Talos Vulnerability Report

TALOS-2020-1047

F2fs-Tools F2fs.Fsck Multiple Devices Code Execution Vulnerability

OCTOBER 14, 2020

CVE NUMBER

CVE-2020-6105

Summary

An exploitable code execution vulnerability exists in the multiple devices functionality of F2fs-Tools F2fs.Fsck 1.13. A specially crafted f2fs filesystem can cause Information overwrite 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-73 - External Control of File Name or Path

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.

First it should be noted that the following vulnerability is completely an intended feature of f2fs in general. It's not entirely clear why this f2fs feature exists in the first place, but there are inherent risks within it, as will be explained. First, let us explore the feature. The quickest way to explain it is to examine the output of f2fs.fsck on a sample file:

```
Info: superblock features = 481 :  encrypt verity quota_ino
Info: superblock encrypt level = 0, salt = 00000000000000000000000000000000
Info: Device[0] : ./sample_f2fs.bin.patched blkaddr = 0--ce4dff  // [0]
Info: Device[1] : /proc/self/mem blkaddr = ce4e00--100ce2fff     // [1]
Info: total FS sectors = 108167128 (52815 MB)
```

A given F2fs filesystem can define up to eight different files that act as a backing to writes and reads for blocks. As shown above, for my given f2fs partition, two devices have been defined, `./sample_f2fs.bin.patched` at [0] and `/proc/self/mem` at [1]. Astute readers will note that `/proc/self/mem` isn't really a file that a program should be accessing, but moving on to where and why this occurs, let us examine the `f2fs_super_block` struct first:

```
struct f2fs_super_block {
    __le32 magic;           /* Magic Number */
    __le16 major_ver;       /* Major Version */
    __le16 minor_ver;       /* Minor Version */
    __le32 log_sectorsize;      /* log2 sector size in bytes */
    __le32 log_sectors_per_block;   /* log2 # of sectors per block */
    __le32 log_blocksize;       /* log2 block size in bytes */
    __le32 log_blocks_per_seg;  /* log2 # of blocks per segment */
    __le32 segs_per_sec;        /* # of segments per section */
    __le32 secs_per_zone;       /* # of sections per zone */
    __le32 checksum_offset;     /* checksum offset inside super block */
    __le64 block_count;     /* total # of user blocks */
    __le32 section_count;       /* total # of sections */
    __le32 segment_count;       /* total # of segments */
    __le32 segment_count_ckpt;  /* # of segments for checkpoint */
    __le32 segment_count_sit;   /* # of segments for SIT */
    __le32 segment_count_nat;   /* # of segments for NAT */
    __le32 segment_count_ssa;   /* # of segments for SSA */
    __le32 segment_count_main;  /* # of segments for main area */
    __le32 segment0_blkaddr;    /* start block address of segment 0 */
    __le32 cp_blkaddr;      /* start block address of checkpoint */
    __le32 sit_blkaddr;     /* start block address of SIT */
    __le32 nat_blkaddr;     /* start block address of NAT */
    __le32 ssa_blkaddr;     /* start block address of SSA */
    __le32 main_blkaddr;        /* start block address of main area */
    __le32 root_ino;        /* root inode number */
    __le32 node_ino;        /* node inode number */
    __le32 meta_ino;        /* meta inode number */
    __u8 uuid[16];          /* 128-bit uuid for volume */
    __le16 volume_name[MAX_VOLUME_NAME];    /* volume name */
    __le32 extension_count;     /* # of extensions below */
    __u8 extension_list[F2FS_MAX_EXTENSION][8]; /* extension array */
    __le32 cp_payload;
    __u8 version[VERSION_LEN];  /* the kernel version */
    __u8 init_version[VERSION_LEN]; /* the initial kernel version */
    __le32 feature;         /* defined features */
    __u8 encryption_level;      /* versioning level for encryption */
    __u8 encrypt_pw_salt[16];   /* Salt used for string2key algorithm */
    struct f2fs_device devs[MAX_DEVICES];   /* device list */
    __le32 qf_ino[F2FS_MAX_QUOTAS]; /* quota inode numbers */
    __u8 hot_ext_count;     /* # of hot file extension */
    __le16  s_encoding;     /* Filename charset encoding */
    __le16  s_encoding_flags;   /* Filename charset encoding flags */
    __u8 reserved[306];     /* valid reserved region */
    __le32 crc;         /* checksum of superblock */
} __attribute__((packed));
```

