Starred by 2 users

Owner: jannh@google.com

CC: proje...@google.com

Status: Fixed *(Closed)*

Components: ----

Modified: Sep 21, 2022

Deadline-90
Vendor-Linux
CCProjectZeroMembers
Severity-High
Finder-jannh
Product-Linux
Methodology-source-review
Reported-2022-Aug-12
Fixed-2022-Aug-21
CVE-2022-41222

Participant's Hotlists: linux-usermm-or-drivermm

Issue 2347: Linux stable 5.4/5.10: page UAF via stale TLB caused by rmap lock not held during PUD move

Reported by jannh@google.com on Thu, Aug 11, 2022, 10:12 PM EDT Project Member

The short version: Commit 97113eb39fa7972722ff490b947d8af023e1f6a2 should've
been backported to 5.4 and 5.10, but wasn't. That commit fixes a race that
leads to a stale TLB entry, which can be used to get access to a freed page.

I've written a reproducer (see attachment) that manages to demonstrate access
to a freed page within seconds when run bare-metal on an AMD machine that
runs the Debian stable kernel
"5.10.0-16-amd64 #1 SMP Debian 5.10.127-2 (2022-07-23)".
When I boot with page_poison=1, it manages to see pages containing PAGE_POISON
within seconds:

```
$ ./mremap-zap-pmdmove
initializing uncached load cutoff...
shortest uncached load = 458
uncached load cutoff = 229
attempt 0
################################################  observed value 0xaaaaaaaaaaaaaaaa
attempt 1
attempt 2
attempt 3
attempt 4
attempt 5
attempt 6
attempt 7
attempt 8
attempt 9
attempt 10
attempt 11
attempt 12
attempt 13
attempt 14
attempt 15
attempt 16
attempt 17
attempt 18
attempt 19
attempt 20
attempt 21
attempt 22
attempt 23
attempt 24
attempt 25

attempt 26
attempt 27
attempt 28
```

attempt 28
attempt 29
attempt 30
attempt 31
attempt 32
attempt 33
attempt 34
attempt 35
attempt 36
attempt 37
attempt 38
observed value 0xaaaaaaaaaaaaaaaa
observed value 0xaaaaaaaaaaaaaaaa
observed value 0xaaaaaaaaaaaaaaaa
observed value 0xaaaaaaaaaaaaaaaa
observed value 0xaaaaaaaaaaaaaaaa
observed value 0xaaaaaaaaaaaaaaaa
observed value 0xaaaaaaaaaaaaaaaa
observed value 0xaaaaaaaaaaaaaaaa
observed value 0xaaaaaaaaaaaaaaaa
observed value 0xaaaaaaaaaaaaaaaa

I have attached backports of commit 97113eb39fa7972722ff490b947d8af023e1f6a2 to 5.4 and 5.10.

In theory, the core race condition only requires the ability to create some kind of writable file (like a memfd), mremap(), ftruncate(), and the ability to run on multiple threads at the same time, which means the security bug is theoretically reachable from inside something like the Chrome renderer sandbox.
However, my reproducer relies on more exotic syscalls to actually make the race work, like sched_setscheduler(), sched_setaffinity() and madvise(MADV_PAGEOUT).

This bug is subject to a 90-day disclosure deadline. If a fix for this issue is made available to users before the end of the 90-day deadline, this bug report will become public 30 days after the fix was made available. Otherwise, this bug report will become public at the deadline.
The scheduled deadline is 2022-11-10.

=== how the reproducer works ===
If you're not interested in how the reproducer works, you can stop reading here.

The core of the race condition involves three tasks A, B and C operating on the same file-backed VMA:

1. B: ftruncate() enters the victim PMD and

   zap_pte_range() loads *pmd via pte_offset_map_lock()
2. A: mremap()'s move_normal_pmd() moves the victim PMD
3. C: create a TLB entry from the victim PTE at the new address

3. C: create a TLB entry from the victim PTE at the new address
4. B: ftruncate()'s zap_pte_range() removes the victim PTE

To make the race window between steps 1 and 4 as wide as possible,
the victim PTE is the last PTE in the page table, and the 511 PTEs
before the victim PTE are swap PTEs, which are expensive to clean
up but don't influence the TLB flush range.
To be able to have swap PTEs, the VMA must be a private VMA.

One difficult part is: How can task C detect when the victim PTE
has been moved to the new address, so that it can instantly create
a TLB entry, and refresh the TLB entry if more TLB flushes occur
before the PTE is removed; but at the same time, avoid hitting the
page fault handler, which would block until mremap() has finished
because mremap() holds the mmap lock in write mode?
I solved that using misspeculation: I took the retpoline pattern
and replaced the speculation trap with some code that tries to
detect whether a given address is readable and contains a specific
value. This helper can then be used to continuously refresh the
TLB entry while first waiting for the PTE to appear, then waiting
for the page contents to change.

ftruncate() is holding the rmap lock throughout this race.
mremap() briefly takes the rmap lock in write mode when it sets up
the new VMA.
This means the start of the syscalls must be ordered as follows:

1. A: mremap() begins
2. A: mremap() takes and drops rmap lock in copy_vma()
3. B: ftruncate() begins
[continue as shown above: task B has to enter the victim PMD
  before task A does move_normal_pmd()]

To create a time window between copy_vma() and the move of the
victim PMD, we can add a series of other PMD in front of the
victim PMD that mremap() has to churn through before reaching
the victim PMD.

To order the ftruncate() start after mremap() passing through
copy_vma(), we can use procfs to monitor VmPTE.
(Although that's so slow that just using a delay might work
just as well, or even better...)

When mremap() has moved the victim PMD, we don't want it to try
to immediately exit, because would involve do_munmap(), which
would first call tlb_gather_mmu() -> inc_tlb_flush_pending(),
then block on the rmap lock. This would cause extra safety
flushes in ftruncate() on every tlb_finish_mmu().
So the reproducer ensures that there are a lot of dummy PMDs
(pointing to empty page tables) after the victim PMD to slow

down mremap().
But every time mremap() moves a PMD, it will do a full TLB
flush on the MM; so to make it stop, the mremap() is given

flush on the MM; so to make it stop, the mremap() is given SCHED_IDLE priority, and once the misspeculation-based detection tells us that mremap() moved the victim PMD, another task is woken that preempts mremap().

One more detail to watch out for is that, in order for zap_pte_range() to actually immediately free the victim page, the victim page must be on the real LRU, not on a percpu pagevec waiting for batched LRU insertion (otherwise the percpu pagevec would hold an extra reference).
To get the victim page off the pagevec, we can create a dummy VMA on the same CPU where the page was allocated, then destroy that VMA, which flushes the local pagevec to the LRU via lru_add_drain().

mremap-zap-pmdmove-clean.c
15.7 KB View Download

5.10-mm-mremap-hold-the-rmap-lock-in-write-mode-when-movi.patch
4.0 KB View Download

5.4-mm-mremap-hold-the-rmap-lock-in-write-mode-when-movi.patch
4.0 KB View Download

Comment 1 by jannh@google.com on Mon, Aug 29, 2022, 1:47 PM EDT Project Member

Status: Fixed (was: New)
Labels: Fixed-2022-Aug-21

Fixed in:

5.4.211 (2022-Aug-25)
5.10.137 (2022-Aug-21)

Comment 2 by jannh@google.com on Tue, Sep 20, 2022, 11:17 AM EDT Project Member

Labels: -Restrict-View-Commit

Comment 3 by jannh@google.com on Wed, Sep 21, 2022, 10:45 AM EDT Project Member

Labels: CVE-2022-41222