_pages`. The filling of the memory helps reduce the risk of information leaks from freed data. This does have a potential performance impact if enabled with the "`page_poison=1`" kernel boot option.

From my LKD book***“Understanding the SLUB layer's poison flags***

The poison flags defined by the kernel are defined as follows:

```
// include/linux/poison.h
#define POISON_INUSE 0x5a /* for use-uninitialised poisoning */
#define POISON_FREE 0x6b /* for use-after-free poisoning */
#define POISON_END 0xa5 /* end-byte of poisoning */
```

Here's the nitty-gritty on these poison values:

- When you use the SLAB_POISON flag when creating a slab cache (typically via the `kmem_cache_create()` kernel API) or set poisoning to on via the kernel parameter `slub_debug=P`, the slab memory gets auto-initialized to the value `0x6b` (which is ASCII `k`, corresponding to the `POISON_FREE` macro). In effect, when this flag is enabled, this (`0x6b`) is the value that *valid but uninitialized slab memory regions are set to on creation*.
- The `POISON_INUSE` value (`0x5a` equals ASCII `Z`) is used to denote padding zones, before or after red zones.
- The *last legal byte* of the slab memory object is set to `POISON_END`, `0xa5`.

A test:

- boot with kernel cmdline containing:
`slub_debug=FZ`

```
Sanity checks (F)
Red zoning (Z)
```

- *Run the UAF – Use After Free - test case:*

Code:

```
git clone https://github.com/PacktPublishing/Linux-Kernel-Programming
cd Linux-Kernel-Programming/ch9/poison_test
...
static void __exit slab_custom_exit(void)
{
    kmem_cache_free(gctx_cachep, obj);
    use_the_object(obj, '!', 10); /* the (here pretty obvious) UAF BUG ! */
    [ ... ]
}
```

(FYI, the value `'!'` is `0x21`).

```
make; sudo insmod ./poison_test.ko ; dmesg
...
```

```
sudo rmmod poison_test
<UAF bug hit!>
```

dmesg

...

```

[ 52.742529] obj: 000000009896f0b3: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkkk
[ 52.745635] obj: 000000007990bbc3: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkkk
[ 52.748802] obj: 000000008b10e5c3: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkkk
[ 52.751956] obj: 00000000b919497c: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkkk
[ 52.755143] obj: 00000000c7f9241c: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkkk
[ 52.758476] obj: 00000000aca65f9a: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkkk
[ 52.761774] obj: 000000004664c253: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b a5 kkkkkkkk.
[ 52.787491] BUG poison_test (Tainted: G      OE      ): Poison overwritten
[ 52.792008] -----
[ 52.795327] Disabling lock debugging due to kernel taint
[ 52.797088] INFO: 0x0000000a34781fc-0x0000000e51816cf. First byte 0x21 instead of 0x6b
[ 52.799226] INFO: Slab 0x000000001893d3f8 objects=16 used=0 fp=0x00000000a34781fc flags=0xfffffc00000200
[ 52.802480] INFO: Object 0x00000000a34781fc @offset=64 fp=0x00000000f46c4630

[ 52.805603] Redzone 000000006d2c6854: bb bb bb bb bb bb bb bb bb bb bb bb bb bb bb bb .....
[ 52.808846] Redzone 00000000b29f4ef7: bb bb bb bb bb bb bb bb bb bb bb bb bb bb bb bb .....
[ 52.812055] Redzone 00000000ac16cfea: bb bb bb bb bb bb bb bb bb bb bb bb bb bb bb bb .....
[ 52.815304] Redzone 0000000077fb7926: bb bb bb bb bb bb bb bb bb bb bb bb bb bb bb bb .....
[ 52.818602] Object 00000000a34781fc: 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 kkkkkkkk
[ 52.822355] Object 00000000971f1782: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkkk
[ 52.825596] Object 000000009278e4b: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkkk
[ 52.828838] Object 000000009896f0b3: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkkk
[ 52.832065] Object 000000007990bbc3: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkkk
[ 52.835232] Object 000000008b10e5c3: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkkk
[ 52.838456] Object 00000000b919497c: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkkk
[ 52.841679] Object 00000000c7f9241c: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkkk
[ 52.844914] Object 00000000aca65f9a: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkkk
[ 52.848125] Object 000000004664c253: 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b a5 kkkkkkkk
[ 52.850304] Redzone 00000000e532a72: bb bb bb bb bb bb bb bb .....
[ 52.852516] Padding 000000005185a36: 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a zzzzzzzzzzzzzzzz
[ 52.855701] Padding 00000000d5d356a4: 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a zzzzzzzz
[ 52.857868] CPU: 3 PID: 1265 Comm: rmmod Tainted: G      B      OE      5.4.0-99-generic #112-Ubuntu
[ 52.860063] Hardware name: innotek GmbH VirtualBox/VirtualBox, BIOS VirtualBox 12/01/2006
[ 52.862162] Call Trace:
[ 52.863469] dump_stack+0x6d/0x8b
[ 52.865054] print_trailer+0x1d8/0x1e5
[ 52.866561] check_bytes_and_report.cold+0x4c/0x68
[ 52.868153] check_object+0x232/0x280
[ 52.869650] __free_slab+0xab/0x2e0
[ 52.871011] discard_slab+0x38/0x50
[ 52.872355] __kmem_cache_shutdown+0x19b/0x240
[ 52.873802] shutdown_cache+0x16/0x160
[ 52.875127] kmem_cache_destroy+0x24a/0x290
[ 52.876545] slab_custom_exit+0x39/0xf06 [poison_test]
[ 52.878051] x64_sys_delete_module+0x147/0x2b0
[ 52.879493] ? exit_to_usermode_loop+0xea/0x160
[ 52.880887] do_syscall_64+0x57/0x190

