Talos Vulnerability Report

TALOS-2020-1046

F2fs-Tools F2fs.Fsck filesystem checking Information Disclosure Vulnerability

OCTOBER 14, 2020

CVE NUMBER

CVE-2020-6104

Summary

An exploitable information disclosure vulnerability exists in the get_dnode_of_data functionality of F2fs-Tools F2fs.Fsck 1.13. A specially crafted f2fs filesystem can cause information disclosure resulting in a information disclosure. 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

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

CWE

CWE-125 - Out-of-bounds Read

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.The initial list of `f2fs_nat_entries` comes straight from disk and is populated in the `build_nat_area_bitmap` function:

```
void build_nat_area_bitmap(struct f2fs_sb_info *sbi)
{
    // [...]

    nat_block = (struct f2fs_nat_block *)calloc(BLOCK_SZ, 1);
    ASSERT(nat_block);

    /* Alloc & build nat entry bitmap */                // 0x9
    nr_nat_blks = (get_sb(segment_count_nat) / 2) << sbi->log_blocks_per_seg;

                                        // * 0x1c7
    fsck->nr_nat_entries = nr_nat_blks * NAT_ENTRY_PER_BLOCK;
    fsck->nat_area_bitmap_sz = (fsck->nr_nat_entries + 7) / 8;
    fsck->nat_area_bitmap = calloc(fsck->nat_area_bitmap_sz, 1);
    ASSERT(fsck->nat_area_bitmap);
                                //sizeof==0x9
    fsck->entries = calloc(sizeof(struct f2fs_nat_entry), fsck->nr_nat_entries); // [1]

    ASSERT(fsck->entries);

    // [...]
```

The main thing to note from this function is that the `fsck->entries` buffer is used for storing all the nat entries, and also that it's a fixed and arbitrary size based on the f2fs input [1]. When utilizing this data and doing nat lookups, the `get_node_info` function is called:

```
void get_node_info(struct f2fs_sb_info *sbi, nid_t nid, struct node_info *ni)
{
    struct f2fs_nat_entry raw_nat;

    ni->nid = nid;
    if (c.func == FSCK) {
        node_info_from_raw_nat(ni, &(F2FS_FSCK(sbi)->entries[nid])); // [1]
        if (ni->blk_addr)
            return;
        /* nat entry is not cached, read it */
    }

    get_nat_entry(sbi, nid, &raw_nat);  // checks journal, then sm_info
    node_info_from_raw_nat(ni, &raw_nat);
}
```

Thus, given a `nid`, the `get_node_function` will put the looked `f2fs_nat_entry` struct into the `node_inof *ni` parameter. For our purposes, we only care about the cached nat lookup at [1], `node_info_from_raw_nat`:

```
static inline void node_info_from_raw_nat(struct node_info *ni,
        struct f2fs_nat_entry *raw_nat)
{
    ni->ino = le32_to_cpu(raw_nat->ino);
    ni->blk_addr = le32_to_cpu(raw_nat->block_addr);
    ni->version = raw_nat->version;
}
```

As shown, it's a rather optimized function, not much else of note except that this function is called many times during execution. Moving on, let us now jump to another function of interest, `f2fs_read`, which is used to read a given file from the f2fs filesystem:

```
u64 f2fs_read(struct f2fs_sb_info *sbi, nid_t ino, u8 *buffer,
                    u64 count, pgoff_t offset){
// [...]
    /* Memory allocation for block buffer and inode. */
    blk_buffer = calloc(BLOCK_SZ, 2);
    ASSERT(blk_buffer);
    inode = (struct f2fs_node*)(blk_buffer + BLOCK_SZ);

    /* Read inode */
    get_node_info(sbi, ino, &ni);                    // [1]
    ASSERT(dev_read_block(inode, ni.blk_addr) >= 0); // [2]
    ASSERT(!S_ISDIR(le16_to_cpu(inode->i.i_mode)));
    ASSERT(!S_ISLNK(le16_to_cpu(inode->i.i_mode)));

    /* Adjust count with file length. */
    filesize = le64_to_cpu(inode->i.i_size);
    if (offset > filesize)
        count = 0;
    else if (count + offset > filesize)
        count = filesize - offset;
```

