The Kernel Address Sanitizer (KASAN)

Overview

KernelAddressSANitizer (KASAN) is a dynamic memory safety error detector designed to find out-of-bound and use-after-free bugs. KASAN has three modes:

1. generic KASAN (similar to userspace ASan),
2. software tag-based KASAN (similar to userspace HWASan),
3. hardware tag-based KASAN (based on hardware memory tagging).

Generic KASAN is mainly used for debugging due to a large memory overhead. Software tag-based KASAN can be used for dogfood testing as it has a lower memory overhead that allows using it with real workloads. Hardware tag-based KASAN comes with low memory and performance overheads and, therefore, can be used in production. Either as an in-field memory bug detector or as a security mitigation.

Software KASAN modes (#1 and #2) use compile-time instrumentation to insert validity checks before every memory access and, therefore, require a compiler version that supports that.

Generic KASAN is supported in GCC and Clang. With GCC, it requires version 8.3.0 or later. Any supported Clang version is compatible, but detection of out-of-bounds accesses for global variables is only supported since Clang 11.

Software tag-based KASAN mode is only supported in Clang.

The hardware KASAN mode (#3) relies on hardware to perform the checks but still requires a compiler version that supports memory tagging instructions. This mode is supported in GCC 10+ and Clang 12+.

Both software KASAN modes work with SLUB and SLAB memory allocators, while the hardware tag-based KASAN currently only supports SLUB.

Currently, generic KASAN is supported for the x86_64, arm, arm64, xtensa, s390, and riscv architectures, and tag-based KASAN modes are supported only for arm64.

Usage

To enable KASAN, configure the kernel with:

```
CONFIG_KASAN=y
```

and choose between `CONFIG_KASAN_GENERIC` (to enable generic KASAN), `CONFIG_KASAN_SW_TAGS` (to enable software tag-based KASAN), and `CONFIG_KASAN_HW_TAGS` (to enable hardware tag-based KASAN).

For software modes, also choose between `CONFIG_KASAN_OUTLINE` and `CONFIG_KASAN_INLINE`. Outline and inline are compiler instrumentation types. The former produces a smaller binary while the latter is 1.1-2 times faster.

To include alloc and free stack traces of affected slab objects into reports, enable `CONFIG_STACKTRACE`. To include alloc and free stack traces of affected physical pages, enable `CONFIG_PAGE_OWNER` and boot with `page_owner=on`.