```

Red zone of free object: 0xbb

Illegal: wrote 0x21 (!) after freeing - UAF

Legal memory 0x6b ('k')

Last legal bytes: 0xa5

Padding zone b/w objects: 0x5a

...

Regarding the 0xbb values seen above [\[link\]](#) :

“... These RED zones are filled with markers to indicate the allocation state of an object. The RED zones of the allocated objects are filled with the value 0xcc (i.e SLUB_RED_ACTIVE) and the RED zones of the free objects are filled with the value 0xbb (i.e SLUB_RED_INACTIVE). An out of bound access can change these marker values. The next memory operation on this SLUB object checks and finds that RED zone markers have changed and this in turn means that an out of bound (OOB) access has happened. ...”

From my LKD book:

240 Debugging Kernel Memory Issues – Part 2

Then run the relevant test cases via the `test_kmembugs.ko` kernel module and associated `run_tests` script for each of these scenarios. The following table summarizes the results.

Test case #	Memory defect type (below) / Infrastructure used (right)	Production kernel with slub_debug=- (off)	Production kernel with slub_debug=FZPU
5	OOB accesses on dynamic kmalloc-ed slab (SLUB) memory		
5.1	Read (right) overflow	N [V1]	N
5.2	Write (right) overflow		Y [V4]
5.3	Read (left) underflow		N
5.4	Write (left) underflow		Y [V4]
Other memory corruption test cases			
6	Use After Free – UAF	N	Y [V5]
7	Double-free	N [V2]	Y [V6]
9	OOB on copy_[to from]_user*()	N	N
10	Uninitialized Memory Read – UMR – on slab (SLUB) memory	N [V3]	N [V7]

Table 6.2 – Summary of findings when running relevant memory defect test cases against both our production kernel without `slub_debug` features and with `slub_debug=FZPU`

...

Tips-

Lookup the discussion in my LKD book, *Ch 6 – Debugging Kernel Memory Issues, Part 2* section

- ‘Learning how to use the `slabinfo` and related utilities’!
 - Also try `slabratetop-bpfcc`
- ‘Catching memory defects in the kernel – comparisons and notes (Part 2)’ – final comparison table.

