Talos Vulnerability Report

TALOS-2020-0988

F2fs-tools fsck.f2fs sanity_check_area_boundary code execution vulnerability

APRIL 9, 2020

CVE NUMBER

CVE-2020-6070

Summary

An exploitable code execution vulnerability exists in the file system checking functionality of fsck.f2fs 1.12.0. A specially crafted f2fs file can cause a logic flaw and out-of-bounds heap operations, resulting in code execution. An attacker can provide a malicious file to trigger this vulnerability.

Tested Versions

F2fs-tools-1.12.0

Product URLs

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

CVSSv3 Score

6.7 - CVSS:3.0/AV:L/AC:L/PR:H/UI:N/S:U/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.

At the top level of processing, the first structure read off of the disk (in order to check and fix it), the superblock is read off the disk:

```
int f2fs_do_mount(struct f2fs_sb_info *sbi)
{
    struct f2fs_checkpoint *cp = NULL;
    struct f2fs_super_block *sb = NULL;
    int ret;

    sbi->active_logs = NR_CURSEG_TYPE;
    ret = validate_super_block(sbi, SB0_ADDR);      // [0]
    if (ret) {
        ret = validate_super_block(sbi, SB1_ADDR);  // [1]
        if (ret)
            return -1;
    }
```

At [0] we read 0x1000 bytes from offset 0x0 of the disk, and if validation of those bytes fail we attempt the same at offset 0x1000 [1]. Looking at the individual reads and validations:

```
int validate_super_block(struct f2fs_sb_info *sbi, enum SB_ADDR sb_addr)
{
    char buf[F2FS_BLKSIZE]; // 0x1000                                    // [1]

    sbi->raw_super = malloc(sizeof(struct f2fs_super_block)); // 0xc00   // [2]
    if (!sbi->raw_super)
        return -ENOMEM;

    if (dev_read_block(buf, sb_addr))
        return -1;
                             // 0x400
    memcpy(sbi->raw_super, buf + F2FS_SUPER_OFFSET,
                sizeof(struct f2fs_super_block));

    if (!sanity_check_raw_super(sbi->raw_super, sb_addr)) {             // [3]
```

The `validate_super_block` function reads the superblock to the len(0x1000) stack buffer [1], and takes the bottom `0xC00` bytes of that buffer and places it into the `sbi->raw_super` object allocated at [2], assuming the bytes pass the superblock validation at [3]. The `sanity_check_raw_super` has quite a few checks, but they all come directly from the disk itself, so it should suffice to summarize the checks (since they all result in returning -1 or 1 if they fail):

```
int sanity_check_raw_super(struct f2fs_super_block *sb, enum SB_ADDR sb_addr) {
    [...]
    if ((get_sb(feature) & F2FS_FEATURE_SB_CHKSUM) && verify_sb_chksum(sb))
    if (F2FS_SUPER_MAGIC != get_sb(magic)) {
    if (F2FS_BLKSIZE != PAGE_CACHE_SIZE) {

    blocksize = 1 << get_sb(log_blocksize);
    if (F2FS_BLKSIZE != blocksize) {

    if (get_sb(log_blocks_per_seg) != 9) {

    if (get_sb(log_sectorsize) > F2FS_MAX_LOG_SECTOR_SIZE || get_sb(log_sectorsize) < F2FS_MIN_LOG_SECTOR_SIZE) {

    if (get_sb(log_sectors_per_block) + get_sb(log_sectorsize) != F2FS_MAX_LOG_SECTOR_SIZE) {

    segment_count = get_sb(segment_count);
    segs_per_sec = get_sb(segs_per_sec);
    secs_per_zone = get_sb(secs_per_zone);
    total_sections = get_sb(section_count);
    blocks_per_seg = 1 << get_sb(log_blocks_per_seg);

    if (segment_count > F2FS_MAX_SEGMENT || segment_count < F2FS_MIN_SEGMENTS) {
    if (total_sections > segment_count || total_sections < F2FS_MIN_SEGMENTS || segs_per_sec > segment_count || !segs_per_sec) {
    if ((segment_count / segs_per_sec) < total_sections) {
    if (segment_count > (get_sb(block_count) >> 9)) {
    if (secs_per_zone > total_sections || !secs_per_zone) {
    if (get_sb(extension_count) > F2FS_MAX_EXTENSION || sb->hot_ext_count > F2FS_MAX_EXTENSION || get_sb(extension_count) + sb-
>hot_ext_count > F2FS_MAX_EXTENSION) {
    if (get_sb(cp_payload) > (blocks_per_seg - F2FS_CP_PACKS)) {
    if (get_sb(node_ino) != 1 || get_sb(meta_ino) != 2 || get_sb(root_ino) != 3) {
    if (c.devices[0].zoned_model == F2FS_ZONED_HM && !(sb->feature & cpu_to_le32(F2FS_FEATURE_BLKZONED))) {

    if (sanity_check_area_boundary(sb, sb_addr)) // [1]
        return -1;

    return 0;
}
```

