Heap use-after-free in lduw_he_p through e1000e_write_to_rx_buffers

Bug #1886362 reported by Alexander Bulekov on 2020-07-06

This bug affects 1 person 8

Affects	Status	Importance	Assigned to	Milestone
QEMU	Fix Released	Undecided	Unassigned	

Bug Description

```
Hello,
This reproducer causes a heap-use-after free. QEMU Built with --enable-
sanitizers:
cat << EOF | ./i386-softmmu/qemu-system-i386 -M q35,accel=qtest \
-qtest stdio -nographic -monitor none -serial none
outl 0xcf8 0x80001010
outl 0xcfc 0xe1020000
outl 0xcf8 0x80001014
outl 0xcf8 0x80001004
outw 0xcfc 0x7
outl 0xcf8 0x800010a2
write 0xe102003b 0x1 0xff
write 0xe1020103 0x1e 0xffffff055c5e5c30be4511d084ffffffffffffffff
ffffffffffffffff
write 0xe1020420 0x4 0xffffffff
write 0xe1020424 0x4 0xffffffff
write 0xe102042b 0x1 0xff
write 0xe1020430 0x4 0x055c5e5c
write 0x5c041 0x1 0x04
write 0x5c042 0x1 0x02
write 0x5c043 0x1 0xe1
write 0x5c048 0x1 0x8a
write 0x5c04a 0x1 0x31
write 0x5c04b 0x1 0xff
write 0xe1020403 0x1 0xff
EOF

The Output:
==22689==ERROR: AddressSanitizer: heap-use-after-free on address
0x62500026800e at pc 0x55b93bb18bfa bp 0x7fffdbe844f0 sp 0x7fffdbe83cb8
READ of size 2 at 0x62500026800e thread T0
    #0 in __asan_memcpy (/build/i386-softmmu/qemu-system-i386+)
    #1 in lduw_he_p /include/qemu/bswap.h:332:5
    #2 in ldn_he_p /include/qemu/bswap.h:550:1
    #3 in flatview_write_continue /exec.c:3145:19
    #4 in flatview_write /exec.c:3186:14
    #5 in address_space_write /exec.c:3280:18
    #6 in address_space_rw /exec.c:3290:16
    #7 in dma_memory_rw_relaxed /include/sysemu/dma.h:87:18
    #8 in dma_memory_rw /include/sysemu/dma.h:113:12
    #9 in pci_dma_rw /include/hw/pci/pci.h:789:5
    #10 in pci_dma_write /include/hw/pci/pci.h:802:12
    #11 in e1000e_write_to_rx_buffers /hw/net/e1000e_core.c:1412:9
    #12 in e1000e_write_packet_to_guest /hw/net/e1000e_core.c:1582:21
    #13 in e1000e_receive_iov /hw/net/e1000e_core.c:1709:9
    #14 in e1000e_nc_receive_iov /hw/net/e1000e.c:213:12
    #15 in net_tx_pkt_sendv /hw/net/net_tx_pkt.c:544:9
    #16 in net_tx_pkt_send /hw/net/net_tx_pkt.c:620:9
    #17 in net_tx_pkt_send_loopback /hw/net/net_tx_pkt.c:633:11
    #18 in e1000e_tx_pkt_send /hw/net/e1000e_core.c:664:16
    #19 in e1000e_process_tx_desc /hw/net/e1000e_core.c:743:17
    #20 in e1000e_start_xmit /hw/net/e1000e_core.c:934:9
    #21 in e1000e_set_tctl /hw/net/e1000e_core.c:2431:9
    #22 in e1000e_core_write /hw/net/e1000e_core.c:3265:9
    #23 in e1000e_mmio_write /hw/net/e1000e.c:109:5
    #24 in memory_region_write_accessor /memory.c:483:5
    #25 in access_with_adjusted_size /memory.c:544:18
    #26 in memory_region_dispatch_write /memory.c:1476:16
    #27 in flatview_write_continue /exec.c:3146:23
    #28 in flatview_write /exec.c:3186:14
    #29 in address_space_write /exec.c:3280:18
    #30 in qtest_process_command /qtest.c:567:9
    #31 in qtest_process_inbuf /qtest.c:710:9
    #32 in qtest_read /qtest.c:722:5
    #33 in qemu_chr_be_write_impl /chardev/char.c:188:9
    #34 in qemu_chr_be_write /chardev/char.c:200:9
    #35 in fd_chr_read /chardev/char-fd.c:68:9
    #36 in qio_channel_fd_source_dispatch /io/channel-watch.c:84:12
    #37 in g_main_context_dispatch (/usr/lib/x86_64-linux-gnu/libglib-
2.0.so.0+)
    #38 in glib_pollfds_poll /util/main-loop.c:219:9
    #39 in os_host_main_loop_wait /util/main-loop.c:242:5
    #40 in main_loop_wait /util/main-loop.c:518:11