Good blog article: [Linux SLUB Allocator Internals and Debugging - SLUB Debugger, Part 2 of 4, Imran Khan, Dec 2022, Oracle Linux.](#)

Kmemleak

Documentation/kmemleak.txt

<https://www.kernel.org/doc/Documentation/kmemleak.txt>

“Kmemleak provides a way of detecting possible kernel memory leaks in a way similar to a tracing garbage collector (https://en.wikipedia.org/wiki/Garbage_collection_%28computer_science%29#Tracing_garbage_collectors), with the difference that the orphan objects are not freed but only reported via `/sys/kernel/debug/kmemleak`. A similar method is used by the Valgrind tool (`memcheck --leak-check`) to detect the memory leaks in user-space applications.

Kmemleak is supported on x86, arm, powerpc, sparc, sh, microblaze, ppc, mips, s390, metag and tile.

Usage

CONFIG_DEBUG_KMEMLEAK in "*Kernel hacking | Memory Debugging | Kernel memory leak detector*" has to be enabled.

NOTE- you have to **explicitly enable it** by passing **kmemleak=on** on the kernel command line.

Once enabled, a kernel thread scans the memory every 10 minutes (by default) and prints the number of new unreferenced objects found. To display the details of all the possible memory leaks:

```
# mount -t debugfs nodev /sys/kernel/debug/
# cat /sys/kernel/debug/kmemleak
```

To trigger an intermediate memory scan:

```
# echo scan > /sys/kernel/debug/kmemleak
```

To clear the list of all current possible memory leaks:

```
# echo clear > /sys/kernel/debug/kmemleak
```

New leaks will then come up upon reading `/sys/kernel/debug/kmemleak` again.

...”

- interface: `/sys/kernel/debug/kmemleak`

- page / BSA allocations (`get_free_page()` / `alloc_page()` and friends) and `ioremap()` are **not** tracked

“ ...

Testing specific sections with kmemleak

Upon initial bootup your `/sys/kernel/debug/kmemleak` output page may be quite extensive. This can also be the case if you have very buggy code when doing development. To work around these situations you can use the 'clear' command to clear all reported unreferenced objects from the `/sys/kernel/debug/kmemleak` output. By issuing a 'scan' after a 'clear' you can find new unreferenced objects; this should help with testing specific sections of code.

To test a critical section on demand with a clean kmemleak do:

```
# echo clear > /sys/kernel/debug/kmemleak
... test your kernel or modules ...
# echo scan > /sys/kernel/debug/kmemleak
```

Then as usual to get your report with:

```
# cat /sys/kernel/debug/kmemleak
```

Freeing kmemleak internal objects

To allow access to previously found memory leaks after kmemleak has been disabled by the user or due to an fatal error, internal kmemleak objects won't be freed when kmemleak is disabled, and those objects may occupy a large part of physical memory.

In this situation, you may reclaim memory with:

```
# echo clear > /sys/kernel/debug/kmemleak
..."
```

syzbot (aka syzkaller) – a system that “*continuously fuzzes main Linux kernel branches and automatically reports found bugs to kernel mailing lists*” – tests for memory leaks using kmemleak: <https://github.com/google/syzkaller/blob/master/docs/syzbot.md#memory-leaks>

Test Case for kmemleak

Run the test script `run_kmleak.sh`

```
# ./run_kmleak.sh
make -C /lib/modules/3.18.22/build
M=/home/seawolf/kaiwanTECH/0prg_testing/kmemleak_kmemcheck_test modules
make[1]: Entering directory '/opt/3.18.22-x86-build/linux-3.18.22'
```

```

Building for ARCH x86 and KERNELRELEASE 3.18.22
Building modules, stage 2.
Building for ARCH x86 and KERNELRELEASE 3.18.22
MODPOST 1 modules
make[1]: Leaving directory '/opt/3.18.22-x86-build/linux-3.18.22'
Running test case now ...
Delaying for 100 seconds now, pl wait ...
Kmemleak report:
-----
#

# cat /sys/kernel/debug/kmemleak
#