Needless to say, there's quite a few sanity checks on the superblock. For context, all `get_sb` function does is pull the named variable out of the superblock struct which will be listed eventually. But quick note before that, the only check we really care about is the `sanity_check_area_boundary` function [1], which we look at now:

```
static inline int sanity_check_area_boundary(struct f2fs_super_block *sb, enum SB_ADDR sb_addr) {
    u32 segment0_blkaddr = get_sb(segment0_blkaddr);
    u32 cp_blkaddr = get_sb(cp_blkaddr);                    // [1]
    u32 sit_blkaddr = get_sb(sit_blkaddr);                  // [2]
    u32 nat_blkaddr = get_sb(nat_blkaddr);
    u32 ssa_blkaddr = get_sb(ssa_blkaddr);
    u32 main_blkaddr = get_sb(main_blkaddr);
    u32 segment_count_ckpt = get_sb(segment_count_ckpt);    // [3]
    u32 segment_count_sit = get_sb(segment_count_sit);      // [4]
    u32 segment_count_nat = get_sb(segment_count_nat);
    u32 segment_count_ssa = get_sb(segment_count_ssa);
    u32 segment_count_main = get_sb(segment_count_main);
    u32 segment_count = get_sb(segment_count);
    u32 log_blocks_per_seg = get_sb(log_blocks_per_seg);
    u64 main_end_blkaddr = main_blkaddr +
            (segment_count_main << log_blocks_per_seg);
    u64 seg_end_blkaddr = segment0_blkaddr +
            (segment_count << log_blocks_per_seg);

    if (segment0_blkaddr != cp_blkaddr) {
        MSG(0, "\tMismatch segment0(%u) cp_blkaddr(%u)\n",
            segment0_blkaddr, cp_blkaddr);
        return -1;
    }

    if (cp_blkaddr + (segment_count_ckpt << log_blocks_per_seg) !=  sit_blkaddr) { // [5]
        MSG(0, "\tWrong CP boundary, start(%u) end(%u) blocks(%u)\n",
            cp_blkaddr, sit_blkaddr,
            segment_count_ckpt << log_blocks_per_seg);
        return -1;
    }

    if (sit_blkaddr + (segment_count_sit << log_blocks_per_seg) !=  nat_blkaddr) { // [6]
        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;
    }

    [...]
```

From a higher level, f2fs partitions maintain a set of different areas that serve different purposes. The `sanity_check_area_boundary` aptly checks all the definitions of these area, given a block addr for the start, and also a size. For the checkpoint area (used for fs restoration), we can see this at [1] and [3] above, likewise for the SIT (Segment Information Table) area (which deals with the validity of individual blocks) we see this at [2] and [4]. Looking at [5] and [6], we can gather that the expectation for the area layout is that the checkpoint area ends where the SIT area begins, and likewise the SIT area ends where the NAT area begins. This sequential layout is expected for all the areas listed above (`segment0_blkaddr`, `cp_blkaddr`, `sit_blkaddr`, `nat_blkaddr`, `ssa_blkaddr`, `main_blkaddr`).

The vulnerability in question is actually within the above function as well, so to continue on within the same function, but summarized:

```
    [...]
    if (segment0_blkaddr != cp_blkaddr) {                                          // [1]
    if (cp_blkaddr + (segment_count_ckpt << log_blocks_per_seg) != sit_blkaddr) { // [2]
    if (sit_blkaddr + (segment_count_sit << log_blocks_per_seg) != nat_blkaddr) { // [3]
    if (nat_blkaddr + (segment_count_nat << log_blocks_per_seg) != ssa_blkaddr) { // [4]
    if (ssa_blkaddr + (segment_count_ssa << log_blocks_per_seg) != main_blkaddr){ // [5]

    if (main_end_blkaddr > seg_end_blkaddr) {         // [6]
    } else if (main_end_blkaddr < seg_end_blkaddr) {
        set_sb(segment_count, (main_end_blkaddr -
                segment0_blkaddr) >> log_blocks_per_seg);

        update_superblock(sb, SB_MASK(sb_addr));
    }
    return 0;
}
```

At all the labels [1-6] above, a different area boundary is tested, as previously mentioned, and if any of them fail, the entire area superblock fails validation. There is however a very interesting edge case if one provides a superblock such that all of the `*_blkaddr` values and `segment_count_*` values are all null. The validation for all the above checks will pass since `0 + (0 << X) == 0` for all possible values of X, and we effectively overlay all of our areas on top of each other, leading to a very unstable state.

