Kernel Electric-Fence (KFENCE)

Kernel Electric-Fence (KFENCE) is a low-overhead sampling-based memory safety error detector. KFENCE detects heap out-of-bounds access, use-after-free, and invalid-free errors.

KFENCE is designed to be enabled in production kernels, and has near zero performance overhead. Compared to KASAN, KFENCE trades performance for precision. The main motivation behind KFENCE's design, is that with enough total uptime KFENCE will detect bugs in code paths not typically exercised by non-production test workloads. One way to quickly achieve a large enough total uptime is when the tool is deployed across a large fleet of machines.

Usage

To enable KFENCE, configure the kernel with:

```
CONFIG_KFENCE=y
```

To build a kernel with KFENCE support, but disabled by default (to enable, set `kfence.sample_interval` to non-zero value), configure the kernel with:

```
CONFIG_KFENCE=y
CONFIG_KFENCE_SAMPLE_INTERVAL=0
```

KFENCE provides several other configuration options to customize behaviour (see the respective help text in `lib/Kconfig.kfence` for more info).

Tuning performance

The most important parameter is KFENCE's sample interval, which can be set via the kernel boot parameter `kfence.sample_interval` in milliseconds. The sample interval determines the frequency with which heap allocations will be guarded by KFENCE. The default is configurable via the Kconfig option `CONFIG_KFENCE_SAMPLE_INTERVAL`. Setting `kfence.sample_interval=0` disables KFENCE.

The sample interval controls a timer that sets up KFENCE allocations. By default, to keep the real sample interval predictable, the normal timer also causes CPU wake-ups when the system is completely idle. This may be undesirable on power-constrained systems. The boot parameter `kfence.deferrable=1` instead switches to a "deferrable" timer which does not force CPU wake-ups on idle systems, at the risk of unpredictable sample intervals. The default is configurable via the Kconfig option `CONFIG_KFENCE_DEFERRABLE`.

> **Warning**
>
> The KUnit test suite is very likely to fail when using a deferrable timer since it currently causes very unpredictable sample intervals.

The KFENCE memory pool is of fixed size, and if the pool is exhausted, no further KFENCE allocations occur. With `CONFIG_KFENCE_NUM_OBJECTS` (default 255), the number of available guarded objects can be controlled. Each object requires 2 pages, one for the object itself and the other one used as a guard page; object pages are interleaved with guard pages, and every object page is therefore surrounded by two guard pages.

The total memory dedicated to the KFENCE memory pool can be computed as:

```
( #objects + 1 ) * 2 * PAGE_SIZE
```

Using the default config, and assuming a page size of 4 KiB, results in dedicating 2 MiB to the KFENCE memory pool.

Note: On architectures that support huge pages, KFENCE will ensure that the pool is using pages of size `PAGE_SIZE`. This will result in additional page tables being allocated.

Error reports

A typical out-of-bounds access looks like this:

```
==================================================================
BUG: KFENCE: out-of-bounds read in test_out_of_bounds_read+0xa6/0x234

Out-of-bounds read at 0xffff8c3f2e291fff (1B left of kfence-#72):
 test_out_of_bounds_read+0xa6/0x234
 kunit_try_run_case+0x61/0xa0
 kunit_generic_run_threadfn_adapter+0x16/0x30
 kthread+0x176/0x1b0
 ret_from_fork+0x22/0x30

kfence-#72: 0xffff8c3f2e292000-0xffff8c3f2e29201f, size=32, cache=kmalloc-32
```

```
allocated by task 484 on cpu 0 at 32.919330s:
 test_alloc+0xfe/0x738
 test_out_of_bounds_read+0x9b/0x234
 kunit_try_run_case+0x61/0xa0
 kunit_generic_run_threadfn_adapter+0x16/0x30
 kthread+0x176/0x1b0
 ret_from_fork+0x22/0x30

CPU: 0 PID: 484 Comm: kunit_try_catch Not tainted 5.13.0-rc3+ #7
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.14.0-2 04/01/2014
==================================================================
```

The header of the report provides a short summary of the function involved in the access. It is followed by more detailed information about the access and its origin. Note that, real kernel addresses are only shown when using the kernel command line option `no_hash_pointers`.