Error reports

A typical KASAN report looks like this:

```
==================================================================
BUG: KASAN: slab-out-of-bounds in kmalloc_oob_right+0xa8/0xbc [test_kasan]
Write of size 1 at addr ffff8801f44ec37b by task insmod/2760

CPU: 1 PID: 2760 Comm: insmod Not tainted 4.19.0-rc3+ #698
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.10.2-1 04/01/2014
Call Trace:
 dump_stack+0x94/0xd8
 print_address_description+0x73/0x280
 kasan_report+0x144/0x187
 __asan_report_store1_noabort+0x17/0x20
 kmalloc_oob_right+0xa8/0xbc [test_kasan]
 kmalloc_tests_init+0x16/0x700 [test_kasan]
 do_one_initcall+0xa5/0x3ae
 do_init_module+0x1b6/0x547
 load_module+0x75df/0x8070
 __do_sys_init_module+0x1c6/0x200
 __x64_sys_init_module+0x6e/0xb0
 do_syscall_64+0x9f/0x2c0
 entry_SYSCALL_64_after_hwframe+0x44/0xa9
RIP: 0033:0x7f96443109da
RSP: 002b:00007ffcf0b51b08 EFLAGS: 00000202 ORIG_RAX: 00000000000000af
RAX: ffffffffffffffda RBX: 000055dc3ee521a0 RCX: 00007f96443109da
```

```
RDX: 00007f96445cff88 RSI: 0000000000057a50 RDI: 00007f9644992000
RBP: 000055dc3ee510b0 R08: 0000000000000003 R09: 0000000000000000
R10: 00007f964430cd0a R11: 0000000000000202 R12: 00007f96445cff88
R13: 000055dc3ee51090 R14: 0000000000000000 R15: 0000000000000000

Allocated by task 2760:
 save_stack+0x43/0xd0
 kasan_kmalloc+0xa7/0xd0
 kmem_cache_alloc_trace+0xe1/0x1b0
 kmalloc_oob_right+0x56/0xbc [test_kasan]
 kmalloc_tests_init+0x16/0x700 [test_kasan]
 do_one_initcall+0xa5/0x3ae
 do_init_module+0x1b6/0x547
 load_module+0x75df/0x8070
 __do_sys_init_module+0x1c6/0x200
 __x64_sys_init_module+0x6e/0xb0
 do_syscall_64+0x9f/0x2c0
 entry_SYSCALL_64_after_hwframe+0x44/0xa9

Freed by task 815:
 save_stack+0x43/0xd0
 __kasan_slab_free+0x135/0x190
 kasan_slab_free+0xe/0x10
 kfree+0x93/0x1a0
 umh_complete+0x6a/0xa0
 call_usermodehelper_exec_async+0x4c3/0x640
 ret_from_fork+0x35/0x40

The buggy address belongs to the object at ffff8801f44ec300
 which belongs to the cache kmalloc-128 of size 128
The buggy address is located 123 bytes inside of
 128-byte region [ffff8801f44ec300, ffff8801f44ec380)
The buggy address belongs to the page:
page:ffffea0007d13b00 count:1 mapcount:0 mapping:ffff8801f7001640 index:0x0
flags: 0x200000000000100(slab)
raw: 0200000000000100 ffffea0007d11dc0 0000001a0000001a ffff8801f7001640
raw: 0000000000000000 0000000080150015 00000001ffffffff 0000000000000000
page dumped because: kasan: bad access detected

Memory state around the buggy address:
 ffff8801f44ec200: fc fc fc fc fc fc fc fc fb fb fb fb fb fb fb fb
 ffff8801f44ec280: fb fb fb fb fb fb fb fb fc fc fc fc fc fc fc fc
>ffff8801f44ec300: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03
                                                              ^
 ffff8801f44ec380: fc fc fc fc fc fc fc fc fb fb fb fb fb fb fb fb
 ffff8801f44ec400: fb fb fb fb fb fb fb fb fc fc fc fc fc fc fc fc
==================================================================
```

The report header summarizes what kind of bug happened and what kind of access caused it. It is followed by a stack trace of the bad access, a stack trace of where the accessed memory was allocated (in case a slab object was accessed), and a stack trace of where the object was freed (in case of a use-after-free bug report). Next comes a description of the accessed slab object and the information about the accessed memory page.

In the end, the report shows the memory state around the accessed address. Internally, KASAN tracks memory state separately for each memory granule, which is either 8 or 16 aligned bytes depending on KASAN mode. Each number in the memory state section of the report shows the state of one of the memory granules that surround the accessed address.

For generic KASAN, the size of each memory granule is 8. The state of each granule is encoded in one shadow byte. Those 8 bytes can be accessible, partially accessible, freed, or be a part of a redzone. KASAN uses the following encoding for each shadow byte: 00 means that all 8 bytes of the corresponding memory region are accessible; number N (1 <= N <= 7) means that the first N bytes are accessible, and other (8 - N) bytes are not; any negative value indicates that the entire 8-byte word is inaccessible. KASAN uses different negative values to distinguish between different kinds of inaccessible memory like redzones or freed memory (see mm/kasan/kasan.h).

In the report above, the arrow points to the shadow byte `03`, which means that the accessed address is partially accessible.

For tag-based KASAN modes, this last report section shows the memory tags around the accessed address (see the Implementation details section).

Note that KASAN bug titles (like `slab-out-of-bounds` or `use-after-free`) are best-effort: KASAN prints the most probable bug type based on the limited information it has. The actual type of the bug might be different.