    #41 in qemu_main_loop /softmmu/vl.c:1664:9
    #42 in main /softmmu/main.c:52:5
    #43 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+)
    #44 in _start (/build/i386-softmmu/qemu-system-i386+)

0x62500026800e is located 14 bytes inside of 138-byte region
[0x625000268000,0x62500026808a)
freed by thread T0 here:
    #0 in free (/build/i386-softmmu/qemu-system-i386+)
    #1 in qemu_vfree /util/oslib-posix.c:238:5
    #2 in address_space_unmap /exec.c:3616:5
    #3 in dma_memory_unmap /include/sysemu/dma.h:148:5
    #4 in pci_dma_unmap /include/hw/pci/pci.h:839:5
    #5 in net_tx_pkt_reset /hw/net/net_tx_pkt.c:453:9
    #6 in e1000e_process_tx_desc /hw/net/e1000e_core.c:749:9
    #7 in e1000e_start_xmit /hw/net/e1000e_core.c:934:9
    #8 in e1000e_set_tctl /hw/net/e1000e_core.c:2431:9
    #9 in e1000e_core_write /hw/net/e1000e_core.c:3265:9
    #10 in e1000e_mmio_write /hw/net/e1000e.c:109:5
```

```
    #11 in memory_region_write_accessor /memory.c:483:5
    #12 in access_with_adjusted_size /memory.c:544:18
    #13 in memory_region_dispatch_write /memory.c:1476:16
    #14 in flatview_write_continue /exec.c:3146:23
    #15 in flatview_write /exec.c:3186:14
    #16 in address_space_write /exec.c:3280:18
    #17 in address_space_rw /exec.c:3290:16
    #18 in dma_memory_rw_relaxed /include/sysemu/dma.h:87:18
    #19 in dma_memory_rw /include/sysemu/dma.h:113:12
    #20 in pci_dma_rw /include/hw/pci/pci.h:789:5
    #21 in pci_dma_write /include/hw/pci/pci.h:802:12
    #22 in e1000e_write_to_rx_buffers /hw/net/e1000e_core.c:1412:9
    #23 in e1000e_write_packet_to_guest /hw/net/e1000e_core.c:1582:21
    #24 in e1000e_receive_iov /hw/net/e1000e_core.c:1709:9
    #25 in e1000e_nc_receive_iov /hw/net/e1000e.c:213:12
    #26 in net_tx_pkt_sendv /hw/net/net_tx_pkt.c:544:9
    #27 in net_tx_pkt_send /hw/net/net_tx_pkt.c:620:9
    #28 in net_tx_pkt_send_loopback /hw/net/net_tx_pkt.c:633:11
    #29 in e1000e_tx_pkt_send /hw/net/e1000e_core.c:664:16

previously allocated by thread T0 here:
    #0 in posix_memalign (/build/i386-softmmu/qemu-system-i386+)
    #1 in qemu_try_memalign /util/oslib-posix.c:198:11
    #2 in qemu_memalign /util/oslib-posix.c:214:27
    #3 in address_space_map /exec.c:3558:25
    #4 in dma_memory_map /include/sysemu/dma.h:138:9
    #5 in pci_dma_map /include/hw/pci/pci.h:832:11
    #6 in net_tx_pkt_add_raw_fragment /hw/net/net_tx_pkt.c:391:24
    #7 in e1000e_process_tx_desc /hw/net/e1000e_core.c:731:14
    #8 in e1000e_start_xmit /hw/net/e1000e_core.c:934:9
    #9 in e1000e_set_tctl /hw/net/e1000e_core.c:2431:9
    #10 in e1000e_core_write /hw/net/e1000e_core.c:3265:9
    #11 in e1000e_mmio_write /hw/net/e1000e.c:109:5
    #12 in memory_region_write_accessor /memory.c:483:5
    #13 in access_with_adjusted_size /memory.c:544:18
    #14 in memory_region_dispatch_write /memory.c:1476:16
    #15 in flatview_write_continue /exec.c:3146:23
    #16 in flatview_write /exec.c:3186:14
    #17 in address_space_write /exec.c:3280:18
    #18 in qtest_process_command /qtest.c:567:9
    #19 in qtest_process_inbuf /qtest.c:710:9
    #20 in qtest_read /qtest.c:722:5
    #21 in qemu_chr_be_write_impl /chardev/char.c:188:9
    #22 in qemu_chr_be_write /chardev/char.c:200:9
    #23 in fd_chr_read /chardev/char-fd.c:68:9
    #24 in qio_channel_fd_source_dispatch /io/channel-watch.c:84:12
    #25 in g_main_context_dispatch (/usr/lib/x86_64-linux-gnu/libglib-
2.0.so.0+)

-Alex
```