```

Nothing yet! Wait a few minutes and retry ...

```

# echo scan > /sys/kernel/debug/kmemleak
# cat /sys/kernel/debug/kmemleak
unreferenced object 0xfffffc900002c3000 (size 20480):
  comm "insmod", pid 4667, jiffies 4297307396 (age 116.420s)
  hex dump (first 32 bytes):
    01 00 00 00 00 00 00 00 ff ff ff ff ff ff ff ff .....
    e2 00 00 00 04 00 00 00 40 00 00 00 00 00 00 00 .....@.....
  backtrace:
    [<fffffffff81704013>] kmemleak_alloc+0x23/0x50
    [<fffffffff8110f73e>] __vmalloc_node_range+0x20e/0x2b0
    [<fffffffff8110f866>] vmalloc+0x46/0x50
    [<fffffffffa009a025>] 0xfffffffffa009a025
    [<fffffffff810002c4>] do_one_initcall+0x84/0x1c0
    [<fffffffff810b17de>] load_module+0x1bee/0x2210
    [<fffffffff810b1f6e>] SyS_finit_module+0x8e/0xa0
    [<fffffffff8170e149>] system_call_fastpath+0x12/0x17
    [<ffffffffffffffff>] 0xffffffffffffffff
# dmesg |grep "fffffc900002c3000"
kern :info : [ +0.000000] @@@ vmalloc_test: vmalloc(20480):
vbuf=fffffc900002c3000 pa=00004100002c3000
kern :debug : [ +0.000000] @@@ fffffc900002c3000: 01 00 00 00 00 00 00 00 ff ff
ff ff ff ff ff ff .....
#

```

Verify

```

# echo "dump=0xfffffc900002c3000" > /sys/kernel/debug/kmemleak
# dmesg
kern :notice: [Sep10 14:12] kmemleak: Object 0xfffffc900002c3000 (size 20480):
kern :notice: [ +0.000009] kmemleak: comm "insmod", pid 4667, jiffies
4297307396
kern :notice: [ +0.000004] kmemleak: min_count = 0
kern :notice: [ +0.000001] kmemleak: count = 0
kern :notice: [ +0.000002] kmemleak: flags = 0x3
kern :notice: [ +0.000002] kmemleak: checksum = 1067443308
kern :notice: [ +0.000002] kmemleak: backtrace:
kern :warn : [ +0.000002] [<fffffffff81704013>] kmemleak_alloc+0x23/0x50
kern :warn : [ +0.000027] [<fffffffff8110f73e>]
__vmalloc_node_range+0x20e/0x2b0
kern :warn : [ +0.000017] [<fffffffff8110f866>] vmalloc+0x46/0x50
kern :warn : [ +0.000004] [<fffffffffa009a025>] 0xfffffffffa009a025

```

```

kern :warn : [ +0.000022]      [<ffffffff810002c4>]
do_one_initcall+0x84/0x1c0
kern :warn : [ +0.000005]      [<ffffffff810b17de>] load_module+0x1bee/0x2210
kern :warn : [ +0.000004]      [<ffffffff810b1f6e>]
SyS_finit_module+0x8e/0xa0
kern :warn : [ +0.000004]      [<ffffffff8170e149>]
system_call_fastpath+0x12/0x17
kern :warn : [ +0.000005]      [<ffffffffffffffff>] 0xffffffffffffffff
#

```

NOTE! kmemcheck has been removed from 4.15

Kmemcheck

"kmemcheck is a debugging feature for the Linux Kernel. More specifically, it is a dynamic checker that detects and warns about some uses of uninitialized memory."

- A bit of a Valgrind-like memcheck for Linux kernel space.
- Works only in kernel-space
- x86 and x86_64 only (?)
- from 2.6.31-rc1

1. First read this short and succinct LWN article: "[kmemcheck](#)", Nov 2007

2. *Documentation/kmemcheck.txt*

<https://www.kernel.org/doc/Documentation/kmemcheck.txt>

-or- (newer)

<https://www.kernel.org/doc/Documentation/dev-tools/kmemcheck.txt>

Contains details on configuring the Linux kernel for kmemcheck – including which options must be switched OFF! - and actual usage.