At [1], we can see the forementioned call to `get_node_info`, which allows the program to grab the underlying inode's data [2], given the inode number. Continuing on in `f2fs_read`:

```
    /* Main loop for file blocks */
    read_count = remained_blkentries = 0;
    while (count > 0) {
        if (remained_blkentries == 0) {   //[1]
            set_new_dnode(&dn, inode, NULL, ino);
            get_dnode_of_data(sbi, &dn, F2FS_BYTES_TO_BLK(offset), LOOKUP_NODE);  //[2]
```

After calculating the size of our file, we enter the while loop at [1], which is used to read in the complete contents of a given inode that might span many blocks. At [2], f2fs.fsck will grab the `dnode_of_data` struct with the aptly named `get_dnode_of_data` function, which is a wrapper object that corresponds to a given `f2fs_inode` to the `f2fs_node` that it points to. The layout of the `dnode_of_data` struct is as such:

```
struct dnode_of_data {
    struct f2fs_node *inode_blk;    /* inode page */
    struct f2fs_node *node_blk; /* cached direct node page */
    nid_t nid;
    unsigned int ofs_in_node;
    block_t data_blkaddr;
    block_t node_blkaddr;
    int idirty, ndirty;
};
```

We must now examine the `get_dnode_of_data` function since the process of going from inode to corresponding data is not that simple. A given inode contains the following two members of import:

```
struct f2fs_inode {
    // [...]
    union {
        // [...]
        __le32 i_addr[DEF_ADDRS_PER_INODE]; /* Pointers to data blocks */
    };
    __le32 i_nid[5];        /* direct(2), indirect(2),
                        double_indirect(1) node id */
}
```

The `i_addr` array consists of 923 `uint32_t`'s that represent block addresses, while the `i_nid` array points to other potential inodes of varying indirectness in case `0x1000*923` is not enough space for the data of the inode. Let us now examine `get_dnode_of_data`:

```
int get_dnode_of_data(struct f2fs_sb_info *sbi, struct dnode_of_data *dn,
                        pgoff_t index, int mode)
{
    int offset[4];
    unsigned int noffset[4];
    struct f2fs_node *parent = NULL;
    nid_t nids[4];
    block_t nblk[4];
    struct node_info ni;
    int level, i;
    int ret;

    level = get_node_path(dn->inode_blk, index, offset, noffset); // [1]

    nids[0] = dn->nid;
    parent = dn->inode_blk;
    if (level != 0)
        nids[1] = get_nid(parent, offset[0], 1); // [2]
    else
        dn->node_blk = dn->inode_blk;  // [3]

    get_node_info(sbi, nids[0], &ni);  // [4]
    nblk[0] = ni.blk_addr;
```

At [1], the `get_node_path` function looks to see how many levels of indirection must be traversed in order to find the appropriate block addresses of the data. If `levels == 0`, this implies that the block addresses we care about are within the inode that we are currently parsing and the lookup can stop at [3]. If the return value of `get_node_path` is greater than zero though, we then must look up the next `nid` in the chain at [2]. Moving on, we must look at exactly how `get_node_path` behaves:

```
static int get_node_path(struct f2fs_node *node, long block, int offset[4], unsigned int noffset[4]) {

    const long direct_index = ADDRS_PER_INODE(&node->i); // => addrs_per_inode // [1]
    const long direct_blks = ADDRS_PER_BLOCK;
    const long dptrs_per_blk = NIDS_PER_BLOCK;
    const long indirect_blks = ADDRS_PER_BLOCK * NIDS_PER_BLOCK;
    const long dindirect_blks = indirect_blks * NIDS_PER_BLOCK;
    int n = 0;
    int level = 0;

    noffset[0] = 0;
    if (block < direct_index) { // F2FS_BYTES_TO_BLOCK(offset) // [2]
        offset[n] = block;
        goto got;
    }

    block -= direct_index;
    if (block < direct_blks) { // direct_blks == 1018   // [3]
        offset[n++] = NODE_DIR1_BLOCK;
        noffset[n]= 1;
        offset[n] = block;
        level = 1;
        goto got;
    }
```

