Talos Vulnerability Report

TALOS-2020-1048

F2fs-Tools F2fs.Fsck init_node_manager Information Disclosure Vulnerability

OCTOBER 14, 2020

CVE NUMBER

CVE-2020-6106

Summary

An exploitable information disclosure vulnerability exists in the `init_node_manager` functionality of F2fs-Tools F2fs.Fsck 1.12 and 1.13. A specially crafted filesystem can be used to disclose information. An attacker can provide a malicious file to trigger this vulnerability.

Tested Versions

F2fs-Tools F2fs.Fsck 1.12
F2fs-Tools F2fs.Fsck 1.13

Product URLs

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

CVSSv3 Score

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

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.

One of the features of the f2fs filesystem is the NAT section, which is an array of `f2fs_nat_entry` structs:

```
struct f2fs_nat_entry {
    __u8 version;       /* latest version of cached nat entry */
    __le32 ino;     /* inode number */
    __le32 block_addr;  /* block address */
} __attribute__((packed));
```

These `f2fs_nat_entry` structs allows for extremely quick lookup of block addresses (i.e. physical location on disk), given either a `nid` (which is the index into the `f2fs_nat_entry` array), or an inode. One of the steps taken during the f2fs.fsck process is validating this NAT section, including the bitmap that is used for representing the validity of any given nat entry within the nat entry table. This functionality is implemented by the `init_node_manager` function:

```
int init_node_manager(struct f2fs_sb_info *sbi)
{
    struct f2fs_super_block *sb = F2FS_RAW_SUPER(sbi); // [1]
    struct f2fs_checkpoint *cp = F2FS_CKPT(sbi);       // [2]
    struct f2fs_nm_info *nm_i = NM_I(sbi);             // [3]
    unsigned char *version_bitmap;
    unsigned int nat_segs;

    // [...]

    nm_i->bitmap_size = __bitmap_size(sbi, NAT_BITMAP); // [4] // le32_to_cpu(ckpt->nat_ver_bitmap_bytesize);

    nm_i->nat_bitmap = malloc(nm_i->bitmap_size);
    if (!nm_i->nat_bitmap)
        return -ENOMkEM;
    version_bitmap = __bitmap_ptr(sbi, NAT_BITMAP);    // [5] // &ckpt->sit_nat_version_bitmap (or +offset); approx.
    if (!version_bitmap)
        return -EFAULT;

    /* copy version bitmap */
    memcpy(nm_i->nat_bitmap, version_bitmap, nm_i->bitmap_size); //[6]
    return f2fs_init_nid_bitmap(sbi);
}
```

It's important to note that most of the fields for generating the nat_bitmap stored in the `f2fs_nm_info` object at [3] come from the `f2fs_super_block` at [1] and the `f2fs_checkpoint` at [2]. These structs are taken directly from the filesystem and are run through a good deal of checking after being read from disk. Keeping on, at [4], we can see the size of our nat bitmap be directly taken from our `f2fs_checkpoint` object, which is then malloc'ed. The source of the `memcpy` at [6] is generated from the `__bitmap_ptr` function at [5] which is as follows:

```
static inline void *__bitmap_ptr(struct f2fs_sb_info *sbi, int flag)
{
    struct f2fs_checkpoint *ckpt = F2FS_CKPT(sbi); // [1]
    int offset;

    if (is_set_ckpt_flags(ckpt, CP_LARGE_NAT_BITMAP_FLAG)) {
        // [...]
        return &ckpt->sit_nat_version_bitmap + offset + chksum_size;
    }

    if (le32_to_cpu(F2FS_RAW_SUPER(sbi)->cp_payload) > 0) {
        if (flag == NAT_BITMAP)
            return &ckpt->sit_nat_version_bitmap;
        else
            return ((char *)ckpt + F2FS_BLKSIZE); // [2]
    } else {
        // [...]
        return &ckpt->sit_nat_version_bitmap + offset;
    }
}
```