Taking out the definitions and information we actually care about:

```
struct f2fs_super_block {
    //[...]
    struct f2fs_device devs[MAX_DEVICES];   /* device list */
    //[...]
}

#pragma pack(push, 1)
struct f2fs_device {
    __u8 path[MAX_PATH_LEN];
    __le32 total_segments;
} __attribute__((packed));

#define MAX_PATH_LEN   64
#define MAX_DEVICES     8
```

To clarify, the `f2fs_super_block` object has an array of eight `f2fs_device` structs in it, each one being 0x44 bytes in length. The first 0x40 bytes determine the path of the backing device, and the last `__le32` member `total_segments` determines exactly which blocks the device backs. Let us examine the example output from above, and the f2fs partition that caused it:

```
Info: superblock features = 481 :  encrypt verity quota_ino
Info: superblock encrypt level = 0, salt = 00000000000000000000000000000000
Info: Device[0] : ./sample_f2fs.bin.patched blkaddr = 0--ce4dff
Info: Device[1] : /proc/self/mem blkaddr = ce4e00--100ce2fff   [0]
Info: total FS sectors = 108167128 (52815 MB)

# Current Layout =>
00000C90   00 00 00 00  00 00 00 00  00 2E 2F 73  61 6D 70 6C  ............/sampl [1]
00000CA0   65 5F 66 32  66 73 2E 62  69 6E 2E 70  61 74 63 68  e_f2fs.bin.patch
00000CB0   65 64 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ed..............
00000CC0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
#                       segment count== 0x6726  V[2]
00000CD0   00 00 00 00  00 00 00 00  00 26 67 00  00 2F 70 72  .............../pr
00000CE0   6F 63 2F 73  65 6C 66 2F  6D 65 6D 00  00 00 00 00  oc/self/mem.....
00000CF0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00000D00   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00000D10   00 00 00 00  00 00 00 00  00 00 00 00  00 F1 FF FF  ................
00000D20   FF 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................

[>_>]> p/x offsetof(struct f2fs_super_block, devs)
$3 = 0x899
```

Assuming the first superblock is valid, we start from offset 0x400 within the f2fs partition (otherwise 0x1400). Thus, our `devs` array starts at offset 0xc99 as shown above at [1]. At [2], we say that the first device `./sample_f2fs.bin.patched` covers 0x6726 segments, which results in 0xce4d00 blocks total (as shown in the output). The next device we define is backed by `/proc/self/mem` (or any other file we choose), and we say that it backs all the rest of the blocks, hence the output at [2]. Stopping now, there's not really a point in looking too deep into the `init_sb_info` function, but since it does read these devices off of disk and into memory, it's worth showing:

```
int init_sb_info(struct f2fs_sb_info *sbi)
{
    struct f2fs_super_block *sb = F2FS_RAW_SUPER(sbi);
    //[...]
                // 0x8
    for (i = 0; i < MAX_DEVICES; i++) {
        if (!sb->devs[i].path[0])   // [0]
            break;

        if (i) {
            c.devices[i].path = strdup((char *)sb->devs[i].path); // [1]
            if (get_device_info(i))                               // [2]
                ASSERT(0);
        } else {
            ASSERT(!strcmp((char *)sb->devs[i].path,
                        (char *)c.devices[i].path));
        }

        c.devices[i].total_segments = le32_to_cpu(sb->devs[i].total_segments); // [3]

        if (i){
            c.devices[i].start_blkaddr = c.devices[i - 1].end_blkaddr + 1;
        }
        c.devices[i].end_blkaddr = c.devices[i].start_blkaddr +   // [3]
            c.devices[i].total_segments * c.blks_per_seg - 1;

        if (i == 0)
            c.devices[i].end_blkaddr += get_sb(segment0_blkaddr);

        c.ndevs = i + 1;
        MSG(0, "Info: Device[%d] : %s blkaddr = %"PRIx64"--%"PRIx64"\n",
                i, c.devices[i].path,
                c.devices[i].start_blkaddr,
                c.devices[i].end_blkaddr);
    }

//[...]
```

At [1], there's a `strdup` from the device into memory, which is actually another bug if one strcpy's eight devices and the last member has no null bytes (since there's an oob read), however it's a pretty useless bug and not really worth noting. At [2], we see the validation that occurs on the filename, which essentially boils down to a `stat64` on the filename. At [3], the actual block definitions occur, one device begins where the previous one ends, etc.