For the purposes of the vulnerabilty, we don't need to go further than this. The overall flow is that if the block (of the inode) we are trying to look up is less than the amount of blocks in our inode, the level of indirection is 0x0 and we can immediately just return the `uint32_t inode->i_addr[block]` [2]. If the block we want is greater than the amount of direct blocks the inode has, we hit the branch at [3]. The interesting part is that while an inode can only hold 923 direct blocks, there's still a function call to `addrs_per_inode` at [1]:

```
unsigned int addrs_per_inode(struct f2fs_inode *i)
{
    return CUR_ADDRS_PER_INODE(i) - get_inline_xattr_addrs(i);
}
```

While this doesn't exactly make things clearer as to how many direct blocks an inode has, it does show that, depending on the inode itself, the amount of direct blocks it has is not always 923. Looking further at `CUR_ADDRS_PER_INODE`:

```
#define CUR_ADDRS_PER_INODE(inode)  (DEF_ADDRS_PER_INODE -  __get_extra_isize(inode))
#define DEF_ADDRS_PER_INODE 923 /* Address Pointers in an Inode */

static inline int __get_extra_isize(struct f2fs_inode *inode)
{
    if (f2fs_has_extra_isize(inode))
        return le16_to_cpu(inode->i_extra_isize) / sizeof(__le32);
    return 0;
}
```

After all the macros we can see that `CUR_ADDRS_PER_INODE` just boils down to `(923 - (inode->i_extra_size / 4))`, rather simple. It's worth noting that the `inode->i_extra_size` is read directly from disk and doesn't really have much sanitation, aside from an upper-bound of `4*923` (0xe6c) checked in `fsck_chk_inode_blk`. Continuing, the other part of the `addrs_per_inode` function we care about is `get_inline_xattr_addrs`:

```
static inline int get_inline_xattr_addrs(struct f2fs_inode *inode)
{
    if (c.feature & cpu_to_le32(F2FS_FEATURE_FLEXIBLE_INLINE_XATTR))
        return le16_to_cpu(inode->i_inline_xattr_size);
    else if (inode->i_inline & F2FS_INLINE_XATTR || inode->i_inline & F2FS_INLINE_DENTRY)  // [1]
        return DEFAULT_INLINE_XATTR_ADDRS;  // 50
    else  // [2]
        return 0;
}
```

Since `c.feature` is something we cannot control under normal circumstances, we can only hit branches [1] and [2]. Since we control the `inode->i_inline` struct member, we can decide to just have this function return `DEFAULT_INLIINE_XATTR_ADDRS`, which is 50. Thus, backing up to a `addrs_per_inode`:

```
unsigned int addrs_per_inode(struct f2fs_inode *i)
{
    return CUR_ADDRS_PER_INODE(i) - get_inline_xattr_addrs(i);

    // (923 - (inode->i_extra_size / 4)) - 50  // [1]
}
```