The most important parts are shown above, we only really care about what the return value is. As shown, all possible code paths are offsets from the `f2fs_checkpoint` object at [1], which is the same `f2fs_checkpoint` talked about in the `init_node_manager` function. Furthermore, with the exception of [2], all return values also base from the `sit_nat_version_bitmap` member of our `f2fs_checkpoint` (offset +0xc0). The next step is to see exactly what this member is:

```
int get_valid_checkpoint(struct f2fs_sb_info *sbi)
{
    // [...]
   cp_payload = get_sb(cp_payload);   // [1]  // vvv-(0x5)
    if (cp_payload > F2FS_BLK_ALIGN(MAX_SIT_BITMAP_SIZE))
        return -EINVAL;

    cp_blks = 1 + cp_payload;
    sbi->ckpt = malloc(cp_blks * blk_size);  // [2]

    // [...]

    if (cp_blks > 1) {
        unsigned int i;
        unsigned long long cp_blk_no;

        cp_blk_no = get_sb(cp_blkaddr);
        if (cur_page == cp2)
            cp_blk_no += 1 << get_sb(log_blocks_per_seg);

        /* copy sit bitmap */
        for (i = 1; i < cp_blks; i++) {
            unsigned char *ckpt = (unsigned char *)sbi->ckpt;
            ret = dev_read_block(cur_page, cp_blk_no + i);   // [3]
            ASSERT(ret >= 0);
            memcpy(ckpt + i * blk_size, cur_page, blk_size); // [4]
        }
    }
    //[...]
}
```

At [1], the most important field for this function is read in, the `cp_payload`, which comes directly from the `f2fs_superblock` and the underlying disk. There is a check on it right below, the so the max `cp_payload` size is 0x5. The `f2fs_checkpoint` object mentioned before is actually malloc'ed at [2], in the total size being `cp_blks` * 0x1000. Thus, the only sizes we can have for our `f2fs_checkpoint` object are 0x1000, 0x2000, 0x3000, 0x4000, and 0x5000. The contents are read in from the blocks directly after the `f2fs_checkpoint` at [3], and then copied to our malloc'ed buffer at [4]. Looking at the actual `f2fs_checkpoint` struct now:

```
struct f2fs_checkpoint {
    __le64 checkpoint_ver;      /* checkpoint block version number */
    __le64 user_block_count;    /* # of user blocks */
    __le64 valid_block_count;   /* # of valid blocks in main area */
    __le32 rsvd_segment_count;  /* # of reserved segments for gc */
    __le32 overprov_segment_count;  /* # of overprovision segments */
    __le32 free_segment_count;  /* # of free segments in main area */

    /* information of current node segments */
    __le32 cur_node_segno[MAX_ACTIVE_NODE_LOGS];
    __le16 cur_node_blkoff[MAX_ACTIVE_NODE_LOGS];
    /* information of current data segments */
    __le32 cur_data_segno[MAX_ACTIVE_DATA_LOGS];
    __le16 cur_data_blkoff[MAX_ACTIVE_DATA_LOGS];
    __le32 ckpt_flags;      /* Flags : umount and journal_present */
    __le32 cp_pack_total_block_count;   /* total # of one cp pack */
    __le32 cp_pack_start_sum;   /* start block number of data summary */
    __le32 valid_node_count;    /* Total number of valid nodes */
    __le32 valid_inode_count;   /* Total number of valid inodes */
    __le32 next_free_nid;       /* Next free node number */
    __le32 sit_ver_bitmap_bytesize; /* Default value 64 */
    __le32 nat_ver_bitmap_bytesize; /* Default value 256 */  // [1]
    __le32 checksum_offset;     /* checksum offset inside cp block */
    __le64 elapsed_time;        /* mounted time */
    /* allocation type of current segment */
    unsigned char alloc_type[MAX_ACTIVE_LOGS];

    /* SIT and NAT version bitmap */
    unsigned char sit_nat_version_bitmap[1];  // [2]
} __attribute__((packed));

[o.O]> p/x sizeof(struct f2fs_checkpoint)
$3 = 0xc1
```

The only things that really matter are [1] the size of the nat bitmap `nat_ver_bitmap_size`, and the `sit_nat_version_bitmap` at [2], since if readers will recall, those were the members used during the initialization of the node manager. Coming full circle:

