Talos Vulnerability Report

TALOS-2020-1050

F2fs-Tools F2fs.Fsck fsck_chk_orphan_node Code Execution Vulnerability

OCTOBER 14, 2020

CVE NUMBER

CVE-2020-6108

Summary

An exploitable code execution vulnerability exists in the fsck_chk_orphan_node functionality of F2fs-Tools F2fs.Fsck 1.13. A specially crafted f2fs filesystem can cause a heap buffer overflow resulting in a code execution. An attacker can provide a malicious file to trigger this vulnerability.

Tested Versions

F2fs-Tools F2fs.Fsck 1.13

Product URLs

https://git.kernel.org/pub/scm/linux/kernel/git/jaegeuk/f2fs-tools.git

CVSSv3 Score

8.2 - CVSS:3.0/AV:L/AC:L/PR:H/UI:N/S:C/C:H/I:H/A:H

CWE

CWE-131 - Incorrect Calculation of Buffer Size

Details

The f2fs-tools set of utilities is used specifically for creating, checking and fixing f2fs (Flash-Friendly File System) files, a file system that has been replacing ext4 more recently in embedded devices, as it was crafted with eMMC chips and sdcards in mind. Fsck.f2fs more specifically is the file-system checking binary for f2fs partitions, and is where this vulnerability lies.

Today's vulnerability deals with the ability of f2fs to recover files, more specifically in f2fs terms `orphan inodes`. For instance if a directory gets corrupted, the f2fs filesystem and f2fs.fsck in particular can recover the files within that directory even if the directory cannot be accessed through normal means. The data structure which stores this recovery information is called `f2fs_orphan_block` structs, and looks like so:

```
[~.~]> ptype struct f2fs_orphan_block
type = struct f2fs_orphan_block {
    __le32 ino[1020];      // [1]
    __le32 reserved;
    __le16 blk_addr;
    __le16 blk_count;
    __le32 entry_count;    // [2]
    __le32 check_sum;
}
```

At [1], an array of all the inodes of all the orphaned datablocks lives, while at [2], the amount of orphan datablocks is stored. To cut to the chase of this vulnerability, there is no check on the `entry_count` member of this struct. Knowing that, let us examine where the orphan nodes are read from disk:

```
int fsck_chk_orphan_node(struct f2fs_sb_info *sbi)
{
    u32 blk_cnt = 0;
    block_t start_blk, orphan_blkaddr, i, j;
    struct f2fs_orphan_block *orphan_blk, *new_blk;
    struct f2fs_super_block *sb = F2FS_RAW_SUPER(sbi);
    u32 entry_count;

    if (!is_set_ckpt_flags(F2FS_CKPT(sbi), CP_ORPHAN_PRESENT_FLAG))  // [1]
        return 0;

    start_blk = __start_cp_addr(sbi) + 1 + get_sb(cp_payload);       //sbi->cp_blkaddr (0x201)
    orphan_blkaddr = __start_sum_addr(sbi) - 1 - get_sb(cp_payload); //ckpt->cp_pack_start_sum
                                                                           // offset(0x8c)
```

At [1], there's a check that we control, and at the comments below that, we get the bounds of the orphan pages on disk, which is completely arbitrary. Each of these orphan pages is read directly into a `f2fs_orphan_block` as we will see shortly. Continuing on:

```
int fsck_chk_orphan_node(struct f2fs_sb_info *sbi){
  //[...]

  f2fs_ra_meta_pages(sbi, start_blk, orphan_blkaddr, META_CP);

  orphan_blk = calloc(BLOCK_SZ, 1);
  ASSERT(orphan_blk);

  new_blk = calloc(BLOCK_SZ, 1);
  ASSERT(new_blk);

  for (i = 0; i < orphan_blkaddr; i++) {  // [1]
      int ret = dev_read_block(orphan_blk, start_blk + i); // [2]
      u32 new_entry_count = 0;

      ASSERT(ret >= 0);
      entry_count = le32_to_cpu(orphan_blk->entry_count); // [3]

      for (j = 0; j < entry_count; j++) {   // [4]
          //[...]
```

At [1], we see a loop that is dependent on the amount of orphan pages in the filesystem being analyzed, and at [2], we start reading each of these pages into `f2fs_orphan_block` structures. At [3], we directly read the `orphan_blk->entry_count` member (i.e. how many orphan inodes there are) and at [4] we see a loop whose iteration amount depends on a user-controlled value. Already this should be ringing bells, as there is no validation in between [3] and [4]. Moving on:

```
      for (j = 0; j < entry_count; j++) {
          nid_t ino = le32_to_cpu(orphan_blk->ino[j]);  // [0]
          DBG(1, "[%3d] ino [0x%x]\n", i, ino);
          struct node_info ni;
          blk_cnt = 1;

          // [...]

          ret = fsck_chk_node_blk(sbi, NULL, ino, F2FS_FT_ORPHAN, TYPE_INODE, &blk_cnt, NULL); // [1]

          if (!ret)
              new_blk->ino[new_entry_count++] = orphan_blk->ino[j]; // [2]
          else if (ret && c.fix_on)
              FIX_MSG("[0x%x] remove from orphan list", ino);
          else if (ret)
              ASSERT_MSG("[0x%x] wrong orphan inode", ino);
      }
```