Generic KASAN also reports up to two auxiliary call stack traces. These stack traces point to places in code that interacted with the object but that are not directly present in the bad access stack trace. Currently, this includes call_rcu() and workqueue queuing.

Boot parameters

KASAN is affected by the generic `panic_on_warn` command line parameter. When it is enabled, KASAN panics the kernel after printing a bug report.

By default, KASAN prints a bug report only for the first invalid memory access. With `kasan_multi_shot`, KASAN prints a report on every invalid access. This effectively disables `panic_on_warn` for KASAN reports.

Alternatively, independent of `panic_on_warn` the `kasan.fault=` boot parameter can be used to control panic and reporting behaviour:

- `kasan.fault=report` or `=panic` controls whether to only print a KASAN report or also panic the kernel (default: `report`). The panic happens even if `kasan_multi_shot` is enabled.

Hardware tag-based KASAN mode (see the section about various modes below) is intended for use in production as a security mitigation. Therefore, it supports additional boot parameters that allow disabling KASAN or controlling features:

- `kasan=off` or `=on` controls whether KASAN is enabled (default: `on`).
- `kasan.mode=sync`, `=async` or `=asymm` controls whether KASAN is configured in synchronous, asynchronous or asymmetric mode of execution (default: `sync`). Synchronous mode: a bad access is detected immediately when a tag check fault occurs. Asynchronous mode: a bad access detection is delayed. When a tag check fault occurs, the information is stored in hardware (in the TFSR_EL1 register for arm64). The kernel periodically checks the hardware and only reports tag faults during these checks. Asymmetric mode: a bad access is detected synchronously on reads and asynchronously on writes.
- `kasan.vmalloc=off` or `=on` disables or enables tagging of vmalloc allocations (default: `on`).
- `kasan.stacktrace=off` or `=on` disables or enables alloc and free stack traces collection (default: `on`).

Implementation details

Generic KASAN

Software KASAN modes use shadow memory to record whether each byte of memory is safe to access and use compile-time instrumentation to insert shadow memory checks before each memory access.

Generic KASAN dedicates 1/8th of kernel memory to its shadow memory (16TB to cover 128TB on x86_64) and uses direct mapping with a scale and offset to translate a memory address to its corresponding shadow address.

Here is the function which translates an address to its corresponding shadow address:

```
static inline void *kasan_mem_to_shadow(const void *addr)
{
    return (void *)((unsigned long)addr >> KASAN_SHADOW_SCALE_SHIFT)
            + KASAN_SHADOW_OFFSET;
}
```

where `KASAN_SHADOW_SCALE_SHIFT = 3`.

Compile-time instrumentation is used to insert memory access checks. Compiler inserts function calls (`__asan_load*(addr)`, `__asan_store*(addr)`) before each memory access of size 1, 2, 4, 8, or 16. These functions check whether memory accesses are valid or not by checking corresponding shadow memory.

With inline instrumentation, instead of making function calls, the compiler directly inserts the code to check shadow memory. This option significantly enlarges the kernel, but it gives an x1.1-x2 performance boost over the outline-instrumented kernel.

Generic KASAN is the only mode that delays the reuse of freed objects via quarantine (see mm/kasan/quarantine.c for implementation).

Software tag-based KASAN

Software tag-based KASAN uses a software memory tagging approach to checking access validity. It is currently only implemented for the arm64 architecture.

Software tag-based KASAN uses the Top Byte Ignore (TBI) feature of arm64 CPUs to store a pointer tag in the top byte of kernel pointers. It uses shadow memory to store memory tags associated with each 16-byte memory cell (therefore, it dedicates 1/16th of the kernel memory for shadow memory).

On each memory allocation, software tag-based KASAN generates a random tag, tags the allocated memory with this tag, and embeds the same tag into the returned pointer.

Software tag-based KASAN uses compile-time instrumentation to insert checks before each memory access. These checks make sure that the tag of the memory that is being accessed is equal to the tag of the pointer that is used to access this memory. In case of a tag mismatch, software tag-based KASAN prints a bug report.

