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 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
```

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

- `kmemleak_alloc_percpu` - notify of a percpu memory block allocation
- `kmemleak_vmalloc` - notify of a vmalloc() memory allocation
- `kmemleak_free` - notify of a memory block freeing
- `kmemleak_free_part` - notify of a partial memory block freeing
- `kmemleak_free_percpu` - notify of a percpu memory block freeing
- `kmemleak_update_trace` - update object allocation stack trace
- `kmemleak_not_leak` - mark an object as not a leak
- `kmemleak_ignore` - do not scan or report an object as leak
- `kmemleak_scan_area` - add scan areas inside a memory block
- `kmemleak_no_scan` - do not scan a memory block
- `kmemleak_erase` - erase an old value in a pointer variable
- `kmemleak_alloc_recursive` - as kmemleak_alloc but checks the recursiveness
- `kmemleak_free_recursive` - as kmemleak_free but checks the recursiveness

The following functions take a physical address as the object pointer and only perform the corresponding action if the address has a lowmem mapping:

- `kmemleak_alloc_phys`
- `kmemleak_free_part_phys`
- `kmemleak_not_leak_phys`
- `kmemleak_ignore_phys`

Dealing with false positives/negatives

The false negatives are real memory leaks (orphan objects) but not reported by kmemleak because values found during the memory scanning point to such objects. To reduce the number of false negatives, kmemleak provides the kmemleak_ignore, kmemleak_scan_area, kmemleak_no_scan and kmemleak_erase functions (see above). The task stacks also increase the amount of false negatives and their scanning is not enabled by default.

The false positives are objects wrongly reported as being memory leaks (orphan). For objects known not to be leaks, kmemleak provides the kmemleak_not_leak function. The kmemleak_ignore could also be used if the memory block is known not to contain other pointers and it will no longer be scanned.

Some of the reported leaks are only transient, especially on SMP systems, because of pointers temporarily stored in CPU registers or stacks. Kmemleak defines MSECS_MIN_AGE (defaulting to 1000) representing the minimum age of an object to be reported as a memory leak.

Limitations and Drawbacks

The main drawback is the reduced performance of memory allocation and freeing. To avoid other penalties, the memory scanning is only performed when the /sys/kernel/debug/kmemleak file is read. Anyway, this tool is intended for debugging purposes where the performance might not be the most important requirement.

To keep the algorithm simple, kmemleak scans for values pointing to any address inside a block's address range. This may lead to an increased number of false negatives. However, it is likely that a real memory leak will eventually become visible.

Another source of false negatives is the data stored in non-pointer values. In a future version, kmemleak could only scan the pointer members in the allocated structures. This feature would solve many of the false negative cases described above.

The tool can report false positives. These are cases where an allocated block doesn't need to be freed (some cases in the init_call functions), the pointer is calculated by other methods than the usual container_of macro or the pointer is stored in a location not scanned by kmemleak.

Page allocations and ioremap are not tracked.

Testing with kmemleak-test

To check if you have all set up to use kmemleak, you can use the kmemleak-test module, a module that deliberately leaks memory. Set CONFIG_DEBUG_KMEMLEAK_TEST as module (it can't be used as built-in) and boot the kernel with kmemleak enabled. Load the module and perform a scan with:

```
# modprobe kmemleak-test
# echo scan > /sys/kernel/debug/kmemleak
```

Note that the you may not get results instantly or on the first scanning. When kmemleak gets results, it'll log `kmemleak: <count of leaks> new suspected memory leaks`. Then read the file to see then:

```
# 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 6b 6b  kkkkkkkkkkkkkkkk
```

```
      6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b 6b a5  kkkkkkkkkkkkkk.
    backtrace:
      [<00000000e0a73ec7>] 0xffffffffc01d2036
      [<000000000c5d2a46>] 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.