Talos Vulnerability Report

TALOS-2020-1131

Microsoft Azure Sphere ASXipFS inode type privilege escalation vulnerability

JULY 31, 2020

CVE NUMBER

None

SUMMARY

A privilege escalation vulnerability exists in the ASXipFS inode type functionality of Microsoft Azure Sphere 20.06. A specially crafted image package can cause access to arbitrary devices. An attacker can flash a malicious image package 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

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

CWE

CWE-668 - Exposure of Resource to Wrong Sphere

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.

Azure Sphere allows developers to flash new high-level applications by supplying an `imagepackage`. Normally, `imagepackage` files are created via the `azsphere` command line tool, and are simply an `asxipfs` image containing the application root with its binaries, `app_manifest.json`, and any optional dependency. Example:

```
$ tree .
.
├── app_manifest.json
└── bin
    └── app
```

When an application is installed, Azure Sphere is actually writing the provided `.imagepackage` file to the flash, after the due signature checks. Note that in order to be able to install applications on the device, a developer would need a device capability.

ASXipFS, namely Azure Sphere XIP file system, is a file system developed by Microsoft and based off CramFS. This newer filesystem has been created mainly to support execute-in-place directly from the flash, but also because the XIP patches for CramFS proper are not officially supported.

To start, inside the ASXipFS filesystem driver the `get_asxipfs_inode` function is used to handle all supported inode types for a given inode operation:

```
// `fs/asxipfs/asxipfs_inode.c`

static struct inode *get_asxipfs_inode(struct super_block *sb,
    const struct asxipfs_inode *asxipfs_inode, unsigned int offset)
{
    ...
    switch (asxipfs_inode->mode & S_IFMT) {
    case S_IFREG:
        // confirm this is a linear FS and that the XIP bit is set on the file
        if(!LINEAR(sbi) || !(asxipfs_inode->mode & S_ISVTX)) {
            printk(KERN_ERR "asxipfs: Non-linear or non-XIP asxipfs is not allowed.\n");
            return ERR_PTR(-EINVAL);
        }
        inode->i_fop = &asxipfs_linear_xip_fops;
        inode->i_data.a_ops = &asxipfs_aops;
        break;
    case S_IFDIR:
        inode->i_op = &asxipfs_dir_inode_operations;
        inode->i_fop = &asxipfs_directory_operations;
        break;
    case S_IFLNK:
        ...
        break;
    default:                                                [1]
        init_special_inode(inode, asxipfs_inode->mode,
                old_decode_dev(asxipfs_inode->size));
```

To summarize, `S_IFREG` corresponds to normal data-backed files, `S_IFDIR` to directories, and `S_IFLNK` to symlinks. The default case [1] however, calls `init_special_inode` (defined in `fs/inode.c`), which allows for additional `inode` types to be used:

```
void init_special_inode(struct inode *inode, umode_t mode, dev_t rdev)
{
    inode->i_mode = mode;
    if (S_ISCHR(mode)) {                                [2]
        inode->i_fop = &def_chr_fops;
        inode->i_rdev = rdev;
    } else if (S_ISBLK(mode)) {                         [3]
        inode->i_fop = &def_blk_fops;
        inode->i_rdev = rdev;
    } else if (S_ISFIFO(mode))
        inode->i_fop = &pipefifo_fops;
    else if (S_ISSOCK(mode))
        ;   /* leave it no_open_fops */
    else
        printk(KERN_DEBUG "init_special_inode: bogus i_mode (%o) for"
                " inode %s:%lu\n", mode, inode->i_sb->s_id,
                inode->i_ino);
}
EXPORT_SYMBOL(init_special_inode);
```

Effectively, this allows one to define character [2] and block [3] devices inside an ASXipFS image package which point to other existing devices already living elsewhere in the system (it should be noted however that in order to do so, we would need to build a custom asxipfs packer, since the standard `azsphere` command-line tool does not support packing `S_ISCHR`, `S_ISBLK`, or `S_ISFIFO` inodes).

Because block and character devices are simply referenced by major and minor numbers assigned to them, it is possible to create a block or character device inode that has any device as its root, essentially acting as a symlink to another device. This is the equivalent to being able to run `mknod` in the device itself, which usually requires root permissions due to the security implications. To give an example, we can create the `mtd1` character device in the `imagepackage`'s root directory (`mtd1` is the littlefs partition mounted to `/mnt/config`) using a `major,minor` of `90,2`:

```
> ls -l mtd1
crwxr-x--T    1 sys      1007       90,   2 Mar 20  1935 mtd1
> hexdump -C mtd1 | head -n1
00000000  15 00 00 00 f0 0f ff f7  6c 69 74 74 6c 65 66 73  |........littlefs|
```

Because permissions on our ASXipFS partition's permissions are always set to 750 by the Azure Sphere device itself on mount, and also since our application will always have the GID that's assigned to the ASXipFS partition, we have read access to the character device.
This read access allows not only for reading from arbitrary devices, but also allows executing ioctls against these arbitrary devices (which could be further utilized, as demonstrated in TALOS-2020-1132).

It is important to recognize that in order to actually trigger this vulnerability, one must be able to flash an application in the first place, which is quite a feat to begin with: an application signature bypass, an Azure Cloud vulnerability, or also a malicious actor with physical access in the supply chain would be required in order actually flash a `.imagepackage` file. And while we do not currently hold any of those, due to the severity of this in combination with TALOS-2020-1132, TALOS-2020-1137 and TALOS-2020-1133, we believed it appropriate to report since `AZURE_SPHERE_CAP_*` permissions are rather dangerous.

TIMELINE

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

CREDIT

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