Use-after-free accesses are reported as:

```
==================================================================
BUG: KFENCE: use-after-free read in test_use_after_free_read+0xb3/0x143

Use-after-free read at 0xffff8c3f2e2a0000 (in kfence-#79):
 test_use_after_free_read+0xb3/0x143
 kunit_try_run_case+0x61/0xa0
 kunit_generic_run_threadfn_adapter+0x16/0x30
 kthread+0x176/0x1b0
 ret_from_fork+0x22/0x30

kfence-#79: 0xffff8c3f2e2a0000-0xffff8c3f2e2a001f, size=32, cache=kmalloc-32

allocated by task 488 on cpu 2 at 33.871326s:
 test_alloc+0xfe/0x738
 test_use_after_free_read+0x76/0x143
 kunit_try_run_case+0x61/0xa0
 kunit_generic_run_threadfn_adapter+0x16/0x30
 kthread+0x176/0x1b0
 ret_from_fork+0x22/0x30

freed by task 488 on cpu 2 at 33.871358s:
 test_use_after_free_read+0xa8/0x143
 kunit_try_run_case+0x61/0xa0
 kunit_generic_run_threadfn_adapter+0x16/0x30
 kthread+0x176/0x1b0
 ret_from_fork+0x22/0x30

CPU: 2 PID: 488 Comm: kunit_try_catch Tainted: G    B             5.13.0-rc3+ #7
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.14.0-2 04/01/2014
==================================================================
```

KFENCE also reports on invalid frees, such as double-frees:

```
==================================================================
BUG: KFENCE: invalid free in test_double_free+0xdc/0x171

Invalid free of 0xffff8c3f2e2a4000 (in kfence-#81):
 test_double_free+0xdc/0x171
 kunit_try_run_case+0x61/0xa0
 kunit_generic_run_threadfn_adapter+0x16/0x30
 kthread+0x176/0x1b0
 ret_from_fork+0x22/0x30

kfence-#81: 0xffff8c3f2e2a4000-0xffff8c3f2e2a401f, size=32, cache=kmalloc-32

allocated by task 490 on cpu 1 at 34.175321s:
 test_alloc+0xfe/0x738
 test_double_free+0x76/0x171
 kunit_try_run_case+0x61/0xa0
 kunit_generic_run_threadfn_adapter+0x16/0x30
 kthread+0x176/0x1b0
 ret_from_fork+0x22/0x30

freed by task 490 on cpu 1 at 34.175348s:
 test_double_free+0xa8/0x171
 kunit_try_run_case+0x61/0xa0
 kunit_generic_run_threadfn_adapter+0x16/0x30
 kthread+0x176/0x1b0
 ret_from_fork+0x22/0x30

CPU: 1 PID: 490 Comm: kunit_try_catch Tainted: G    B             5.13.0-rc3+ #7
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.14.0-2 04/01/2014
```

```
==================================================================
```

KFENCE also uses pattern-based redzones on the other side of an object's guard page, to detect out-of-bounds writes on the unprotected side of the object. These are reported on frees:

```
==================================================================
BUG: KFENCE: memory corruption in test_kmalloc_aligned_oob_write+0xef/0x184

Corrupted memory at 0xffff8c3f2e33aff9 [ 0xac . . . . . ] (in kfence-#156):
 test_kmalloc_aligned_oob_write+0xef/0x184
 kunit_try_run_case+0x61/0xa0
 kunit_generic_run_threadfn_adapter+0x16/0x30
 kthread+0x176/0x1b0
 ret_from_fork+0x22/0x30

kfence-#156: 0xffff8c3f2e33afb0-0xffff8c3f2e33aff8, size=73, cache=kmalloc-96

allocated by task 502 on cpu 7 at 42.159302s:
 test_alloc+0xfe/0x738
 test_kmalloc_aligned_oob_write+0x57/0x184
 kunit_try_run_case+0x61/0xa0
 kunit_generic_run_threadfn_adapter+0x16/0x30
 kthread+0x176/0x1b0
 ret_from_fork+0x22/0x30

CPU: 7 PID: 502 Comm: kunit_try_catch Tainted: G    B             5.13.0-rc3+ #7
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.14.0-2 04/01/2014
==================================================================
```

For such errors, the address where the corruption occurred as well as the invalidly written bytes (offset from the address) are shown; in this representation, '.' denote untouched bytes. In the example above `0xac` is the value written to the invalid address at offset 0, and the remaining '.' denote that no following bytes have been touched. Note that, real values are only shown if the kernel was booted with `no_hash_pointers`; to avoid information disclosure otherwise, '!' is used instead to denote invalidly written bytes.

And finally, KFENCE may also report on invalid accesses to any protected page where it was not possible to determine an associated object, e.g. if adjacent object pages had not yet been allocated:

```
==================================================================
BUG: KFENCE: invalid read in test_invalid_access+0x26/0xe0

Invalid read at 0xffffffffb670b00a:
 test_invalid_access+0x26/0xe0
 kunit_try_run_case+0x51/0x85
 kunit_generic_run_threadfn_adapter+0x16/0x30
 kthread+0x137/0x160
 ret_from_fork+0x22/0x30

CPU: 4 PID: 124 Comm: kunit_try_catch Tainted: G        W         5.8.0-rc6+ #7
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.13.0-1 04/01/2014
==================================================================
```

DebugFS interface

Some debugging information is exposed via debugfs:

- The file `/sys/kernel/debug/kfence/stats` provides runtime statistics.
- The file `/sys/kernel/debug/kfence/objects` provides a list of objects allocated via KFENCE, including those already freed but protected.