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

    nm_i->bitmap_size = __bitmap_size(sbi, NAT_BITMAP); // [1] // le32_to_cpu(ckpt->nat_ver_bitmap_bytesize);

    nm_i->nat_bitmap = malloc(nm_i->bitmap_size);
    if (!nm_i->nat_bitmap)
        return -ENOMkEM;
    version_bitmap = __bitmap_ptr(sbi, NAT_BITMAP);    // [2] // &ckpt->sit_nat_version_bitmap (or +offset); approx.
    if (!version_bitmap)
        return -EFAULT;

    /* copy version bitmap */
    memcpy(nm_i->nat_bitmap, version_bitmap, nm_i->bitmap_size); //[3]
    return f2fs_init_nid_bitmap(sbi);
```

To populate the `nm_i->nat_bitmap` at [3], the size utilized comes directly from `ckpt->nat_ver_bitmap_bytesize` at [1]. Also important to reiterate, the pointer at [2] that we read from corresponds directly with the 0x1000-0x5000 size chunk allocated within `get_valid_checkpoint`. So now the question becomes, where does the `ckpt->nat_ver_bitmap_bytesize` come from and what constraints does it have? As shown by "cscope", there's not really that many mentions in the first place:

```
Text string: nat_ver_bitmap_bytesize

  File          Line
0 f2fs.h         340 return le32_to_cpu(ckpt->nat_ver_bitmap_bytesize);
1 f2fs.h         361 le32_to_cpu(ckpt->nat_ver_bitmap_bytesize) : 0;
2 mount.c        388 DISP_u32(cp, nat_ver_bitmap_bytesize);
3 mount.c       1157 nat_bitmap_size = get_cp(nat_ver_bitmap_bytesize);
4 mount.c       1473 nm_i->bitmap_size = __bitmap_size(sbi, NAT_BITMAP); //  le32_to_cpu(ckpt->nat_ver_bitmap_bytesize);
```

Looking through all the above examples, the only real validation on the `nat_bitmap_size` field is appropriately in the `sanity_check_ckpt` function:

```
int sanity_check_ckpt(struct f2fs_sb_info *sbi){

    sit_segs = get_sb(segment_count_sit);
    fsmeta += sit_segs;
    nat_segs = get_sb(segment_count_nat); // [1]
    fsmeta += nat_segs;

    // [...]

    sit_bitmap_size = get_cp(sit_ver_bitmap_bytesize);
    nat_bitmap_size = get_cp(nat_ver_bitmap_bytesize);

    if (sit_bitmap_size != ((sit_segs / 2) << log_blocks_per_seg) / 8 ||
        nat_bitmap_size != ((nat_segs / 2) << log_blocks_per_seg) / 8) {   //[2]
        MSG(0, "\tWrong bitmap size: sit(%u), nat(%u)\n",
                sit_bitmap_size, nat_bitmap_size);
        return 1;
    }

[...]
```

So in the end, the only real checking [2] is to make sure that the `sit_ver_bitmap_bytesize` corresponds correctly to the superblock's `segment_count_nat` member at [1]. Next we examine the `segment_count_nat` field for its constraints:

```
static inline int sanity_check_area_boundary(struct f2fs_super_block *sb, enum SB_ADDR sb_addr) {

    // [...]

    if (sit_blkaddr + (segment_count_sit << log_blocks_per_seg) !=  nat_blkaddr) {  // [1]
        MSG(0, "\tWrong SIT boundary, start(%u) end(%u) blocks(%u)\n",
            sit_blkaddr, nat_blkaddr,
            segment_count_sit << log_blocks_per_seg);
        return -1;
    }

    if (nat_blkaddr + (segment_count_nat << log_blocks_per_seg) != ssa_blkaddr) {  // [2]
        MSG(0, "\tWrong NAT boundary, start(%u) end(%u) blocks(%u)\n",
            nat_blkaddr, ssa_blkaddr,
            segment_count_nat << log_blocks_per_seg);
        return -1;
    }