But what can be done from here? It should be noted there is quite a bit other checks to hit/pass before getting to these exploit vectors, but there are actually quite a few options. One possible exploitation vector (seen in f2fstools 1.12.0):

```
void build_sit_entries(struct f2fs_sb_info *sbi)
{
    struct sit_info *sit_i = SIT_I(sbi);
    struct curseg_info *curseg = CURSEG_I(sbi, CURSEG_COLD_DATA);
    struct f2fs_journal *journal = &curseg->sum_blk->journal; // [1]
    struct f2fs_sit_block *sit_blk;
    struct seg_entry *se;
    struct f2fs_sit_entry sit;
    unsigned int i, segno;

    sit_blk = calloc(BLOCK_SZ, 1);
    ASSERT(sit_blk);
    for (segno = 0; segno < TOTAL_SEGS(sbi); segno++) {    // [2]
        se = &sit_i->sentries[segno];

        get_current_sit_page(sbi, segno, sit_blk);
        sit = sit_blk->entries[SIT_ENTRY_OFFSET(sit_i, segno)];

        check_block_count(sbi, segno, &sit); // does literally nothing...
        seg_info_from_raw_sit(se, &sit);     // asdf
    }

    free(sit_blk);
    for (i = 0; i < sits_in_cursum(journal); i++) {        // [3]      //  (journal->n_sits)
        segno = le32_to_cpu(segno_in_journal(journal, i));             //  (jounal->sit_j.entries[i].segno)
        se = &sit_i->sentries[segno];         // [4]
        sit = sit_in_journal(journal, i);

        check_block_count(sbi, segno, &sit);  // [5]
        seg_info_from_raw_sit(se, &sit);      // [6]
    }

}
```

At [1], due to the unstable state of the overlaid areas, the `f2fs_journal` points to un-initialized memory. Assuming that the attacker can manipulate the heap and control the contents where `&curseg->sum_blk` points, we skip the loop at [2], due to `TOTAL_SEGS(sbi) == 0`, and end up entering a loop at [3] whose iteration count we control.

The loop tries to read from a chunk allocated in `build_sit_info` like so `sit_i->sentries = calloc(TOTAL_SEGS(sbi) * sizeof(struct seg_entry), 1);`, which is based on the `main_segments` area that was assigned 0x0 back in the sanity checking (`#define TOTAL_SEGS(sbi) (SM_I(sbi)->main_segments)`).

One interesting thing to note about the main_segment, is that for the most part, we actually can set this value to something other than 0x0 without failing the sanity checks because it is the last area parsed from our f2fs partition. In summary, the `sit_i->sentries` buffer is also completely user controlled, along with which index of the array for this buffer. [5] does absolutely nothing, so we then reach [6]:

```
seg_info_from_raw_sit(struct seg_entry *se, struct f2fs_sit_entry *raw_sit) {
            // #define GET_SIT_VBLOCKS(raw_sit)  (le16_to_cpu((raw_sit)->vblocks) & SIT_VBLOCKS_MASK)
    se->valid_blocks = GET_SIT_VBLOCKS(raw_sit);                       // [1]
    memcpy(se->cur_valid_map, raw_sit->valid_map, SIT_VBLOCK_MAP_SIZE);
    se->type = GET_SIT_TYPE(raw_sit);
    se->orig_type = GET_SIT_TYPE(raw_sit);
    se->mtime = le64_to_cpu(raw_sit->mtime);
}
```

Thus, at [1], we write to a user controlled offset within (or outside) a buffer that's also user controlled, along with more writes occurring further in the function, resulting in an out of bounds write on the heap.