Philippe Mathieu-Daudé (philmd) wrote on 2020-07-07: #1

Download full text (8.0 KiB)

```
Running with '-trace e1000\*':

e1000e_cb_pci_realize E1000E PCI realize entry
e1000e_mac_set_permanent Set permanent MAC: 52:54:00:12:34:56
e1000e_cfg_support_virtio Virtio header supported: 0
e1000e_rx_set_cso RX CSO state set to 0
e1000e_cb_qdev_reset E1000E qdev reset entry
e1000x_mac_indicate Indicating MAC to guest: 52:54:00:12:34:56
e1000x_rx_can_recv_disabled link_up: 1, rx_enabled 0, pci_master 0
e1000x_rx_can_recv_disabled link_up: 1, rx_enabled 0, pci_master 0
e1000e_vm_state_running VM state is running
[R +0.094581] outl 0xcf8 0x80001010
[S +0.094604] OK
[R +0.094632] outl 0xcfc 0xe1020000
[S +0.094654] OK
[R +0.094668] outl 0xcf8 0x80001014
[S +0.094675] OK
[R +0.094694] outl 0xcf8 0x80001004
[S +0.094702] OK
[R +0.094712] outw 0xcfc 0x7
e1000e_rx_start_recv
[S +0.096938] OK
[R +0.096960] outl 0xcf8 0x800010a2
[S +0.096972] OK
[R +0.096986] write 0xe102003b 0x1 0xff
e1000e_core_write Write to register 0x38, 4 byte(s), value: 0xff
e1000e_vlan_vet Setting VLAN ethernet type 0xFF
[S +0.097019] OK
[R +0.097034] write 0xe1020103 0x1e 0xffffff055c5e5c30be4511d084ff
ffffffffffffffffffffffffffffffff
e1000e_core_write Write to register 0x100, 4 byte(s), value: 0xff
e1000e_rx_set_rctl RCTL = 0xff
e1000e_rx_desc_buff_sizes buffer sizes: [2048, 0, 0, 0]
e1000e_rx_desc_len RX descriptor length: 16
e1000e_rx_start_recv
e1000e_wrn_regs_write_unknown WARNING: Write to unknown register 0x104, 4
byte(s), value: 0x5c05ffff
e1000e_core_write Write to register 0x2820, 4 byte(s), value: 0xbe305c5e
e1000e_irq_rdtr_fpd_not_running FPD written while RDTR was not running
e1000e_wrn_regs_write_unknown WARNING: Write to unknown register 0x10c, 4
byte(s), value: 0x84d01145
e1000e_core_write Write to register 0x2800, 4 byte(s), value: 0xffffffff
e1000e_core_write Write to register 0x2804, 4 byte(s), value: 0xffffffff
e1000e_core_write Write to register 0x2808, 4 byte(s), value: 0xffffffff
e1000e_wrn_regs_write_unknown WARNING: Write to unknown register 0x11c, 4
byte(s), value: 0xffffffff
e1000e_core_write Write to register 0x2810, 4 byte(s), value: 0xff
[S +0.097143] OK
[R +0.097159] write 0xe1020420 0x4 0xffffffff
e1000e_core_write Write to register 0x3800, 4 byte(s), value: 0xffffffff
[S +0.097173] OK
[R +0.097183] write 0xe1020424 0x4 0xffffffff
e1000e_core_write Write to register 0x3804, 4 byte(s), value: 0xffffffff
[S +0.097196] OK
```

```
[R +0.097208] write 0xe102042b 0x1 0xff
e1000e_core_write Write to register 0x3808, 4 byte(s), value: 0xff
[S +0.097221] OK
[R +0.097231] write 0xe1020430 0x4 0x055c5e5c
e1000e_core_write Write to register 0x3810, 4 byte(s), value: 0x5c5e5c05
[S +0.097243] OK
[R +0.097253] write 0x5c041 0x1 0x04
[S +0.097914] OK
[R +0.097942] write 0x5c042 0x1 0x02
[S +0.097953] OK
[R +0.097964] write 0x5c043 0x1 0xe1
[S +0.097972] OK
[R +0.097984] write 0x5c048 0x1 0x8a
[S +0.097992] OK
[R +0.098003] write 0x5c04a 0x1 0x31
[S +0.098011] OK
[R +0.098022] write 0x5c04b 0x1 0xff
[S +0.098029] OK
[R +0.098040] write 0xe1020403 0x1 0xff
e1000e_core_write Write to register 0x400, 4 byte(s), value: 0xff
e1000e_tx_descr 0xe1020400 : ff31008a 0
e1000e_core_read Read from register 0x400, 4 byte(s), value: 0xff
e1000e_wrn_regs_read_unknown WARNING: Rea...
```