Stepping back, what are the implications of this all? What does it mean for a specific block to be backed by a specific file? The best place to look would be the `dev_read_block` and `dev_write_block` functions:

```
int dev_read_block(void *buf, __u64 blk_addr) {
    return dev_read(buf, blk_addr << F2FS_BLKSIZE_BITS, F2FS_BLKSIZE);
}

int dev_write_block(void *buf, __u64 blk_addr) {
    return dev_write(buf, blk_addr << F2FS_BLKSIZE_BITS, F2FS_BLKSIZE);
}
```

It's important to note that all reads from disk to memory and all writes from memory to disk occur in `F2FS_BLKSIZE` chunks (0x1000), and as such, all reads and writes done by f2fs.fsck happen with these two functions. Now let us examine the inner functions, `dev_write` and `dev_read`:

```
int dev_read(void *buf, __u64 offset, size_t len)
{
    int fd;
    //[...]

    fd = __get_device_fd(&offset);
    if (fd < 0)
        return fd;

    if (lseek64(fd, (off64_t)offset, SEEK_SET) < 0)
        return -1;
    if (read(fd, buf, len) < 0)
        return -1;
    return 0;
}


int dev_write(void *buf, __u64 offset, size_t len)
{
    int fd;
    //[...]

    fd = __get_device_fd(&offset); // [1]
    if (fd < 0)
        return fd;

    if (lseek64(fd, (off64_t)offset, SEEK_SET) < 0) // [2]
        return -1;
    if (write(fd, buf, len) < 0) // [3]
        return -1;
    return 0;
}
```

As shown, they're essentially the same, with the sole difference being a `write` syscall versus a `read` syscall. At [1], an offset is found that we will examine soon, and at [2], we `lseek` to that offset. It should be noted that for the linux kernel, it doesn't seem like `lseek` can fail, no matter the offset given, while with the android kernel, `lseek` can return -1 if given a big enough value. Moving on to [3], either the read or the write occurs at the given offset provided. Looking at `__get_device_fd`:

```
static int __get_device_fd(__u64 *offset)
{
    __u64 blk_addr = *offset >> F2FS_BLKSIZE_BITS;       // [1]
    int i;

    for (i = 0; i < c.ndevs; i++) {                      // [2]
        if (c.devices[i].start_blkaddr <= blk_addr &&
                c.devices[i].end_blkaddr >= blk_addr) {
            *offset -= c.devices[i].start_blkaddr << F2FS_BLKSIZE_BITS;  // [3]

            return c.devices[i].fd;  // [4]
        }
    }
    return -1;
}
```

At [1], the offset we provide is shifted by 0xC bits, thus transforming it to which block we care about (i.e. offset 0x1123000 -> 0x1123). At [2], we iterate over all eight of our f2fs partition's devices, and if the block number falls within the boundaries of a given device, we return that file descriptor at [4]. Also of interest is [3], since the input offset gets transformed once again, and is turned into an offset from the beginning of the device which backs a given block. To make clearer, looking again at our example layout:

```
Info: Device[0] : ./sample_f2fs.bin.patched blkaddr = 0--ce4dff
Info: Device[1] : /proc/self/mem blkaddr = ce4e00--100ce2fff
```

If we have a read on address 0xce4e01000, the offset would shift over 0xc bits, resulting in a read from block 0xce4e01, which falls in the block range for the `/proc/self/mem` device. We then subtract the start block of `/proc/self/mem`, which results in us reading the first block (0xce4e01-0xce4e00 => 0x1), which then shifts back 0xC bits to tell us to seek to offset 0x1000 within `/proc/self/mem`. The resulting seek might succeed depending on the underlying kernel, but the read will assuredly fail, resulting in a assert probably being triggered.

Assuming that we have a way around ASLR, it could be possible to define a filesystem that can read and write to its own address space successfully via this method. Since f2fs.fsck uses the read and write syscalls, the program does not crash if it tries to read or write from unmapped memory within `/proc/self/mem`, however (with one exception) all calls to `dev_read_block` and `dev_write_block` are wrapped with `assert` calls to make sure the reads and writes succeed. An alternative exploitation method would be to just define an arbitrary other file on the filesystem to be read and written to. The file must be at least 0x1000 in size in order to succeed, but it becomes possible to alter other filesystems than the f2fs partition on fs-check.

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.

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

None - Public Release

CREDIT

Discovered by Lilith >_> of Cisco Talos