    // [...]
```

At [1], we can see there's one check to make sure that the previous SIT section lines up with our nat section, and at [2], we can see that the end of the nat section is also checked to see if it's lined up correctly against the SSA section. Aside from that, there's no check on the size of the NAT segment itself. Back to `init_node_manager` for emphasis:

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

    nm_i->bitmap_size = __bitmap_size(sbi, NAT_BITMAP); // [1] // le32_to_cpu(ckpt->nat_ver_bitmap_bytesize);

    nm_i->nat_bitmap = malloc(nm_i->bitmap_size);
    if (!nm_i->nat_bitmap)
        return -ENOMkEM;
    version_bitmap = __bitmap_ptr(sbi, NAT_BITMAP);    // [2] // &ckpt->sit_nat_version_bitmap (or +offset); approx.
    if (!version_bitmap)
        return -EFAULT;

    /* copy version bitmap */
    memcpy(nm_i->nat_bitmap, version_bitmap, nm_i->bitmap_size); //[3]
    return f2fs_init_nid_bitmap(sbi);
```

Since the `nm_i->bitmap_size` variable [1] is not really meaningfully limited in magnitude, and also since we know the size of our `version_bitmap` heap chunk [2] as 0x1000,0x2000,0x3000,0x4000, or 0x5000, the out of bounds heap read at [3] becomes apparent, allowing us to propagate our `nm_i->nat_bitmap` field with out of bounds heap data for further use.

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. 0x00007f6eda62206f in ?? () from /lib/x86_64-linux-gnu/libc.so.6

```
[^_^] SIGSEGV

***********************************************************************************
***********************************************************************************
rax        : 0x7f6ed9a76010          | r13[S]     : 0x7ffc94046b70
rbx        : 0x0                                  | r14        : 0x0
rcx        : 0x102490                       | r15        : 0x0
rdx        : 0x1200c0                       | rip[L]     : 0x7f6eda62206f
rsi[H]     : 0x5581585f3000         | eflags     : 0x10202
rdi        : 0x7f6ed9a93c40           | cs         : 0x33
rbp[S]     : 0x7ffc940468c0         | ss         : 0x2b
rsp[S]     : 0x7ffc94046868         | ds         : 0x0
r8         : 0xffffffffffffffff              | es         : 0x0
r9         : 0x5581586f5490          | fs         : 0x0
r10        : 0x22                               | gs         : 0x0
r11        : 0x9                                 | fs_base    : 0x7f6edb2fa840
r12        : 0x558157cc7800    (_start) | gs_base    : 0x0
***********************************************************************************
0x7f6eda62205f : lea    r9,[rsi+rdx*1]
0x7f6eda622063 : cmp    rdi,r9
0x7f6eda622066 : jb     0x7f6eda622231
0x7f6eda62206c : mov    rcx,rdx
=>0x7f6eda62206f : rep movs BYTE PTR es:[rdi],BYTE PTR ds:[rsi]
0x7f6eda622071 : ret
0x7f6eda622072 : cmp    dl,0x10
0x7f6eda622075 : jae    0x7f6eda62208e
0x7f6eda622077 : cmp    dl,0x8
0x7f6eda62207a : jae    0x7f6eda6220a3
***********************************************************************************
#0 0x00007f6eda62206f: in memcpy (0x7f6eda4f9000 0x7f6eda68e000 0x195000 0x0 /lib/x86_64-linux-gnu/libc-2.24.so)
#1  0x0000558157cdb6b6 in init_node_manager (sbi=0x558157efcd60 <gfsck>) at mount.c:1483
#2  0x0000558157cdb705 in build_node_manager (sbi=0x558157efcd60 <gfsck>) at mount.c:1494
#3  0x0000558157ce0ce6 in f2fs_do_mount (sbi=0x558157efcd60 <gfsck>) at mount.c:3317
#4  0x0000558157cc9c61 in main (argc=3, argv=0x7ffc94046b78) at main.c:800
***********************************************************************************

[o.o]> x/10gx 0x5581585f3000
0x5581585f3000: Cannot access memory at address 0x5581585f3000
```

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