The comment at [1] is a final reduction of what is actually going on in this function, and since we control `inode->i_extra_size`, we can also control exactly what is returned here. Jumping back up to `get_node_path`:

```
static int get_node_path(struct f2fs_node *node, long block, int offset[4], unsigned int noffset[4]) {

    const long direct_index = ADDRS_PER_INODE(&node->i); // -> user controlled
    const long direct_blks = ADDRS_PER_BLOCK;
    const long dptrs_per_blk = NIDS_PER_BLOCK;
    const long indirect_blks = ADDRS_PER_BLOCK * NIDS_PER_BLOCK;
    const long dindirect_blks = indirect_blks * NIDS_PER_BLOCK;
    int n = 0;
    int level = 0;

    noffset[0] = 0;
    if (block < direct_index) { // F2FS_BYTES_TO_BLOCK(offset) // [1]
        offset[n] = block;
        goto got;
    }

    block -= direct_index;     // 0 - 0
    if (block < direct_blks) { // [2]
        offset[n++] = NODE_DIR1_BLOCK;  // [3]
        noffset[n]= 1;
        offset[n] = block;
        level = 1;
        goto got;
    }

    // [...]
    got:
        return level;

}
```

Since `addrs_per_inode` is user-controlled, if we set the return value equal to the `block` argument, we actually end up avoiding the branch at [1], but since 0 is less than 1018, we take the branch at [2], resulting in the `level` variable being 0x1 which we then return back up to `get_dnode_of_data`. Also important to note is that at [3], the offset array gets assigned with the offset of our supposed block address (`NODE_DIR1_BLOCK` (1019)), since the program thinks the block is located inside an direct data block instead of the inode itself. Returning back to `get_dnode_of_data`:

```
int get_dnode_of_data(struct f2fs_sb_info *sbi, struct dnode_of_data *dn,
                        pgoff_t index, int mode)
{
// [...]

level = get_node_path(dn->inode_blk, index, offset, noffset); // returns 0

nids[0] = dn->nid;
parent = dn->inode_blk;
if (level != 0)
    nids[1] = get_nid(parent, offset[0], 1); // [0]
else
    dn->node_blk = dn->inode_blk;
```

Since `get_node_path` returns 0x1 (even thought the block offset we looked up is less that the max amount of blocks it has), we end up hitting the code path at [0]:

```
static inline nid_t get_nid(struct f2fs_node * rn, int off, int i)
{
    if (i)                            // 1019 - 1019
        return le32_to_cpu(rn->i.i_nid[off - NODE_DIR1_BLOCK]);
    else
        return le32_to_cpu(rn->in.nid[off]);
}
```

The `off` param is the `NODE_DIR1_BLOCK` parameter from before, which results in the return of our `inode->i_nid` essentially, which is a completely arbitrary value and is never checked in this code path. Continuing in `get_dnode_of_data`:

```
int get_dnode_of_data(struct f2fs_sb_info *sbi, struct dnode_of_data *dn,
                        pgoff_t index, int mode)
{
     // [...]
            nids[1] = get_nid(parent, offset[0], 1); // [0]
     // [...]

    get_node_info(sbi, nids[0], &ni);
    nblk[0] = ni.blk_addr;

    for (i = 1; i <= level; i++) {
        if (!nids[i] && mode == ALLOC_NODE) {
            // [...]
        } else {
            /* If Sparse file no read API, */
            struct node_info ni;

            get_node_info(sbi, nids[i], &ni); // [2]
            dn->node_blk = calloc(BLOCK_SZ, 1);
            ASSERT(dn->node_blk);

            ret = dev_read_block(dn->node_blk, ni.blk_addr);
            ASSERT(ret >= 0);

            nblk[i] = ni.blk_addr;
        }
```

At [0], the arbitrary value from `inode->i_nid[0]` gets put into the `nids[1]` variable. At this point `nids[0]` is the original quota inode that started this all, and `nids[1]` is arbitrary, but is supposed to be the nid of a data block (consisting of all `blk_addrs`) where our data is located. The program then tries to look up all blocks in the lookup chain, and subsequently has to resolve `nids[1]` to a block address at [2]. Let us now show `get_node_info` with `nids[1]` being arbitrary:

```
void get_node_info(struct f2fs_sb_info *sbi, nid_t nid, struct node_info *ni)
{
    struct f2fs_nat_entry raw_nat;

    ni->nid = nid;
    if (c.func == FSCK) {
        node_info_from_raw_nat(ni, &(F2FS_FSCK(sbi)->entries[nid])); // [0]
```

