Talos Vulnerability Report

TALOS-2020-1132

Microsoft Azure Sphere mtd character device driver privilege escalation vulnerability

JULY 31, 2020

CVE NUMBER

CVE-2020-16982

Summary

An arbitrary flash write vulnerability exists in the mtd character device driver of Microsoft Azure Sphere 20.06. A specially crafted ioctl can bypass file permissions and allow writes to flash by unauthorized users. An attacker can issue a MEMWRITE ioctl to trigger this vulnerability.

Tested Versions

Microsoft Azure Sphere 20.06

Product URLs

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

CVSSv3 Score

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

CWE

CWE-284 - Improper Access Control

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. This is a partition (`/dev/mtd1`) that is mounted to `/mnt/config` and is meant to be writeable by applications.

Normally, applications can only write to their own directory in `/mnt/config`. This is a sample layout for the whole mount point:

```
# ls -l /mnt/config
total 1
drwx------    2 1001     1001             0 Jan  1  1970 7ba05ff7-7835-4b26-9eda-29af0c635280
-rw-r-----    1 sys      appman         341 Jan  1  1970 uid_map                                [1]

# ls -l 7ba05ff7-7835-4b26-9eda-29af0c635280
total 1
-rw-------    1 1001     1001            70 Jan  1  1970 network_interfaces
----------    1 root     root             0 Jan  1  1970 networkd_certs.cfg
-rwxrwx---    1 1001     1001           322 Jan  1  1970 wpa_supplicant.conf
```

As we can see above, this mountpoint is used by both applications and the underlying system in order to store non-volatile data. Each application gets a directory at `/mnt/config/<component_id>` in which it can write a limited amount of data, and in this case "7ba05ff7-7835-4b26-9eda-29af0c635280" is the `component_id` and storage space for `networkd`. At [1] we can also see the `uid_map`, a file which contains a map of Linux user ids and their corresponding Azure Sphere `component_id`. This file is used by `application-manager` in order to set unique user ids for each application when it is started, and here we provide an example `uid_map` to clarify:

```
# cat uid_map
1008
1001,7ba05ff7-7835-4b26-9eda-29af0c635280
1002,641f94d9-7600-4c5b-9955-5163cb7f1d75
1003,48a22e96-d078-4e34-9d7a-91b3404031da
1004,a65f3686-e50a-4fff-b25d-415c206537af
1005,89ecd022-0bdd-4767-a527-d756dd784a19
1007,11223344-1234-1234-1234-aabbccddeeff
```

This file is of course a prime target of modification by an attacker. If one were to change the UID for the app that had been compromised in order to get to this point, the application would then have a shinier UID. This assumes writability of the file however, and of course, the `uid_map` is normally not writable by applications, as seen in [1].

As previously demonstrated in TALOS-2020-1131, an attacker may be able to access any device in `/dev/` with read permissions. In this advisory we will show how it's possible to exploit read permissions in order to write to `/dev/mtd1`, targeting `uid_map` in particular and modify its contents.

Azure Sphere's kernel has a custom driver for their flash device, as shown in `drivers/mtd/chips/azure_sphere_flash.c`. In the function that looks for the flash device, we can see that some flags are set:

```
static struct mtd_info *azure_sphere_flash_probe(struct map_info *map)
{
    struct mtd_info *mtd;
    int err;
    struct azure_sphere_sm_flash_info flash_info;

    printk(KERN_INFO "Probing for flash devices via SK client\n");

   //[ ...]

    map->fldrv = &azure_sphere_flash_chipdrv;
    mtd->priv = map;
    mtd->name = map->name;
    mtd->type = MTD_NORFLASH;
    mtd->size = map->size;
    mtd->_erase = azure_sphere_flash_erase;
    mtd->_get_unmapped_area = azure_sphere_flash_unmapped_area;
    mtd->_read = azure_sphere_flash_read;
    mtd->_write = azure_sphere_flash_write;
    mtd->_panic_write = azure_sphere_flash_write;
    mtd->_sync = azure_sphere_flash_nop;
    mtd->flags = MTD_CAP_NORFLASH;                                      [1]
    mtd->_write_oob = azure_sphere_write_oob;
    mtd->_read_oob = azure_sphere_read_oob;
    mtd->ecc_strength = 0;
    mtd->writesize = flash_info.write.min_length;
    mtd->writebufsize = flash_info.write.max_length;
    mtd->erasesize = flash_info.erase.preferred_length;
```

The `MTD_CAP_NORFLASH` flag corresponds to `MTD_WRITEABLE | MTD_BIT_WRITEABLE`:

```
#define MTD_WRITEABLE       0x400   /* Device is writeable */
#define MTD_BIT_WRITEABLE   0x800   /* Single bits can be flipped */
...
#define MTD_CAP_NORFLASH    (MTD_WRITEABLE | MTD_BIT_WRITEABLE)
```

And this means that the flash device can be written to under certain circumstances. For confirmation, we can execute an `ioctl` call for `MEMGETINFO` against `mtd0` and `mtd1`, and we obtain the following flags:

```
mtd1: 0xc00 -> MTD_WRITEABLE | MTD_BIT_WRITEABLE
mtd0: 0x800 -> MTD_BIT_WRITEABLE
```