Read more...

Jason Wang (jasowang) wrote on 2020-07-14: **Re: [Bug 1886362] [NEW] Heap use-after-free in lduw_he_p through e1000e_write_to_rx_buffers** #2

```
On 2020/7/10 下午6:37, Li Qiang wrote:
> Paolo Bonzini <email address hidden> 于2020年7月10日周五 上午1:36写道:
>> On 09/07/20 17:51, Li Qiang wrote:
>>> Maybe we should check whether the address is a RAM address in
'dma_memory_rw'?
>>> But it is a hot path. I'm not sure it is right. Hope more discussion.
>> Half of the purpose of dma-helpers.c (as opposed to address_space_*
>> functions in exec.c) is exactly to support writes to MMIO. This is
> Hi Paolo,
>
> Could you please explain more about this(to support writes to MMIO).
> I can just see the dma helpers with sg DMA, not related with MMIO.

Please refer doc/devel/memory.rst.

The motivation of memory API is to allow support modeling different
memory regions. DMA to MMIO is allowed in hardware so Qemu should
emulate this behaviour.

[...]

Probably a TX bh.

> So even if we again trigger the MMIO write, then
> second bh will not be executed?

Bh is serialized so no re-entrancy issue.

Thanks

>
>
> Thanks,
> Li Qiang
>
>> Paolo
>>
```

Jason Wang (jasowang) wrote on 2020-07-15: #3

```
On 2020/7/14 下午6:48, Li Qiang wrote:
[...]

I think the point is to make DMA to MMIO work as real hardware. For
e1000e and other networking devices we need make sure such DMA doesn't
break anything.

Thanks

[...]
```

