Linux Kernel Memory Leak Detection

Catalin Marinas

LinuxCon Europe 2011

Agenda

- Introduction
- Overview
- Object Tracking
- Memory Scanning
- Limitations
- Usage
- Tips
- Example

Introduction

- First kmemleak patches posted on LKML – January 2006
  - Based on Linux 2.6.15
  - Support for both ARM and x86
  - Using the object size as a weak form of type identification
  - Precise addresses using modified `container_of` macro
  - False positives caused by imprecise type identification
- Found real leaks from the first versions
- Well received by the community
  - LWN article – http://lwn.net/Articles/187979
- Merged into the mainline kernel – June 2009
  - Linux 2.6.31
  - x86, ARM, PPC, MIPS, S390, SPARC64, SuperH, Microblaze, TILE

Overview

- Memory leak example:

```
device->name = kstrdup(device_path, GFP_NOFS);
ret = find_next_devid(root, &device->devid);
if (ret) {
        kfree(device);
        return ret;
}
```

- Small leak initially but may have consequences with long uptime
- If the code is simple, the error could be caught by static analysers
  - Not trivial for more difficult leaks (for example reference counting)

Overview (cont'd)

- Kmemleak is similar to a tracing garbage collector using tri-colour marking (Wikipedia http://bit.ly/q2cSle)
  - White: objects that could be memory leaks
  - Grey: objects known not to be memory leaks
  - Black: objects that have no references to other objects in the white set
- Kmemleak tracks objects allocated via `kmalloc`, `kmem_cache_alloc`, `vmalloc`, `alloc_bootmem` and `pcpu_alloc`
- Page allocations are not tracked
  - Overlapping with other allocators
  - Page cache pages do not contain kernel objects
- There are several calls to `kmemleak_alloc` outside standard allocators

Object Tracking

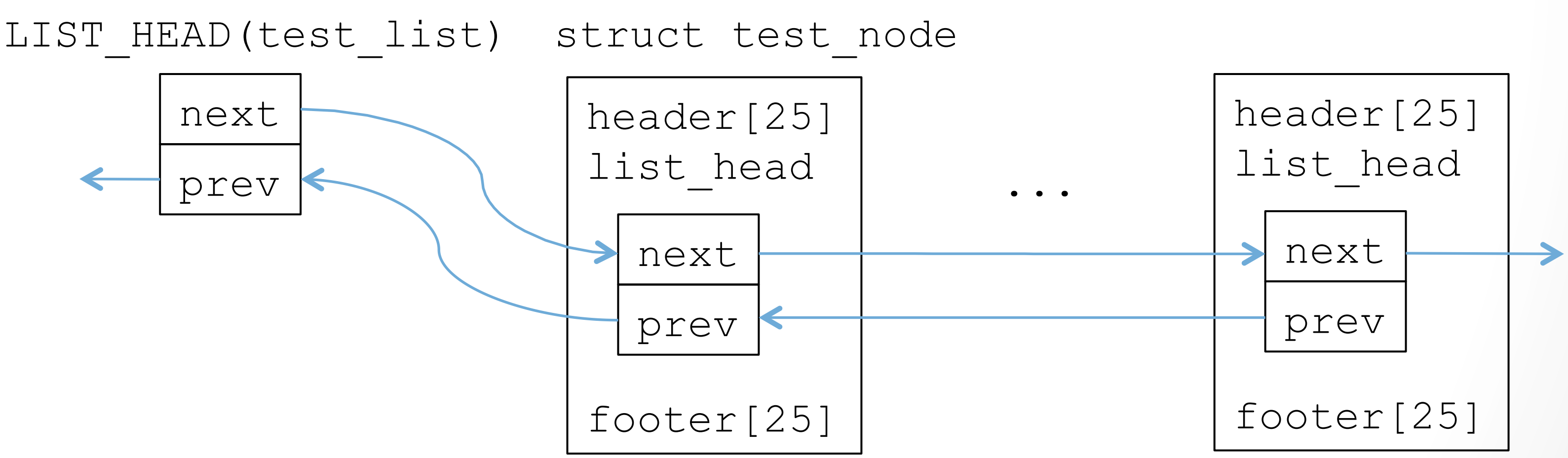
- Kernel memory allocations are recorded by kmemleak
  - It is important that all memory allocations are tracked to avoid false positives
- Kmemleak records the pointer to object, size, backtrace, `jiffies`, `current->pid` and `current->comm`
  - Metadata is stored in a `kmemleak_object` structure
  - The `kmemleak_object` cache is created with `SLAB_NOLEAKTRACE` flag to avoid recursive calls into kmemleak
  - Additional `kmemleak_scan_area` structures may be allocated for an object when only part of the object is relevant
  - Allocation mask preserves `GFP_KERNEL` and `GFP_ATOMIC` used by the allocator caller
- Object boundaries added to a priority search tree