If we then look into the ioctl handler for MTD character devices (in `drivers/mtd/mtdchar.c`):

```
static int mtdchar_ioctl(struct file *file, u_int cmd, u_long arg)
{
    struct mtd_file_info *mfi = file->private_data;
    struct mtd_info *mtd = mfi->mtd;
    void __user *argp = (void __user *)arg;
    int ret = 0;
    u_long size;
    struct mtd_info_user info;

    pr_debug("MTD_ioctl\n");

    ...

    case MEMERASE:
    case MEMERASE64:                                        [2]
    {
        struct erase_info *erase;

        if(!(file->f_mode & FMODE_WRITE))
            return -EPERM;
        ...
    }
    ...
    case MEMWRITE:                                          [3]
    {
        ret = mtdchar_write_ioctl(mtd,
              (struct mtd_write_req __user *)arg);
        break;
    }
```

At [2] we see that the `MEMERASE` ioctl makes sure that the file has been opened for writing, while at [3] however, the `MEMWRITE` ioctl explicitly does not check `FMODE_WRITE` and instead calls `mtdchar_write_ioctl`:

```
static int mtdchar_write_ioctl(struct mtd_info *mtd,
        struct mtd_write_req __user *argp)
{
    struct mtd_write_req req;
    struct mtd_oob_ops ops;
    const void __user *usr_data, *usr_oob;
    int ret;

    if (copy_from_user(&req, argp, sizeof(req)))
        return -EFAULT;

    usr_data = (const void __user *)(uintptr_t)req.usr_data;
    usr_oob = (const void __user *)(uintptr_t)req.usr_oob;
    if (!access_ok(VERIFY_READ, usr_data, req.len) ||
        !access_ok(VERIFY_READ, usr_oob, req.ooblen))
        return -EFAULT;

    if (!mtd->_write_oob)
        return -EOPNOTSUPP;

    ops.mode = req.mode;
    ops.len = (size_t)req.len;
    ops.ooblen = (size_t)req.ooblen;
    ops.ooboffs = 0;

    if (usr_data) {
        ops.datbuf = memdup_user(usr_data, ops.len);
        if (IS_ERR(ops.datbuf))
            return PTR_ERR(ops.datbuf);
    } else {
        ops.datbuf = NULL;
    }

    if (usr_oob) {
        ops.oobbuf = memdup_user(usr_oob, ops.ooblen);
        if (IS_ERR(ops.oobbuf)) {
            kfree(ops.datbuf);
            return PTR_ERR(ops.oobbuf);
        }
    } else {
        ops.oobbuf = NULL;
    }

    ret = mtd_write_oob(mtd, (loff_t)req.start, &ops);           [4]

    kfree(ops.datbuf);
    kfree(ops.oobbuf);

    return ret;
}
```

This function does not do any kind of permission checks on the device, and eventually calls `mtd_write_oob` at [4].

```
int mtd_write_oob(struct mtd_info *mtd, loff_t to,
                struct mtd_oob_ops *ops)
{
    ops->retlen = ops->oobretlen = 0;
    if (!mtd->_write_oob)
        return -EOPNOTSUPP;
    if (!(mtd->flags & MTD_WRITEABLE))            [5]
        return -EROFS;
    ledtrig_mtd_activity();
    return mtd->_write_oob(mtd, to, ops);
}
```

In this function, only the `mtd` flags are checked, to make sure that the device is writable. However, nowhere in this code path has been checked if the file as been opened for writing (`FMODE_WRITE`), allowing anyone with read-only access to the device to issue a `MEMWRITE` ioctl.

Normally, one should erase the flash device before being able to write. As shown before, we are not able to use `MEMERASE` since we cannot open the file for writing. In reality however the “erase before writing” requirement depends on the underlying flash device and is usually simply a way to reset bits in a flash memory block from 0 to 1 (or vice-versa). This means that often times, it’s possible to still write to the device without erasing, but it will only be allowed to flip bits from 1 to 0 (or vice-versa). This clearly leaves room for exploitation.

In the context of Azure Sphere, while it would be technically possible to flip individual bits, because of the way littlefs utilizes the flash, the exploitation is simpler.
If the flash has not been entirely written to yet (after an erase has been performed), there will be many erased blocks (composed by all `0xff` bytes). In order to overwrite a file in littlefs, it’s enough to write a new “revision” of the file, with a version higher than the one before. When the file is requested, the littlefs driver will scan the device and pick the revision with the highest version. Theoretically, an attacker could also force erasing such blocks by either writing a big file containing only `0xff` (thus deleting older revision of files), or by corrupting the MTD device, which would cause the littlefs driver to erase the entire partition on reboot.

Putting it all together, an attacker that previously exploited TALOS-2020-1131 will have read access to the `mtd1` character device. By using this vulnerability, the attacker could issue a `MEMWRITE` ioctl and write a new revision for the `uid_map` file in an already erased block in `mtd1`. For example, the attacker could write:

```
1001,<component-id>
```

Where “component-id” is the id of an app under the attacker’s control. After a reboot (or a service crash), potentially from a vulnerability like TALOS-2020-1117, the component-id will have the specific UID assigned from the `uid_map`.

Because of restrictions in `application-manager`’s code, it’s not possible to simply set UID 0 to an arbitrary `component_id`, specifically because there is a check to make sure all UIDs listed in the `uid_map` are over 999. However, in TALOS-2020-1137 and TALOS-2020-1133 we will show how it’s possible to exploit this further and gain higher privileges.

Timeline

2020-07-28 - Vendor Disclosure
2020-07-31 - Public Release

CREDIT

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

VULNERABILITY REPORTS

PREVIOUS REPORT TALOS-2020-1131

NEXT REPORT TALOS-2020-1137