P J P (pjps) wrote on 2020-07-21: #4

```
Another reproducer: (just to record)

cat << EOF | ./i386-softmmu/qemu-system-i386 -M pc-q35-5.0 \
-netdev user,id=qtest-bn0 -device e1000e,netdev=qtest-bn0 \
-display none -nodefaults -nographic -qtest stdio
outl 0xcf8 0x80000810
outl 0xcfc 0xe0000000
outl 0xcf8 0x80000804
outw 0xcfc 0x7
write 0xe0000758 0x4 0xfffff1ff
write 0xe0000760 0x6 0xffffdf000000
write 0xe0000768 0x4 0x0effff1
write 0xe0005008 0x4 0x18ffff27
write 0xe0000c 0x1 0x66
write 0xe03320 0x1 0xff
write 0xe03620 0x1 0xff
write 0xe00000f3 0x1 0xdf
write 0xe0000100 0x6 0xdffffdf0000
write 0xe0000110 0x5 0xdffffdf00
write 0xe000011a 0x3 0xffffff
write 0xe0000128 0x5 0x7e00ffffff
write 0xe0000403 0x1 0xdf
write 0xe0000420 0x4 0xdffffdf
write 0xe000042a 0x3 0xffffff
write 0xe0000438 0x1 0x7e
EOF
```

-> https://lists.gnu.org/archive/html/qemu-devel/2020-07/msg05709.html

Peter Maydell (pmaydell) wrote on 2020-07-21: #5

On Wed, 15 Jul 2020 at 09:36, Jason Wang <email address hidden> wrote:
> I think the point is to make DMA to MMIO work as real hardware.

I wouldn't care to give a 100% guarantee that asking a real
h/w device to DMA to itself didn't cause it to misbehave :-)
It's more likely to happen-to-work because the DMA engine bit
of a real h/w device is going to be decoupled somewhat from
the respond-to-memory-transactions-for-registers logic, but
it probably wasn't something the designers were actively
thinking about either...

> For
> e1000e and other networking devices we need make sure such DMA doesn't
> break anything.

Yeah, this is the interesting part for QEMU. How should we
structure devices that do DMA so that we can be sure that
the device emulation at least doesn't crash? We could have
a rule that all devices that do DMA must always postpone
all of that DMA to a bottom-half, but that's a lot of
refactoring of a lot of device code...

thanks
-- PMM

Jason Wang (jasowang) wrote on 2020-07-21: #6

On 2020/7/21 下午8:31, Peter Maydell wrote:
> On Wed, 15 Jul 2020 at 09:36, Jason Wang <email address hidden> wrote:
>> I think the point is to make DMA to MMIO work as real hardware.
> I wouldn't care to give a 100% guarantee that asking a real
> h/w device to DMA to itself didn't cause it to misbehave :-)
> It's more likely to happen-to-work because the DMA engine bit
> of a real h/w device is going to be decoupled somewhat from
> the respond-to-memory-transactions-for-registers logic, but
> it probably wasn't something the designers were actively
> thinking about either...

I think some device want such peer to peer transactions:

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/
tree/Documentation/driver-api/pci/p2pdma.rst

>
>> For
>> e1000e and other networking devices we need make sure such DMA doesn't
>> break anything.
> Yeah, this is the interesting part for QEMU. How should we
> structure devices that do DMA so that we can be sure that
> the device emulation at least doesn't crash? We could have
> a rule that all devices that do DMA must always postpone
> all of that DMA to a bottom-half, but that's a lot of
> refactoring of a lot of device code...

It looks to me the issue happens only for device with loopback

Simply git grep loopback in hw/net tells me we probably need only to
audit dp8393x and rt18139.

Qiang, want to help to audit those devices?

Thanks

>
> thanks
> -- PMM
>

Peter Maydell (pmaydell) wrote on 2020-07-21: #7

On Tue, 21 Jul 2020 at 14:21, Jason Wang <email address hidden> wrote:
> On 2020/7/21 下午8:31, Peter Maydell wrote:
> > On Wed, 15 Jul 2020 at 09:36, Jason Wang <email address hidden> wrote:
> >> I think the point is to make DMA to MMIO work as real hardware.
> > I wouldn't care to give a 100% guarantee that asking a real
> > h/w device to DMA to itself didn't cause it to misbehave :-)
> > It's more likely to happen-to-work because the DMA engine bit
> > of a real h/w device is going to be decoupled somewhat from
> > the respond-to-memory-transactions-for-registers logic, but
> > it probably wasn't something the designers were actively
> > thinking about either...

