Finding errors with

kmemcheck

Vegard Nossum <vegardno@ifi.uio.no>

Etymology

- **“k”** is for kernel
- **“memcheck”** is a reference to Valgrind's memcheck
- kmemcheck and memcheck: same concept, different application and implementation

Error classes

- Using memory before it has been assigned a value (“use-before-assign”)
- Using memory after it has been deallocated (“use-after-free”)
- Leaking uninitialised memory to userspace applications (“information leaks”)

Use-before-assign errors

- Memory is allocated, but not initialised right away
- For arrays: Not all elements are initialised
- Typically: Caller (incorrectly) assumes completely initialised object

- Example:

```
struct foo {
        int x;
        int y;
};

struct foo *f = kmalloc(...);
f->x = 0;
return f;
```

Use-after-free errors

- Pointers to freed memory still exist
- Example:

```
journal_destroy(sb->journal);
sb->journal = NULL;
...
/* in a different function, also operating on journal objects: */
if (sb->journal)
        journal_foo(sb->journal);
```

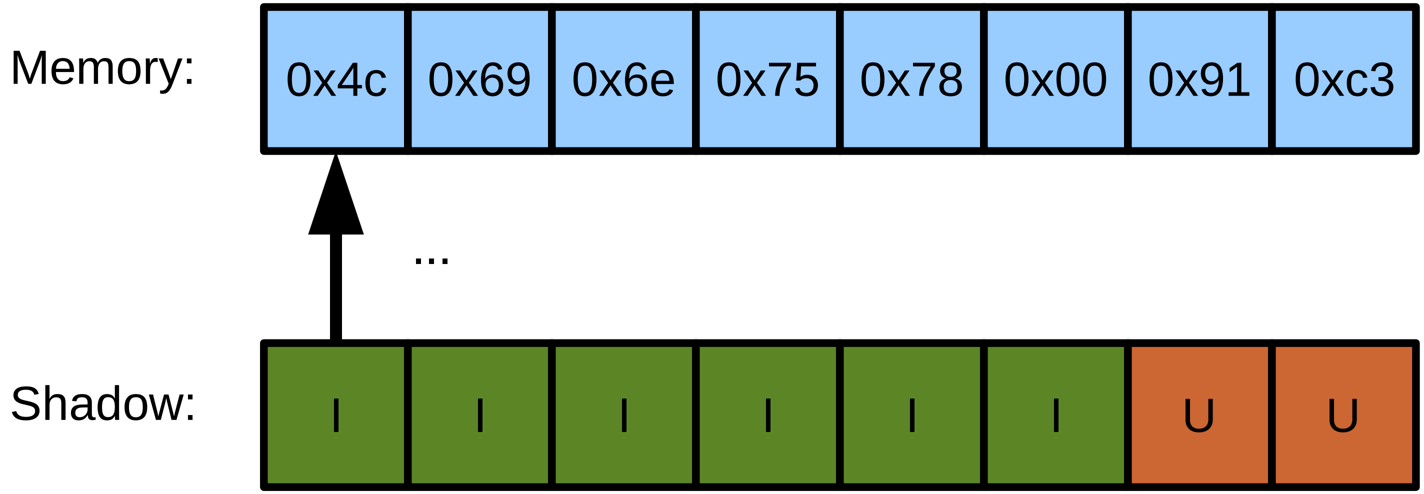
Information leak errors

- Use-before-assign and use-after-free errors can also be information leaks
- More typically, some (partly or completely uninitialised) data block is copied directly to userspace
- Could disclose sensitive information: encryption keys, private data (unlikely)

Concept

- Every byte of dynamically-allocatable memory has a corresponding shadow state
- Shadow state can be (simplified):
  - initialised, uninitialised, freed
- Track every memory access:
  - Writes set shadow state to “initialised”
  - Reads are checked to make sure what is read is “initialised”

Shadow state



Memory allocator hooks

- kmalloc() and kfree()
- alloc_pages() and free_pages()

- Allocate (and initialise!) and deallocate the shadow-state “bytemap”

Tracking memory accesses

- We exploit the paging mechanism of the MMU
  - Pages are marked “non-present”
  - Forces a page fault exception (#PF) on access
  - Inspect the instruction and register state
  - Update/verify shadow state
  - Pages are marked “present”
  - Continue execution

Single-stepping

- We exploit the built-in debugging mechanisms of the CPU
  - Page fault handler enables instruction single-stepping
  - Forces a Debug Exception (#DB) after the instruction has executed
  - Pages are marked “non-present” again (to catch the next memory access too!)
  - Continue execution

Performance impact

- No hard numbers, but:
- Kernel needs about 2x RAM
- Kernel boot takes about 10x the time
- Slowdown depends on workload
  - Userspace is unaffected!
- My 1.4 GHz laptop can boot X and play MP3s

Results

- About 10 patches in mainline Linux with fixes for real problems
- Use-before-assign: 2
- Use-after-free: 4
- Information leaks: 4
- Not too much :-(