The above LWN article “[kmemcheck](#)” is reproduced below:

“Using uninitialized memory can lead to some seriously annoying bugs. If you are lucky, the kernel will crash with the telltale slab poisoning pattern (0x5a5a5a5a or similar) in the traceback. Other times, though, something more subtly wrong happens, forcing a long hunt for the stupid mistake. Wouldn't it be nicer if the kernel could simply detect references to uninitialized memory and scream loudly at the time?

The kmemcheck patch recently posted by Vegard Nossum offers just that functionality, though, perhaps, in a somewhat heavy-handed manner. A kernel with kmemcheck enabled is unlikely to be suitable for production use, but it should, indeed, do a good job at finding code using memory which

has not yet been set to a useful value.

Kmemcheck is a relatively simple patch; the approach used is, essentially, this:

- Every memory allocation is trapped at the page-allocator level. For each allocation, the requested order is increased by one, doubling the size of the allocation. The additional ("shadow") pages are initialized to zero and kept hidden.
- The allocated memory is returned to the caller, but with the "present" bit cleared in the page tables. As a result, every attempt to access that memory will cause a page fault.
- Once the fault happens, kmemcheck (through some ugly, architecture-specific code) determines the exact address and size of the attempted access. If the access is a write, the corresponding bytes in the shadow page are set to 0xff and the operation is allowed to complete.
- For read accesses, the corresponding shadow page bytes are tested; if any of them are zero, the code concludes that the read is trying to access uninitialized data. A stack traceback is printed to enable the developer to find the location where this access is happening.

As should be evident, running with kmemcheck enabled will have certain performance impacts. Taking a page fault on every access to slab memory just cannot be fast. Doubling the size of every allocation will impose costs of its own, including the cache effects of simply working with twice as much memory. But that is a cost which can be paid when the kernel is being run in a debugging mode.

Vegard has posted some sample output which shows how the system responds to reads from uninitialized memory. If this output is to be believed, access to unset memory is not an especially uncommon occurrence in current kernels. If some of references flagged here, once tracked down, turn out to be real bugs, the kmemcheck patch will have earned its keep, even if it never finds its way into the mainline.”

The Documentation clearly mentions that several kernel configurables need to be **turned OFF for kmemcheck to function correctly**. Among them is ftrace:

...

```
CONFIG_FUNCTION_TRACER=n
```

This option is located under "Kernel hacking" / "Tracers" / "Kernel Function Tracer"

When function tracing is compiled in, gcc emits a call to another function at the beginning of every function. This means that when the page fault handler is called, the ftrace framework will be called before kmemcheck has had a chance to handle the fault. If ftrace then

modifies memory that was tracked by kmemcheck, the result is an endless recursive page fault.

...

So, we do an experiment: we attempt configuring the kernel (make menuconfig); by default, ftrace functionality is On. **So we do not even see the kmemcheck configurable under “Kernel Hacking” / “Memory Debugging” by default!**

Now, turn Off Ftrace and retry:

```
[ ] Fault-injection framework
[ ] Latency measuring infrastructure
[ ] Strict user copy size checks
[ ] Tracers -----
Runtime Testing --->
```

```
[ ] Debug page memory allocations
[ ] Debug object operations
[ ] SLUB debugging on by default
[ ] Enable SLUB performance statistics
[ ] Kernel memory leak detector
[ ] Stack utilization instrumentation
[ ] Debug VM
[ ] Debug VM translations
[ ] Debug access to per_cpu maps
[*] Check for stack overflows
[ ] kmemcheck: trap use of uninitialized memory -----
```

Clearly, the last option is *kmemcheck*! Turn it On and proceed with the config, save and build ...

kaiwanTECH Linux OS Corporate Training Programs

Please do check out our current offering of world-class, seriously-valuable, high on returns, technical Linux OS corporate training programs [here](#).