Object Tracking (cont'd)

- Objects are no longer tracked once freed (`kfree` etc.)
  - Kmemleak frees the corresponding metadata
  - To avoid a recursive call to the `kmem_cache_free` function, kmemleak metadata is freed via an RCU callback
- Memory allocations happen before kmemleak is fully initialized
  - Prior calls to the kmemleak API are logged into a static `early_log[CONFIG_DEBUG_KMEMLEAK_EARLY_LOG_SIZE]` array
  - The early log is replayed after kmemleak is initialized

Memory Scanning

- Kmemleak thread scans the memory periodically to identify object referencing (graph)
  - There is no type information for memory locations
  - An address can point anywhere inside an object (`list_head`)
  - Kmemleak API allows scanning of specific areas within an object



Memory Scanning (cont'd)

- Scanning preparation
  - Most objects coloured as white initially (with few exceptions)
  - Known false positives are marked as grey
  - Ignored objects are coloured black
- Scanning starts with the root memory blocks
  - Data and BSS sections
  - Per-CPU sections
  - The `mem_map` array
  - Task stacks (optional)
- Objects referenced during the root memory scanning are marked as grey
- CRC32 calculated for each scanned object

Memory Scanning (cont'd)

- Scanning continues with the grey objects
  - Any referenced white object is marked as grey
  - Completes when all the grey objects have been scanned
- The remaining white objects are considered unreferenced and reported as memory leaks
- Scanning the memory could take a long time (minutes)
  - Cannot lock the system during scanning
  - Memory allocation/freeing can still happen during scanning
  - Objects are modified (added/removed from lists etc.)
- Kmemleak uses RCU list traversal to avoid locking
- Recently allocated objects or objects modified since the previous scan are coloured grey initially

Limitations

- False negatives
  - Leaks may be hidden by memory locations looking like real addresses
  - Type identification is not possible
  - Task stacks have many address-like values
  - The leak will eventually be found if running for long enough or on a wider range of platforms
- False positives
  - Objects falsely reported as leaks
  - Usually for objects referenced from other objects that are not tracked by kmemleak (like page allocations)
  - Object referenced via a modified pointer (like physical address)
  - API provided for annotating false positives

Limitations (cont'd)

- Does not allow overlapping objects
  - All pointers must be real addresses in the kernel virtual space
  - Per-CPU allocations are scanned but never considered leaks
  - Non-virtual address pointers cannot be tracked (IOMMU etc.)
- `ioremap` mappings are not tracked
- System performance slightly affected
  - Every slab memory allocation is logged by kmemleak together with the backtrace
  - Memory freeing is logged by kmemleak and an RCU callback scheduled to clean up the metadata

Usage

- Kmemleak API described in Documentation/kmemleak.txt
- `CONFIG_DEBUG_KMEMLEAK=y`
- `CONFIG_DEBUG_KMEMLEAK_EARLY_LOG_SIZE=400` (default)
- Allocation/freeing API
  - `kmemleak_alloc`: memory allocation callback
  - `kmemleak_free`: memory freeing callback
  - `kmemleak_alloc_recursive`: slab allocation callback
  - `kmemleak_free_recursive`: slab freeing callback
- False positive annotation API
  - `kmemleak_not_leak`: never report an object as leak
  - But prefer to find where the object is referenced from and use `kmemleak_alloc`

Usage (cont'd)