Example

```
kmemcheck: Caught 16-bit read from uninitialized memory (f6c1ba30)
0500110001508abf050010000500000002017300140000006f72672e66726565
 i i i i i i i i i i i i i u u u u u u u u u u u u u u u u u u u
                                 ^

Pid: 3462, comm: wpa_supplicant Not tainted (2.6.27-rc3-00054-g6397ab9-dir
EIP: 0060:[<c05de64a>] EFLAGS: 00010296 CPU: 0
EIP is at nla_parse+0x5a/0xf0
EAX: 00000008 EBX: fffffffd ECX: c06f16c0 EDX: 00000005
ESI: 00000010 EDI: f6c1ba30 EBP: f6367c6c ESP: c0a11e88
 DS: 007b ES: 007b FS: 00d8 GS: 0033 SS: 0068
CR0: 8005003b CR2: f781cc84 CR3: 3632f000 CR4: 000006d0
DR0: c0ead9bc DR1: 00000000 DR2: 00000000 DR3: 00000000
DR6: ffff4ff0 DR7: 00000400
 [<c05d4b23>] rtnl_setlink+0x63/0x130
 [<c05d5f75>] rtnetlink_rcv_msg+0x165/0x200
 [<c05ddf66>] netlink_rcv_skb+0x76/0xa0
 [<c05d5dfe>] rtnetlink_rcv+0x1e/0x30
 [<c05dda21>] netlink_unicast+0x281/0x290
 [<c05ddbe9>] netlink_sendmsg+0x1b9/0x2b0
 [<c05beef2>] sock_sendmsg+0xd2/0x100
 [<c05bf945>] sys_sendto+0xa5/0xd0
 [<c05bf9a6>] sys_send+0x36/0x40
 [<c05c03d6>] sys_socketcall+0x1e6/0x2c0
 [<c020353b>] sysenter_do_call+0x12/0x3f
 [<ffffffff>] 0xffffffff
```

Example, continued

```
/* nla_ok - check if the netlink attribute fits into the remaining bytes
 * @remaining: number of bytes remaining in attribute stream */
static inline int nla_ok(const struct nlattr *nla, int remaining) {
    return remaining >= sizeof(*nla)
        && nla->nla_len >= sizeof(*nla) && nla->nla_len <= remaining;
}

/* nla_next - next netlink attribute in attribute stream
 * @remaining: number of bytes remaining in attribute stream */
static inline struct nlattr *nla_next(const struct nlattr *nla, int *remaining) {
        int totlen = NLA_ALIGN(nla->nla_len);
        *remaining -= totlen;
        return (struct nlattr *) ((char *) nla + totlen);
}

/* nla_for_each_attr - iterate over a stream of attributes
 * @pos: loop counter, set to current attribute
 * @head: head of attribute stream
 * @len: length of attribute stream
 * @rem: initialized to len, holds bytes currently remaining in stream */
#define nla_for_each_attr(pos, head, len, rem) \
        for (pos = head, rem = len; \
             nla_ok(pos, rem); \
             pos = nla_next(pos, &(rem)))
```

Complications

- Instructions with more than one memory operand (we still only get one page fault)
- Processor peculiarities
- DMA accesses (doesn't go through the MMU)
- SMP (Symmetric Multi-Processing)
- Local (on-stack) variables
- Bitfields (shadow state has byte granularity)

SMP

- Updating shared page tables is racy:
  - CPU 1 marks page “non-present”
  - CPU 2 writes data to page
  - CPU 1 marks page “present”
- Currently limited to 1 CPU

- Solution 1: Per-CPU page tables
- Solution 2: Instruction emulation

Local (on-stack) variables

- Can't mark stack pages “non-present”
- (Would cause triple fault when trying to call the page fault handler)
- Will cause false-positive reports; example:

```
void func(struct foo *x) {
        /* Oops: */
        struct foo y = *x;
        ...
}
```

Local variables, continued

- Solution 1: Don't track known-problematic allocations
  - Trade-off between coverage and false positives
- Solution 2: Single-step all instructions

Bitfields

- Example:

```
struct foo {
        int x:1;
        int y:1;
};

struct foo *f = kmalloc(...);
f->x = 1;
f->y = 2;
```

- Assembly code:

```
...
call kmalloc

# Oops:
movzbl (%eax), %edx
andl $-2, %edx
orl $2, %edx
movb %dl, (%eax)
```

Bitfields, continued

- Solution 1: Annotate bitfields
  - Requires marking up the source code
  - Not fool-proof
- Solution 2: Single-step all instructions
- Solution 3: Change gcc to emit kmemcheck-friendly code?

Single-stepping everything

- Most kernel code would be single-stepped
- Every instruction is decoded in software
- Bit (instead of byte) granularity!
- No need for page faults
- We get (for free):
  - SMP, bitfields, local variables
- Drawbacks:
  - Slowdown?

Changing GCC

- C code:

```
void func(int *p) {
        *p = 40;
}
```

- Assembly code:

```
- movl 8(%ebp), %eax
- movl $40, (%eax)
________________________________
+ movl $40, 4(%esp)
+ movl 8(%ebp), %eax
+ movl %eax, (%esp)
+ call kmemcheck_write_int
```

Changing GCC, continued

- Difficulty?
- We get (probably with hard work):
  - SMP, bitfields, local variables
- Drawback:
  - Who wants to do it? ;-)

Thanks & credits

- Pekka Enberg (slab maintainer, kmemcheck co-maintainer)
- Ingo Molnar (x86 maintainer)
- Many others for showing interest!