Crash Information

```
SUMMARY: AddressSanitizer: SEGV /root/boop/f2fs/actually_normal_building/f2fs-tools-1.12.0/f2fs-tools-1.12.0/fsck/mount.c in
seg_info_from_raw_sit

Program received signal SIGSEGV, Segmentation fault.
[ Legend: Modified register | Code | Heap | Stack | String ]
──────── registers ────
$rax   : 0x0000617dcdcdd430  →  0x0000000000000000
$rbx   : 0x00007ffe8c0abda0  →  0x00007ffe8c0abd20  →  0xbebebebebebebebe
$rcx   : 0x0000617dcdcd02be  →  0x0000000000000000
$rdx   : 0x00007ffe8c0ab501  →  0x00ffffffffffffff
$rsp   : 0x00007ffe8c0abbd0  →  0x01000060000008f0
$rbp   : 0x00007ffe8c0abcf0  →  0x00007ffe8c0abfc0  →  0x00007ffe8c0ac1e0  →  0x00007ffe8c0ac430  →  0x00007ffe8c0ac6d0  →
0x000000000062c340  →  <__libc_csu_init+0> push r15
$rsi   : 0x0
$rdi   : 0x0000617dcdcdd401  →  0x0000000000000000
$rip   : 0x000000000057de75  →  <seg_info_from_raw_sit+517> mov WORD PTR [rax], cx
$r8    : 0x0
$r9    : 0x0
$r10   : 0x0
$r11   : 0x1
$r12   : 0x000000000041b300  →  <_start+0> xor ebp, ebp
$r13   : 0x00007ffe8c0ac7b0  →  0x0000000000000002
$r14   : 0x0
$r15   : 0x0
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow RESUME virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
────────── stack ────
0x00007ffe8c0abbd0|+0x0000: 0x01000060000008f0   ← $rsp
0x00007ffe8c0abbd8|+0x0008: 0x000061600000bebe  →  0x0000000000000000
0x00007ffe8c0abbe0|+0x0010: 0x00007ffe8c0abd21  →  0xbebebebebebebebe
0x00007ffe8c0abbe8|+0x0018: 0x00007ffe8c0abd20  →  0xbebebebebebebebe
0x00007ffe8c0abbf0|+0x0020: 0x01007ffebebebebe
0x00007ffe8c0abbf8|+0x0028: 0x00007ffe0000bebe
0x00007ffe8c0abc00|+0x0030: 0x00007ffe8c0abd21  →  0xbebebebebebebebe
0x00007ffe8c0abc08|+0x0038: 0x00007ffe8c0abd20  →  0xbebebebebebebebe
───────── code:x86:64 ────
0x57de68 <seg_info_from_raw_sit+504> call   0x4b5ea0 <__asan::__asan_report_store2(__sanitizer::uptr)>
0x57de6d <seg_info_from_raw_sit+509> mov    rax, QWORD PTR [rbp-0x38]
0x57de71 <seg_info_from_raw_sit+513> mov    cx, WORD PTR [rbp-0x2c]
→   0x57de75 <seg_info_from_raw_sit+517> mov    WORD PTR [rax], cx
0x57de78 <seg_info_from_raw_sit+520> mov    rdx, QWORD PTR [rbp-0x8]
0x57de7c <seg_info_from_raw_sit+524> cmp    rdx, 0x0
0x57de80 <seg_info_from_raw_sit+528> setne  sil
0x57de84 <seg_info_from_raw_sit+532> mov    rdi, rdx
0x57de87 <seg_info_from_raw_sit+535> and    rdi, 0x7
─ source:mount.c+1617 ────
1612  }
1613
1614  void seg_info_from_raw_sit(struct seg_entry *se,
1615                  struct f2fs_sit_entry *raw_sit)
1616  {
// se=0x00007ffe8c0abce8  →  [...]  →  0x0000000000000000, raw_sit=0x00007ffe8c0abce0  →  [...]  →  0xbebebebebebebebe
→ 1617         se->valid_blocks = GET_SIT_VBLOCKS(raw_sit);
1618         memcpy(se->cur_valid_map, raw_sit->valid_map, SIT_VBLOCK_MAP_SIZE);
1619         se->type = GET_SIT_TYPE(raw_sit);
1620         se->orig_type = GET_SIT_TYPE(raw_sit);
1621         se->mtime = le64_to_cpu(raw_sit->mtime);
1622  }
─ threads ────
[#0] Id 1, stopped 0x57de75 in seg_info_from_raw_sit (), reason: SIGSEGV
─── trace ────
[#0] 0x57de75 → seg_info_from_raw_sit(se=0x617dcdcdd430, raw_sit=0x7ffe8c0abd20)
[#1] 0x582daa → build_sit_entries(sbi=0x1188720 <gfsck>)
[#2] 0x583e96 → build_segment_manager(sbi=0x1188720 <gfsck>)
[#3] 0x59854d → f2fs_do_mount(sbi=0x1188720 <gfsck>)
[#4] 0x4f5cae → main(argc=0x2, argv=0x7ffe8c0ac7b8)
────
0x000000000057de75 in seg_info_from_raw_sit (se=0x617dcdcdd430, raw_sit=0x7ffe8c0abd20) at mount.c:1617
1617            se->valid_blocks = GET_SIT_VBLOCKS(raw_sit);
```

Timeline

2020-01-29 - Initial Contact

2020-02-03 - Vendor Disclosure

2020-03-20 - 60+ day follow up

2020-04-08 - Vendor acknowledged; Issue discovered as patched in 1.13 release but not disclosed as a security issue

2020-04-09 - Public Release

CREDIT

Discovered by Lilith [-,-] of Cisco Talos.

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