Software tag-based KASAN also has two instrumentation modes (outline, which emits callbacks to check memory accesses; and inline, which performs the shadow memory checks inline). With outline instrumentation mode, a bug report is printed from the function that performs the access check. With inline instrumentation, a `brk` instruction is emitted by the compiler, and a dedicated `brk` handler is used to print bug reports.

Software tag-based KASAN uses 0xFF as a match-all pointer tag (accesses through pointers with the 0xFF pointer tag are not checked). The value 0xFE is currently reserved to tag freed memory regions.

Software tag-based KASAN currently only supports tagging of slab, page_alloc, and vmalloc memory.

Hardware tag-based KASAN

Hardware tag-based KASAN is similar to the software mode in concept but uses hardware memory tagging support instead of compiler instrumentation and shadow memory.

Hardware tag-based KASAN is currently only implemented for arm64 architecture and based on both arm64 Memory Tagging Extension (MTE) introduced in ARMv8.5 Instruction Set Architecture and Top Byte Ignore (TBI).

Special arm64 instructions are used to assign memory tags for each allocation. Same tags are assigned to pointers to those allocations. On every memory access, hardware makes sure that the tag of the memory that is being accessed is equal to the tag of the pointer that is used to access this memory. In case of a tag mismatch, a fault is generated, and a report is printed.

Hardware tag-based KASAN uses 0xFF as a match-all pointer tag (accesses through pointers with the 0xFF pointer tag are not checked). The value 0xFE is currently reserved to tag freed memory regions.

Hardware tag-based KASAN currently only supports tagging of slab, page_alloc, and VM_ALLOC-based vmalloc memory.

If the hardware does not support MTE (pre ARMv8.5), hardware tag-based KASAN will not be enabled. In this case, all KASAN boot parameters are ignored.

Note that enabling CONFIG_KASAN_HW_TAGS always results in in-kernel TBI being enabled. Even when `kasan.mode=off` is provided or when the hardware does not support MTE (but supports TBI).

Hardware tag-based KASAN only reports the first found bug. After that, MTE tag checking gets disabled.

Shadow memory

The contents of this section are only applicable to software KASAN modes.

The kernel maps memory in several different parts of the address space. The range of kernel virtual addresses is large: there is not enough real memory to support a real shadow region for every address that could be accessed by the kernel. Therefore, KASAN only maps real shadow for certain parts of the address space.

Default behaviour

By default, architectures only map real memory over the shadow region for the linear mapping (and potentially other small areas). For all other areas - such as vmalloc and vmemmap space - a single read-only page is mapped over the shadow area. This read-only shadow page declares all memory accesses as permitted.

This presents a problem for modules: they do not live in the linear mapping but in a dedicated module space. By hooking into the module allocator, KASAN temporarily maps real shadow memory to cover them. This allows detection of invalid accesses to module globals, for example.

This also creates an incompatibility with `VMAP_STACK`: if the stack lives in vmalloc space, it will be shadowed by the read-only page, and the kernel will fault when trying to set up the shadow data for stack variables.

CONFIG_KASAN_VMALLOC

With `CONFIG_KASAN_VMALLOC`, KASAN can cover vmalloc space at the cost of greater memory usage. Currently, this is supported on x86, arm64, riscv, s390, and powerpc.

This works by hooking into vmalloc and vmap and dynamically allocating real shadow memory to back the mappings.

Most mappings in vmalloc space are small, requiring less than a full page of shadow space. Allocating a full shadow page per mapping would therefore be wasteful. Furthermore, to ensure that different mappings use different shadow pages, mappings would have to be aligned to `KASAN_GRANULE_SIZE * PAGE_SIZE`.

Instead, KASAN shares backing space across multiple mappings. It allocates a backing page when a mapping in vmalloc space uses a particular page of the shadow region. This page can be shared by other vmalloc mappings later on.

KASAN hooks into the vmap infrastructure to lazily clean up unused shadow memory.

To avoid the difficulties around swapping mappings around, KASAN expects that the part of the shadow region that covers the vmalloc space will not be covered by the early shadow page but will be left unmapped. This will require changes in arch-specific code.