After all the above, we finally hit [0] with an arbitrary nid, resulting in an out-of-bounds heap read that casts heap data as a nat entry. This can be used in a couple of ways, depending on how hardened the target executing f2fs.fsck is. If dealing with an Android 10 device, one must consider that the `fsck->entries` malloc is always too big to be malloc'ed without mmap (even though it is user-controlled). And since it will always be mmap'ed, Android 10 will insert a randomized gap in between the mmap'ed memory and the normal heap, which means that one cannot really utilize this for defeating address space randomization on Android 10. The bug is not completely useless in this case though, as the ability to cast an arbitrary nid allows one to bypass all of the nid, `block_address`, and nat sanitation that occurs before any of this happens.

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

[^_^] SIGSEGV

```
************************************************************************************
************************************************************************************
x0[X]      : 0x555557a000               | x18        : 0x7fb791a000
x1         : 0xdeadface                       | x19[S]     : 0x7ffffff128
x2[S]      : 0x7ffffff128                    | x20        : 0xdeadface
x3         : 0x1                                     | x21[X]     : 0x555557a000
x4         : 0x7fb5c54000                  | x22        : 0x7fb5c53000
x5         : 0x4                                     | x23        : 0xdeadface
x6         : 0x0                                     | x24[S]     : 0x7ffffff148
x7         : 0x162000                              | x25        : 0x7fb6d8f020
x8[H]      : 0x555557a140               | x26        : 0x1
x9         : 0x877f2a593e                  | x27[S]     : 0x7ffffff178
x10        : 0x7d41dd13e                  | x28        : 0x8
x11[S]     : 0x7ffffff178                  | x29[S]     : 0x7ffffff110
x12        : 0x0                                    | x30[X]     : 0x555556456c
x13        : 0xce4dff                            | sp[S]      : 0x7ffffff0b0
x14        : 0x52ad22605f5bd9af     | pc[X]      : 0x555556a000     <get_node_info+96>
x15        : 0x68                                  | cpsr       : 0x60000000
x16[X]     : 0x5555579450              | fpsr       : 0x0
x17[L]     : 0x7fb6716620               | fpcr       : 0x0
************************************************************************************
0x5555569ff0 : mov      w10, w20
0x5555569ff4 : add      x10, x10, w20, uxtw #3
0x5555569ff8 : ldr      x9, [x9,#288]
0x5555569ffc : add      x9, x9, x10
=>0x555556a000 : ldur   w10, [x9,#1]
0x555556a004 : str      w10, [x19,#4]
0x555556a008 : ldur     w10, [x9,#5]
0x555556a00c : str      w10, [x19,#8]
0x555556a010 : ldrb     w9, [x9]
0x555556a014 : strb     w9, [x19,#12]
************************************************************************************
#0  0x000000555556a000 in get_node_info ()
#1  0x000000555556456c in get_dnode_of_data ()
#2  0x0000005555564c38 in f2fs_read ()
#3  0x00000055555561ba8 in quota_read_nomount ()
#4  0x0000005555563b74 in v2_check_file ()
#5  0x0000005555561944 in quota_file_open ()
#6  0x0000005555561510 in quota_compare_and_update ()
#7  0x00000055555735d4 in fsck_chk_quota_files ()
#8  0x0000005555566a48 in main ()