Implementation Details

Guarded allocations are set up based on the sample interval. After expiration of the sample interval, the next allocation through the main allocator (SLAB or SLUB) returns a guarded allocation from the KFENCE object pool (allocation sizes up to PAGE_SIZE are supported). At this point, the timer is reset, and the next allocation is set up after the expiration of the interval.

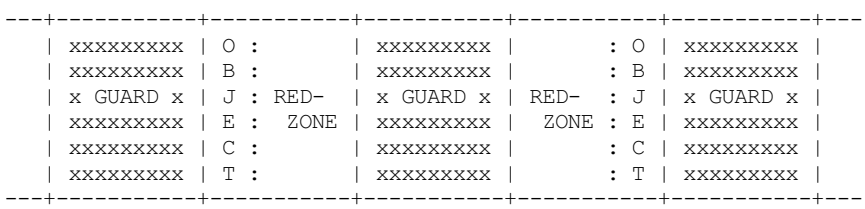
When using `CONFIG_KFENCE_STATIC_KEYS=y`, KFENCE allocations are "gated" through the main allocator's fast-path by relying on static branches via the static keys infrastructure. The static branch is toggled to redirect the allocation to KFENCE. Depending on sample interval, target workloads, and system architecture, this may perform better than the simple dynamic branch. Careful benchmarking is recommended.

KFENCE objects each reside on a dedicated page, at either the left or right page boundaries selected at random. The pages to the left and right of the object page are "guard pages", whose attributes are changed to a protected state, and cause page faults on any attempted access. Such page faults are then intercepted by KFENCE, which handles the fault gracefully by reporting an out-of-bounds access, and marking the page as accessible so that the faulting code can (wrongly) continue executing (set `panic_on_warn` to panic instead).

To detect out-of-bounds writes to memory within the object's page itself, KFENCE also uses pattern-based redzones. For each

object page, a redzone is set up for all non-object memory. For typical alignments, the redzone is only required on the unguarded side of an object. Because KFENCE must honor the cache's requested alignment, special alignments may result in unprotected gaps on either side of an object, all of which are redzoned.

The following figure illustrates the page layout:



Upon deallocation of a KFENCE object, the object's page is again protected and the object is marked as freed. Any further access to the object causes a fault and KFENCE reports a use-after-free access. Freed objects are inserted at the tail of KFENCE's freelist, so that the least recently freed objects are reused first, and the chances of detecting use-after-frees of recently freed objects is increased.

If pool utilization reaches 75% (default) or above, to reduce the risk of the pool eventually being fully occupied by allocated objects yet ensure diverse coverage of allocations, KFENCE limits currently covered allocations of the same source from further filling up the pool. The "source" of an allocation is based on its partial allocation stack trace. A side-effect is that this also limits frequent long-lived allocations (e.g. pagecache) of the same source filling up the pool permanently, which is the most common risk for the pool becoming full and the sampled allocation rate dropping to zero. The threshold at which to start limiting currently covered allocations can be configured via the boot parameter `kfence.skip_covered_thresh` (pool usage%).

Interface

The following describes the functions which are used by allocators as well as page handling code to set up and deal with KFENCE allocations.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\dev-tools\[linux-master][Documentation][dev-tools]kfence.rst, line 305)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/linux/kfence.h
   :functions: is_kfence_address
               kfence_shutdown_cache
               kfence_alloc kfence_free __kfence_free
               kfence_ksize kfence_object_start
               kfence_handle_page_fault
```

Related Tools

In userspace, a similar approach is taken by GWP-ASan. GWP-ASan also relies on guard pages and a sampling strategy to detect memory unsafety bugs at scale. KFENCE's design is directly influenced by GWP-ASan, and can be seen as its kernel sibling. Another similar but non-sampling approach, that also inspired the name "KFENCE", can be found in the userspace Electric Fence Malloc Debugger.

In the kernel, several tools exist to debug memory access errors, and in particular KASAN can detect all bug classes that KFENCE can detect. While KASAN is more precise, relying on compiler instrumentation, this comes at a performance cost.

It is worth highlighting that KASAN and KFENCE are complementary, with different target environments. For instance, KASAN is the better debugging-aid, where test cases or reproducers exists: due to the lower chance to detect the error, it would require more effort using KFENCE to debug. Deployments at scale that cannot afford to enable KASAN, however, would benefit from using KFENCE to discover bugs due to code paths not exercised by test cases or fuzzers.