This allows `VMAP_STACK` support on x86 and can simplify support of architectures that do not have a fixed module region.

For developers

Ignoring accesses

Software KASAN modes use compiler instrumentation to insert validity checks. Such instrumentation might be incompatible with some parts of the kernel, and therefore needs to be disabled.

Other parts of the kernel might access metadata for allocated objects. Normally, KASAN detects and reports such accesses, but in

some cases (e.g., in memory allocators), these accesses are valid.

For software KASAN modes, to disable instrumentation for a specific file or directory, add a `KASAN_SANITIZE` annotation to the respective kernel Makefile:

- For a single file (e.g., main.o):

```
KASAN_SANITIZE_main.o := n
```

- For all files in one directory:

```
KASAN_SANITIZE := n
```

For software KASAN modes, to disable instrumentation on a per-function basis, use the KASAN-specific `__no_sanitize_address` function attribute or the generic `noinstr` one.

Note that disabling compiler instrumentation (either on a per-file or a per-function basis) makes KASAN ignore the accesses that happen directly in that code for software KASAN modes. It does not help when the accesses happen indirectly (through calls to instrumented functions) or with the hardware tag-based mode that does not use compiler instrumentation.

For software KASAN modes, to disable KASAN reports in a part of the kernel code for the current task, annotate this part of the code with a `kasan_disable_current()`/`kasan_enable_current()` section. This also disables the reports for indirect accesses that happen through function calls.

For tag-based KASAN modes (include the hardware one), to disable access checking, use `kasan_reset_tag()` or `page_kasan_tag_reset()`. Note that temporarily disabling access checking via `page_kasan_tag_reset()` requires saving and restoring the per-page KASAN tag via `page_kasan_tag`/`page_kasan_tag_set`.

Tests

There are KASAN tests that allow verifying that KASAN works and can detect certain types of memory corruptions. The tests consist of two parts:

1. Tests that are integrated with the KUnit Test Framework. Enabled with `CONFIG_KASAN_KUNIT_TEST`. These tests can be run and partially verified automatically in a few different ways; see the instructions below.

2. Tests that are currently incompatible with KUnit. Enabled with `CONFIG_KASAN_MODULE_TEST` and can only be run as a module. These tests can only be verified manually by loading the kernel module and inspecting the kernel log for KASAN reports.

Each KUnit-compatible KASAN test prints one of multiple KASAN reports if an error is detected. Then the test prints its number and status.

When a test passes:

```
ok 28 - kmalloc_double_kzfree
```

When a test fails due to a failed `kmalloc`:

```
# kmalloc_large_oob_right: ASSERTION FAILED at lib/test_kasan.c:163
Expected ptr is not null, but is
not ok 4 - kmalloc_large_oob_right
```

When a test fails due to a missing KASAN report:

```
# kmalloc_double_kzfree: EXPECTATION FAILED at lib/test_kasan.c:974
KASAN failure expected in "kfree_sensitive(ptr)", but none occurred
not ok 44 - kmalloc_double_kzfree
```

At the end the cumulative status of all KASAN tests is printed. On success:

```
ok 1 - kasan
```

Or, if one of the tests failed:

```
not ok 1 - kasan
```

There are a few ways to run KUnit-compatible KASAN tests.

1. Loadable module

   With `CONFIG_KUNIT` enabled, KASAN-KUnit tests can be built as a loadable module and run by loading `test_kasan.ko` with `insmod` or `modprobe`.

2. Built-In

   With `CONFIG_KUNIT` built-in, KASAN-KUnit tests can be built-in as well. In this case, the tests will run at boot as a late-init call.

3. Using kunit_tool

   With `CONFIG_KUNIT` and `CONFIG_KASAN_KUNIT_TEST` built-in, it is also possible to use `kunit_tool` to see the results of KUnit tests in a more readable way. This will not print the KASAN reports of the tests that passed. See KUnit

documentation for more up-to-date information on `kunit_tool`.