************************************************************************************

************************************************************************************
rax        : 0x7f19a962f14e                     | r13[S]     : 0x7ffd70feba90
rbx        : 0x0                                            | r14        : 0x0
rcx        : 0x7f11d5452010                    | r15        : 0x0
rdx        : 0x7f19a962f14e                     | rip        : 0x559c0a4c5319 <node_info_from_raw_
rsi        : 0x7f19a962f14e                      | eflags     : 0x202
rdi[S]     : 0x7ffd70feb4b0                     | cs         : 0x33
rbp[S]     : 0x7ffd70feb440                    | ss         : 0x2b
rsp[S]     : 0x7ffd70feb440                    | ds         : 0x0
r8         : 0x3                                              | es         : 0x0
r9         : 0x2010                                       | fs         : 0x0
r10        : 0x53                                           | gs         : 0x0
r11[L]     : 0x7f11dddcada0                  | fs_base    : 0x7f11de82a840
r12        : 0x559c0a4b6800                  | gs_base    : 0x0
************************************************************************************
0x559c0a4c530a : mov    rbp,rsp
0x559c0a4c530d : mov    QWORD PTR [rbp-0x8],rdi
0x559c0a4c5311 : mov    QWORD PTR [rbp-0x10],rsi
0x559c0a4c5315 : mov    rax,QWORD PTR [rbp-0x10]
=>0x559c0a4c5319 : mov    edx,DWORD PTR [rax+0x1]
0x559c0a4c531c : mov    rax,QWORD PTR [rbp-0x8]
0x559c0a4c5320 : mov    DWORD PTR [rax+0x4],edx
0x559c0a4c5323 : mov    rax,QWORD PTR [rbp-0x10]
0x559c0a4c5327 : mov    edx,DWORD PTR [rax+0x5]
0x559c0a4c532a : mov    rax,QWORD PTR [rbp-0x8]
************************************************************************************
540                     struct f2fs_nat_entry *raw_nat)
541     {
542             ni->ino = le32_to_cpu(raw_nat->ino);    <[^.^]
543             ni->blk_addr = le32_to_cpu(raw_nat->block_addr);
544             ni->version = raw_nat->version;
545     }
546
************************************************************************************
#0  0x0000559c0a4c5319 in node_info_from_raw_nat (ni=0x7ffd70feb4b0, raw_nat=0x7f19a962f14e) at f2fs.h:542
#1  0x0000559c0a4cc409 in get_node_info (sbi=0x559c0a6ebd60 <gfsck>, nid=3735943886, ni=0x7ffd70feb4b0) at mount.c:2114
#2  0x0000559c0a4d3c3b in get_dnode_of_data (sbi=0x559c0a6ebd60 <gfsck>, dn=0x7ffd70feb590, index=0, mode=1) at node.c:234
#3  0x0000559c0a4d48ce in f2fs_read (sbi=0x559c0a6ebd60 <gfsck>, ino=4, buffer=0x7ffd70feb684 "", count=8, offset=0) at segment.c:167
#4  0x0000559c0a4daec2 in quota_read_nomount (qf=0x7ffd70feb750, offset=0, buf=0x7ffd70feb684, size=8) at quotaio.c:91
#5  0x0000559c0a4dd09b in v2_read_header (h=0x7ffd70feb740, dqh=0x7ffd70feb684) at quotaio_v2.c:141
#6  0x0000559c0a4dd0e5 in v2_check_file (h=0x7ffd70feb740, type=0) at quotaio_v2.c:157
#7  0x0000559c0a4db08e in quota_file_open (sbi=0x559c0a6ebd60 <gfsck>, h=0x7ffd70feb740, qtype=USRQUOTA, flags=0) at quotaio.c:137
#8  0x0000559c0a4dab1c in quota_compare_and_update (sbi=0x559c0a6ebd60 <gfsck>, qtype=USRQUOTA, usage_inconsistent=0x7ffd70feb810,
preserve_limits=1) at mkquota.c:378
#9  0x0000559c0a4bf0e6 in fsck_chk_quota_files (sbi=0x559c0a6ebd60 <gfsck>) at fsck.c:1811
#10 0x0000559c0a4b85cb in do_fsck (sbi=0x559c0a6ebd60 <gfsck>) at main.c:655
#11 0x0000559c0a4b8cd4 in main (argc=3, argv=0x7ffd70feba98) at main.c:811
************************************************************************************
```

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