Talos Vulnerability Report

TALOS-2020-1049

F2fs-Tools F2fs.Fsck dev_read Information Disclosure Vulnerability

OCTOBER 14, 2020

CVE NUMBER

CVE-2020-6107

Summary

An exploitable information disclosure vulnerability exists in the `dev_read` functionality of F2fs-Tools F2fs.Fsck 1.13. A specially crafted f2fs filesystem can cause an uninitialized read resulting in an 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-253 - Incorrect Check of Function Return Value

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.

When dealing with disk or file I/O, f2fs.fsck will always read and write in `F2FS_BLKSIZE` size chunks (0x1000) for performance's sake. We can see this in the two functions used for basically all disk reads and disk writes:

```
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);
}
```

In the above functions, `F2FS_BLKSIZE_BITS` is 0xC and `F2FS_BLKSIZE` is 0x1000, which we keep in mind as we examine the `dev_read` function.

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

    if (c.sparse_mode)  // never hit this.
        return sparse_read_blk(offset / F2FS_BLKSIZE,
                               len / F2FS_BLKSIZE, buf);


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

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

For the purposes of this writeup, `dev_write` is ignored, but they're essentially the same, with the sole significant 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], the read 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 it clearer, looking again at our example layout:

```
Info: Device[0] : ./sample_f2fs.bin.patched blkaddr = 0--ce4dff
Info: Device[1] : ./sample_f2fs.bin_2 blkaddr = ce4e00--100ce2fff
```

If I 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 `./sample_f2fs.bin_2` device. We then subtract the start block of `./sample_f2fs.bin_2`, 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 `./sample_f2fs.bin_2`. Assuming the `lseek64` and `read` syscalls fail, there's always assert statements making sure that `dev_read` returns 0 (with maybe one or two exceptions). But what does failure mean in terms of `fseek64` and `read` syscalls? For `lseek64`, there's actually different behavior depending on the kernel. In linux, `lseek64` doesn't seem to fail no matter what offset is provided, whereas on android `lseek64` will fail if given a really big value, which does cause different behavior for potential vulnerabilities (but not this one). It is important to note that `lseek64` does not fail on values that are past the end of the file, and if it's not too big a number, it does not fail on either linux or android.

The more interesting facet for our purposes is the return value of `read` syscall, which reads as such:

```
RETURN VALUE
       On  success,  the  number  of  bytes read is returned (zero indicates end of file), and the file position is advanced by this number.
It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer
       bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a
terminal), or because read() was interrupted by a signal.  See also NOTES.

       On error, -1 is returned, and errno is set appropriately.  In this case, it is left unspecified whether the file position (if any)
changes.
```

The most important thing out of the above is that `zero indicates end of file`. Which, if we review the `dev_read` code again:

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

    if (read(fd, buf, len) < 0) // [0]
        return -1;
    return 0;
}
```

At [0], we clearly see that `dev_read` only makes sure that the return value is not -1, which, if the file being read is at EOF, will not fail since `read` returns 0. This leaves us with the `void *buf` parameter that should be filled with disk contents now being filled with whatever was there before. In our case, that always means uninitialized heap data being re-used.

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.