- False negative reduction API
  - `kmemleak_scan_area`: only scan an object area
  - `kmemleak_no_scan`: do not scan the object
  - `kmemleak_ignore`: do not scan or report an object as leak
  - `kmemleak_erase`: erase a pointer variable
- User level reporting
  - `kmemleak: N new suspected memory leaks (see /sys/kernel/debug/kmemleak`)
  - `mount -t debugfs nodev /sys/kernel/debug/`
  - `cat /sys/kernel/debug/kmemleak`
- Kmemleak behaviour can be modified at runtime by writing to the `/sys/kernel/debug/kmemleak` file
  - `off`: disables kmemleak (irreversible)

Usage (cont'd)

- `stack=on|off`: enable|disable task stack scanning (default `on`)
- `scan=on|off`: enable|disable the scanning thread (default `on`)
- `scan=<secs>`: set the scanning period (default `600`)
- `scan`: trigger a memory scan
- `clear`: clear the list of memory leaks reported (current white objects coloured grey)
- `dump=<addr>`: show information about an object at `<addr>`

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

Tips

- Kmemleak only shows backtrace to the allocation point
- Objects are reported in the order they were allocated
- Check the kernel commit log and possibly bisect
- Analyse the backtrace (using `addr2line`) and follow where the reported pointer was stored
  - Possibly add `printk` calls throughout the kernel
- Check the status of the object referencing the leaked pointer
  - `echo dump=<addr> > /sys/kernel/debug/kmemleak`
  - If freed, check the clean-up code
  - If still present, check for code overriding the leaked pointer
- Check list deletion, reference counting
- It could be a false positive

Example

```
kmemleak: 1 new suspected memory leaks (see /sys/kernel/
debug/kmemleak)
# cat /sys/kernel/debug/kmemleak
unreferenced object 0xef42d000 (size 28):
  comm "khubd", pid 189, jiffies 4294937550 (age 2543.370s)
  hex dump (first 28 bytes):
    00 01 10 00 00 02 20 00 08 d0 42 ef 08 d0 42 ef
    00 00 00 00 00 00 00 00 ff ff ff ff
  backtrace:
    [<c0080fe1>] create_object+0xa1/0x1ac
    [<c007eac5>] kmem_cache_alloc+0x8d/0xdc
    [<c01a966d>] isp1760_urb_enqueue+0x2f9/0x358
    [<c019bbbd>] usb_hcd_submit_urb+0x75/0x574
    [<c019d8f1>] usb_start_wait_urb+0x29/0x80
    [<c019daad>] usb_control_msg+0x89/0xac
    [<c0197f43>] hub_port_init+0x4fb/0x9c8
```

Slab allocator invoked

Example (cont'd)

```
# addr2line -i -f -e vmlinux c01a966d
qh_alloc
drivers/usb/host/isp1760-hcd.c:382
isp1760_urb_enqueue
drivers/usb/host/isp1760-hcd.c:1531
# vi drivers/usb/host/isp1760-hcd.c +1531
       ...
       ep_queue = &priv->controlqhs;
       ...
       qh = qh_alloc(GFP_ATOMIC);
       if (!qh) {
               retval = -ENOMEM;
               goto out;
       }
       list_add_tail(&qh->qh_list, ep_queue);
       urb->ep->hcpriv = qh;
```

Inlined function

Pointer stored

Example (cont'd)

```
# grep -n list_del drivers/usb/host/isp1760-hcd.c
1017: list_del(&qh->qh_list);
# vi drivers/usb/host/isp1760-hcd.c +1017
void schedule_ptds(struct usb_hcd *hcd)
      ...
      list_for_each_entry_safe(qh, ...) {
            ...
            list_del(&qh->qh_list);
            if (ep->hcpriv == NULL) {
                  /* Endpoint has been disabled, so we
                  can free the associated queue head. */
                  qh_free(qh);
            }
            ...
      }
```

Object removed from list

Condition false

Function not called

Example (cont'd)

```
# grep -n hcpriv drivers/usb/host/isp1760-hcd.c
1634: ep->hcpriv = NULL;
# vi drivers/usb/host/isp1760-hcd.c +1634
static void isp1760_endpoint_disable(...)
      ...
      ep->hcpriv = NULL;
      /* Cannot free qh here since it will be parsed by
         schedule_ptds() */
      schedule_ptds(hcd);
      ...
```

Last reference overridden

Leak possibly caused by a race condition: `schedule_ptds` called from `isp1760_irq` before `isp1760_endpoint_disable` cleared `ep->hcpriv`.

Questions