> I think some device want such peer to peer transactions:
>
> https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/
tree/Documentation/driver-api/pci/p2pdma.rst

That's a device DMAing to another device, not DMAing to *itself*
(device-to-another-device DMA should work fine in QEMU). And only
a very few devices will ever be sensible targets of the DMA --
basically things like nvme that have a looks-like-memory area,
or special cases like doorbell registers.

> > Yeah, this is the interesting part for QEMU. How should we
> > structure devices that do DMA so that we can be sure that
> > the device emulation at least doesn't crash? We could have
> > a rule that all devices that do DMA must always postpone
> > all of that DMA to a bottom-half, but that's a lot of
> > refactoring of a lot of device code...
>
>
> It looks to me the issue happens only for device with loopback

I think in principle we have a problem for any device that
(a) has memory mapped registers and (b) does DMA reads
whose address is guest-controlled. Loopback isn't a
requirement -- if the guest programs, say, an RX descriptor
base address to point at the device's own registers, you
get exactly the same kind of unexpected-reentrancy.

thanks
-- PMM

Alexander Bulekov (a1xndr) wrote on 2020-07-21: #8

On 200721 1444, Peter Maydell wrote:
[...]
I searched around but couldn't find anything talking about this case for
real hardware. I also looked at some HDL code for FPGAs that do DMA, but
it seems most of the PCI DMA components are contained in proprietary
IPs, though maybe I'm missing something (I've never programmed
a DMA-capable FPGA).

[...]

Could this be something that we check for in the
pci_dma_* functions in hw/pci/pci.h? There we still have context about
the source device for the dma read/write and could, compare addr against
the device's PCI BARr's. Not sure about:
1.) How to do this without the overhead of convering the addr
to a MemoryRegion, which is normally done, once, at the flatview_write
stage.
2.) What to do if we catch such a DMA request? Quietly drop it?
3.) Non-PCI devices.

I think this still doesn't cover the even crazier case where:
CPU writes to DEV_A's MMIO
DEV_A writes to DEV_B's MMIO
DEV_B writes to DEV_A's MMIO
and neither DEV_A or DEV_B use BHs...

-Alex

> thanks
> -- PMM
>

Jason Wang (jasowang) wrote on 2020-07-22: #9

On 2020/7/21 下午9:44, Peter Maydell wrote:
[...]

Well, my understanding is:

- it's not about whether or not we have an actual device that can do DMA
into itself but whether it's allowed by PCI spec
- it's not really matter whether or not it tries to DMA into itself.
Devices could be taught to DMA into each other's RX:

e1000e(1) RX DMA to e1000e(2) MMIO (RX)
e1000e(2) RX DMA to e1000e(1) RX

So we get re-reentrancy again.

[...]

Right, so about the solution, instead of refactoring DMA I wonder we can
simply detect and fail the RX by device itself.

Thanks

>
> thanks
> -- PMM
>

Jason Wang (jasowang) wrote on 2020-07-22: #10

On 2020/7/21 下午9:46, Li Qiang wrote:
[...]

Yes.

>> Simply git grep loopback in hw/net tells me we probably need only to
>> audit dp8393x and rtl8139.
>>
>> Qiang, want to help to audit those devices?
> No problem. Once I finish the e1000e patch I will try to audit those and
> also try to audit some no-loopback device re-entry issue.

Thanks.

>
> Thanks,
> Li Qiang
>
>> Thanks
>>
>>
>>> thanks
>>> -- PMM
>>>

Thomas Huth (th-huth) wrote on 2021-05-26: #11

I can reproduce this problem with QEMU v5.0, but with the current
version, it does not run into this assertion anymore. Seems like this
problem got fixed in the course of time? Could you please check whether
you could still reproduce this?

Changed in qemu:
status:New → Incomplete

Alexander Bulekov (a1xndr) wrote on 2021-06-14 (last edit on 2021-06-14): #12

This should have been fixed by the qemu_receive_packet/network loopback
patches from a few months ago

Thomas Huth (th-huth) wrote on 2021-06-15: #13

Ok, let's mark this as fixed.

Changed in qemu:
status:Incomplete → Fix Released

See full activity log

To post a comment you must log in.