

Kernel Memory Leak Detector

Kmemleak provides a way of detecting possible kernel memory leaks in a way similar to a [tracing garbage collector](#), 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.

Usage

CONFIG_DEBUG_KMEMLEAK in "Kernel hacking" has to be enabled. A kernel thread scans the memory every 10 minutes (by default) and prints the number of new unreferenced objects found. If the `debugfs` isn't already mounted, mount with:

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

To display the details of all the possible scanned memory leaks:

```
# 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.

Note that the orphan objects are listed in the order they were allocated and one object at the beginning of the list may cause other subsequent objects to be reported as orphan.

Memory scanning parameters can be modified at run-time by writing to the `/sys/kernel/debug/kmemleak` file. The following parameters are supported:

- `off`
disable kmemleak (irreversible)
- `stack=on`
enable the task stacks scanning (default)
- `stack=off`
disable the tasks stacks scanning
- `scan=on`
start the automatic memory scanning thread (default)
- `scan=off`
stop the automatic memory scanning thread
- `scan=<secs>`
set the automatic memory scanning period in seconds (default 600, 0 to stop the automatic scanning)
- `scan`
trigger a memory scan
- `clear`
clear list of current memory leak suspects, done by marking all current reported unreferenced objects grey, or free all kmemleak objects if kmemleak has been disabled.
- `dump=<addr>`
dump information about the object found at <addr>

Kmemleak can also be disabled at boot-time by passing `kmemleak=off` on the kernel command line.

Memory may be allocated or freed before kmemleak is initialised and these actions are stored in an early log buffer. The size of this buffer is configured via the CONFIG_DEBUG_KMEMLEAK_MEM_POOL_SIZE option.

If CONFIG_DEBUG_KMEMLEAK_DEFAULT_OFF are enabled, the kmemleak is disabled by default. Passing `kmemleak=on` on the kernel command line enables the function.

If you are getting errors like "Error while writing to stdout" or "write_loop: Invalid argument", make sure kmemleak is properly enabled.

Basic Algorithm

The memory allocations via `:c:func:'kmalloc'`, `:c:func:'vmalloc'`, `:c:func:'kmem_cache_alloc'` and friends are traced and the pointers,

together with additional information like size and stack trace, are stored in a rbtree. The corresponding freeing function calls are tracked and the pointers removed from the kmemleak data structures.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\dev-tools\linux-master) (Documentation) (dev-tools)kmemleak.rst, line 83); [backlink](#)

Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\dev-tools\linux-master) (Documentation) (dev-tools)kmemleak.rst, line 83); [backlink](#)

Unknown interpreted text role "c:func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\dev-tools\linux-master) (Documentation) (dev-tools)kmemleak.rst, line 83); [backlink](#)

Unknown interpreted text role "c:func".

An allocated block of memory is considered orphan if no pointer to its start address or to any location inside the block can be found by scanning the memory (including saved registers). This means that there might be no way for the kernel to pass the address of the allocated block to a freeing function and therefore the block is considered a memory leak.

The scanning algorithm steps:

1. mark all objects as white (remaining white objects will later be considered orphan)
2. scan the memory starting with the data section and stacks, checking the values against the addresses stored in the rbtree. If a pointer to a white object is found, the object is added to the gray list
3. scan the gray objects for matching addresses (some white objects can become gray and added at the end of the gray list) until the gray set is finished
4. the remaining white objects are considered orphan and reported via /sys/kernel/debug/kmemleak

Some allocated memory blocks have pointers stored in the kernel's internal data structures and they cannot be detected as orphans. To avoid this, kmemleak can also store the number of values pointing to an address inside the block address range that need to be found so that the block is not considered a leak. One example is `__vmalloc()`.

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 a 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
```

Kmemleak API

See the include/linux/kmemleak.h header for the functions prototype.

- `kmemleak_init` - initialize kmemleak
- `kmemleak_alloc` - notify of a memory block allocation

```
# cat /sys/kernel/debug/kmemleak
unreferenced object 0xffff89862ca702e8 (size 32):
  comm "modprobe", pid 2088, jiffies 4294680594 (age 375.486s)
  hex dump (first 32 bytes):
    6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b kkkkkkkkkkkkkkkk
```

```
6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b a5  kkkkkkkkkkkkkkkk.
backtrace:
[<00000000e0a73ec7>] 0xffffffffc01d2036
[<00000000c5d2a46>] do_one_initcall+0x41/0x1df
[<0000000046db7e0a>] do_init_module+0x55/0x200
[<00000000542b9814>] load_module+0x203c/0x2480
[<00000000c2850256>] __do_sys_finit_module+0xba/0xe0
[<000000006564e7ef>] do_syscall_64+0x43/0x110
[<000000007c873fa6>] entry_SYSCALL_64_after_hwframe+0x44/0xa9
...
```

Removing the module with `rmmod kmemleak_test` should also trigger some `kmemleak` results.