At [0], we can see an out-of-bounds read here, since `j` has an upper bound of `entry_count`, and it's important to note for purposes of exploitation that this out of bounds read pointer (which starts at the top of the orphan blocks's inodes) always advances, regardless of the value it reads. At [1], there is a sanity check on the inode block number read with `fsck_chk_node_blk`, and while this checking is very intensive and quite thorough, the validity of the inode numbers read is determined by the f2fs partition itself. But why does this sanity checking matter? Because at [2], we also have an out of bounds write that occurs, but the write only happens (and the pointer only advances) if the check at [1] is valid.

This interesting scenario is technically two different vulnerabilities and is a situation that is extremely exploitable. To start (in android) the read pointer is 0x1000 bytes before the write pointer, but since the validity of inodes is controlled by the partition, the read pointer can be moved ahead of the write pointer, allowing someone to effectively store arbitrary memory in unused heap areas. Then, since we also control the amount of orphan pages (not just the orphan inodes), we can reset the out of bounds pointers while keeping the read pointer before the write pointer, such that we can overwrite other values in memory with our desired value. For specific targets to overwrite, there are function pointers further down in the heap belonging to `dict_t` objects that get called inside `fsck_chk_node_blk` during these loops.

Additional note on the exploitation on Android:
In Google Pixel 3 running Android 10, the f2fs filesystem is used for the /data partition, and, due to the fstab configuration, f2fs.fsck is is always executed on boot on the /data partition. Moreover, since full-disk encryption has been deprecated in favor of file-based encryption, it is possible to corrupt metadata in a reproducible manner. This means that a vulnerability in f2fs.fsck would allow an attacker to gain privileges in its context during boot, which could be the first step to start a chain to maintain persistence on the device, bypassing Android verified boot. Such an attack would require either physical access to the Android device, or a temporary root access in a context that allows to write to block devices from the Android OS.

Crash Information

Program received signal SIGSEGV, Segmentation fault.

```
[^_^] SIGSEGV

***************************************************************************
x0         : 0x7f00000101                    | x18       : 0x7fb7c10000
x1         : 0x0                             | x19       : 0x1000
x2         : 0x1000                          | x20       : 0x7fb5a7a000
x3[X]      : 0x55555580e0                    | x21       : 0x7fb5a92000       //quota_ctx
x4[X]      : 0x55555580ed                    | x22       : 0x7f00000101
x5         : 0x7fb600a07c                    | x23       : 0x0
x6         : 0x203e2d20                      | x24[X]    : 0x555557a000
x7         : 0xa30203e                       | x25       : 0x7f
x8         : 0x2000                          | x26       : 0x25
x9         : 0xbc4190939fe90dbd              | x27       : 0x7fb6d8f020
x10        : 0x0                             | x28       : 0x0
x11[H]     : 0x555557a510                    | x29[S]    : 0x7ffffff320
x12        : 0x0                             | x30[X]    : 0x55555610ac
x13        : 0xce4dff                        | sp[S]     : 0x7ffffff300
x14        : 0x10                            | pc[X]     : 0x5555560990
x15[L]     : 0x7fb675b40a                    | cpsr      : 0x80000000
x16[X]     : 0x5555579458                    | fpsr      : 0x0
x17[L]     : 0x7fb673fe80                    | fpcr      : 0x0
***************************************************************************
0x5555560980 : stp      x22, x21, [sp,#-48]!
0x5555560984 : stp      x20, x19, [sp,#16]
0x5555560988 : stp      x29, x30, [sp,#32]
0x555556098c : add      x29, sp, #0x20
=>0x5555560990 : ldr    x19, [x0]
0x5555560994 : cmp      x19, x0
0x5555560998 : b.eq     0x55555609cc <dict_lookup+76>
0x555556099c : mov      x20, x0
0x55555609a0 : mov      x21, x1
0x55555609a4 : ldr      x8, [x20,#64]
***************************************************************************
#0  0x0000005555560990 in dict_lookup ()
#1  0x00000055555610ac in quota_data_add ()
#2  0x00000055555614a0 in quota_add_inode_usage ()
#3  0x000000555556f12c in fsck_chk_node_blk ()
#4  0x0000005555573178 in fsck_chk_orphan_node ()
#5  0x0000005555566a20 in main ()
***************************************************************************
```

Timeline

2020-05-08 - Vendor Disclosure

2020-07-02 - 60 day follow up

2020-07-20 - 90 day follow up

2020-10-14 - Zero day public release

CREDIT

Discovered by Lilith >_> of Cisco Talos.

© 2022 Cisco Systems, Inc. and/or its affiliates. All rights reserved. View our Privacy Policy.